MetaModelica – A Symbolic-Numeric Modelica Language and Comparison to Julia

Peter Fritzson Adrian Pop Martin Sjölund Adeel Asghar

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

Abstract

The need for integrating system modeling with advanced tool capabilities is becoming increasingly pronounced. For example, a set of simulation experiments may give rise to new data that are used to systematically construct a series of new models, e.g. for further simulation and design optimization. Such combined symbolic-numeric capabilities have been pioneered by dynamically typed interpreted languages such as Lisp and Mathematica. Such capabilities are also relevant for advanced modeling and simulation applications but lacking in the standard Modelica language. Therefore, this is a topic of long-running design discussions in the Modelica Design group. One contribution in this direction is MetaModelica, that has been developed to extend Modelica with symbolic operations and advanced data structures, while preserving safe engineering practices through static type checking and a compilation-based efficient implementation. Another recent effort is Modia, implemented using the Julia macro mechanism, making it dynamically typed but also adding new capabilities. The Julia language has appeared rather recently and has expanded into a large and fast-growing ecosystem. It is dynamically typed, provides both symbolic and numeric operations, advanced data structures, and has a just-in-time compilation-based efficient implementation. Despite independent developments there are surprisingly many similarities between Julia and MetaModelica. This paper presents MetaModelica and its environment as a large case study, together with a short comparison to Julia. Since Julia may be important for the future Modelica, some integration options between Modelica tools and Julia are also discussed, including a possible approach for implementing MetaModelica (and OpenModelica) in Julia.

Keywords: Modelica, MetaModelica, symbolic, Julia, meta-programming, language, compilation

1 Introduction

Advanced development of today's complex products requires integrated environments and equation-based object-oriented declarative languages such as Modelica (Fritzson, 2014; Modelica Association, 2017) for modeling and simulation. Such combined symbolic-numeric capabilities and advanced data structures have been pioneered by dynamically typed interpreted languages such as Lisp (Steel, 1993) and Mathematica (Wolfram, 2003), but are also relevant for modeling and simulation applications. Therefore, this is a topic of design discussions in the Modelica Design group regarding the future Modelica, and has also motivated the development of MetaModelica (Fritzson et al 2005; Pop et al, 2006, Fritzson et al, 2011) and Modia (Elmqvist et al, 2016; Elmqvist et al 2017);

1.1 Motivation and Design Goals

At the time when the MetaModelica effort was started, MetaModelica 1.0 (Fritzson et al 2005), there was no existing efficiently compiled language that combined strong numeric and symbolic capabilities. Our vision was to extend Modelica in that direction via MetaModelica, in a backwards compatible way, supporting the Modelica design goals of safe engineering practices through static type checking, and explicitly declared types for increased model readability and efficient compilation. In the longer term the goal was an efficient interactive environment based on incremental compilation or just-in-time compilation (Section 8.4).

However, in the meantime the rather young language Julia (Bezanson et al 2017; Julialang 2018) has matured, (Julia 1.0 was released in August 2018), with similar design goals of an efficiently compiled interactive symbolic-numeric language. However, also with the goals of dynamic typing and automatic interfacing with libraries in other languages, and no special requirement of integrating with the Modelica modeling language.

The design of MetaModelica has been mostly influenced by Modelica, Standard ML (Milner et al, 1997) and RML (Pettersson 1989), whereas Julia has been more influenced by dynamic languages such as Lisp and Mathematica.

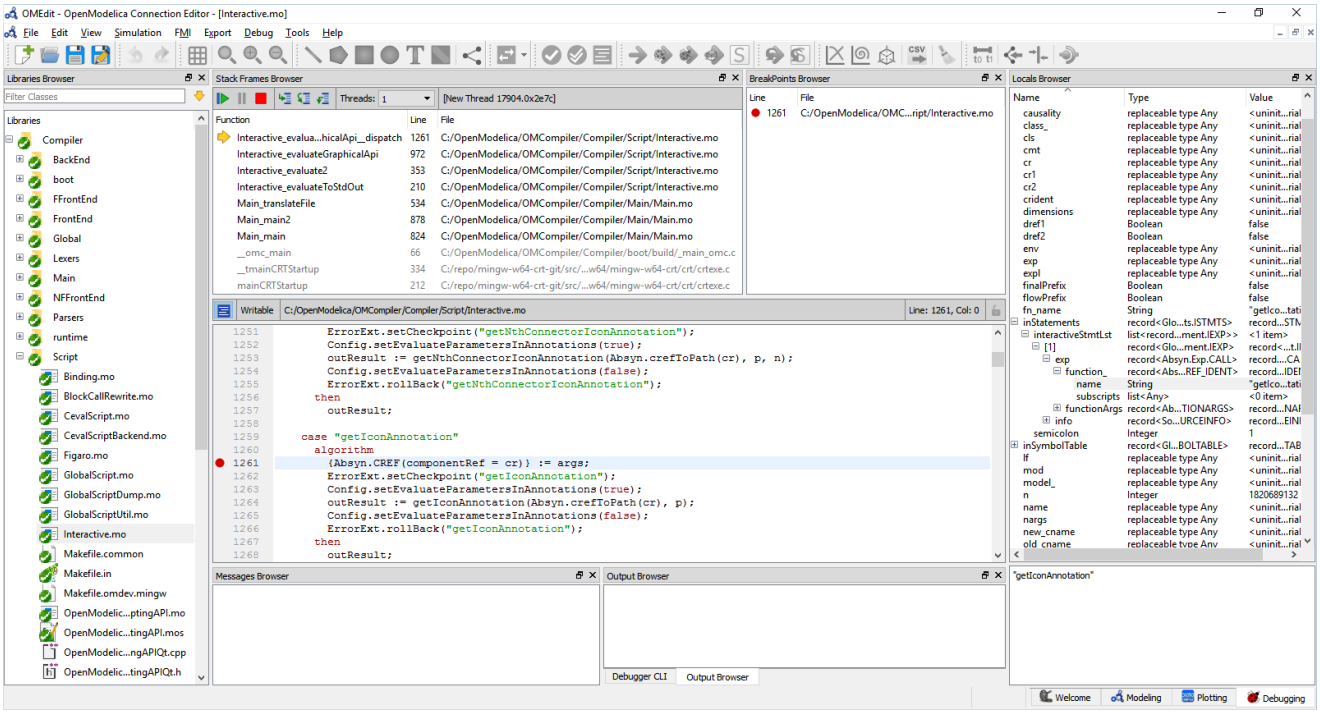

Figure 1. The integrated MetaModelica OMEdit-based development environment in debugging mode. *Left*: the package browser. *Top*: the active stack frames (including C routines) and breakpoints. *Middle*: text editing and breakpoint setting. *Right*: the local variables browser. The user can switch to modeling mode which has both textual and graphical editing.

The advent of Julia has changed the situation, as pointed out by (Elmqvist et al, 2016). Julia is a very capable and efficient symbolic-numeric language available with a rapidly growing ecosystem and set of libraries. Thus, it seems likely that Julia will influence the future of Modelica. The Modia prototype in Julia demonstrates several advantages but lacks support for safe engineering practices via static type checking. In Section 8 we briefly discuss ways of integrating Modelica tools with Julia without losing the Modelica static type checking.

A further discussion of related work is available in Section 10.

In the following, when we mention MetaModelica, we usually also include Modelica, since MetaModelica is an extension of Modelica.

1.2 Contributions

The contributions of this paper are not about inventing new language constructs. The introduced constructs have already been well proven in several other languages. Similar statements have been made regarding the Julia language. However, in the context of Modelica there are contributions on integrating such constructs into the Modelica language including the Modelica type system in a backwards compatible way.

Another contribution is the comparison of Julia and MetaModelica, showing many similarities and how Julia-like features have been integrated into the Modelica language via MetaModelica.

There are also contributions in the form of the very large case study of implementing the OpenModelica compiler in MetaModelica in an efficient way, and using the language and the associated developed environment (Figure 1, Section 8) for this large effort. Large case studies are valuable from a scientific point of view since it is often the case that results from investigations of small toy problems may not be true when problem sizes are scaled up.

1.3 Paper Organization

This paper is organized as follows.

Section 2 compares basic properties of MetaModelica and Julia. Sections 3 and 4 introduces uniontypes, tree and list data structures. Section 5 presents pattern matching including a symbolic example. Section 7 discusses compiler performance.

Section 8 presents the new OMEdit-based development environment for MetaModelica 3.0 and gives a comparison to the Eclipse-based MDT plug-in

Section 9 discuses integration of Modelica tools with Julia, whereas Section 10 presents related work and makes a short comparison to functional languages and languages such as Julia and Python. Finally, Section 11 gives conclusions and future work.

DOI 10.3384/ecp19157289

2 Some Properties of MetaModelica and Julia

We start by briefly summarizing and comparing some basic properties of MetaModelica and Julia.

2.1 Syntax

The syntax of the MetaModelica extension of Modelica is strongly influenced by Modelica, and to a lesser extent by Standard ML and C++. The Julia syntax is more influenced by languages like Python. Both are influenced by Matlab. The Julia syntax is more concise whereas the MetaModelica syntax is more verbose and descriptive, with more keywords.

2.2 Type System and Dynamic/Static Typing

MetaModelica/Modelica is structurally typed with some nominal typing parts, whereas Julia has a completely nominal type system. Thus, in Julia, concrete types may not be subtypes of each other. Both languages have concrete and abstract types, and parameterized types.

MetaModelica is a statically typed language; there are rules for determining the type of every expression in the program. Conversely, Julia is dynamically typed, types are properties of data values, and are dynamically created at runtime and implied by the way data flows through the program during execution. In static languages *expressions* have types, in dynamic languages *values* have types.

However, Julia has a rather sophisticated language for describing types, and it is possible to annotate expressions with types. For example, in Julia, `z::T` is an assertion that `z` is a value of type `T`; if that is true, `z::T` evaluates to the value of `z`, otherwise an error is raised. Type annotations in function signatures are slightly different: instead of asserting the type of an existing value, they indicate that the function only applies if the corresponding argument is of the indicated type.

To summarize, MetaModelica/Modelica is static, structural, and parametric, whereas Julia is dynamic, nominal, and parametric.

2.3 Multiple Dispatch and Overloading

Overloading of an operator or function means that in the presence of multiple implementations/definitions the definition with matching argument types is selected.

For some reason Julia has chosen to change from the well-established *overloading* terminology to instead use the term *multiple dispatch*. The new term might be more descriptive, but this change may cause some initial confusion for users. There is some arguing that dynamic selection is a reason for the new term, but one could instead talk about dynamic overloading.

MetaModelica provides user-defined overloading of both functions and operators, whereas standard Modelica only provides operator overloading. In both cases the selection is made at compile time based on statically available types of argument expressions. In Julia, the selection is done either at compile-time if the type can be inferred by the compiler, or at run-time based on runtime type tags of argument values.

3 Tree Data Structures

What are the needs for data structures and operations for symbolic (meta-programming) capabilities? One of the most common examples of programs that manipulate and produce other programs are compilers, which translate programs in some language into the same or another language. A small symbolic manipulation example is presented in Section 5.3.

The most common data type representation for programs in compilers are tree structures, and typical operations are transformations of such trees into trees during the translation process. Lists are a special case of tree data types but are typically given special support in many symbolic programming languages.

Tree data types have two interesting properties:

- Uniontype – a tree data type is typically the union of a number of node types, each representing a tree node.
- Recursive type – the children of a tree node may a type which is the tree data type itself.

Below we describe the MetaModelica uniontype language extension, give some examples of its usage, and briefly compare to Julia.

3.1 Uniontypes

The uniontype MetaModelica construct is a restricted class that can be viewed as the union of the record classes it contains. The keyword `uniontype` is followed by the name of the uniontype, in the example below called `Exp`

A record type belonging to a uniontype is called a union member record.

This example shows a small expression tree using uniontype `Exp` containing six different node types represented as Modelica record types, which must be declared within the scope of the union type. The `uniontype` restricted class construct has been extensively used in a Modelica context.

```
uniontype Exp
 record RCONST   Real rval;    end RCONST;
 record INTconst Integer exp1; end INTconst;
 record ADDop Exp exp1; Exp exp2; end ADDop;
 record SUBop Exp exp1; Exp exp2; end SUBop;
 record MULop Exp exp1; Exp exp2; end MULop;
 record DIVop Exp exp1; Exp exp2; end DIVop;
 record NEGop Exp exp1;           end NEGop;
end Exp;
```

The uniontype class grammar is as follows:

```
class_prefixes :
[ partial ]
```

```
( class | model | [ operator ] record |
block | [ expandable ] connector
| type | package | [ ( pure | impure ) ] [
operator] function | operator | uniontype)
```

The uniontype construct is used by functional languages such as OCAML, Standard ML, Haskell, etc. In several of these languages the uniontype construct is called *datatype*.

Uniontypes are also very common in Julia. However, in Julia the uniontypes are constructed dynamically at run-time since they are properties of values, not of expressions. Uniontypes in Julia can also be named and explicitly defined using the `Union` keyword:

```
IntOrString = Union{Int,AbstractString}
```

3.2 Main Properties of Uniontypes

The MetaModelica uniontype construct is a restricted class with the following main properties:

- Uniontype elements can be record declarations, replaceable type declarations declared using keywords replaceable type, *only* allowed to be used for type parameterization of the member records (and function(s)) and *not* to introduce uniontype member records. A record type declared within a uniontype is called a *uniontype member record*.
- Uniontypes can be *recursive*, i.e., reference themselves. That is the case in the above `Exp` example, where `Exp` is referenced inside its member record types.
- The typing rules for a uniontype are similar to operator records, i.e., nominal typing comparing type names. To check subtyping, (currently type identity) of two uniontypes, it is tested whether they belong to a subtype with the same name.
- Uniontypes can be parameterized by other types, using replaceable, similar to other restricted classes in Modelica.
- Inheritance, `extends`, between uniontypes is currently not allowed. The reason is that all issues for efficient implementation of such a feature are not yet resolved.
- Inheritance between member records is allowed e.g. `record ADDop2 = ADDop;` or using the long form: `record ADDop2 extends ADDop; … end ADDop2`
- Uniontypes provides a type-safe mechanism for variant records.

3.3 Calling Member Record Constructors

A *uniontype member record* constructor can be called using function syntax similar to standard record constructors, where the uniontype name is prefixed to the member record name to disambiguate:

```
UnionTName.MemberRecord()
```

If the union type is imported into a scope, the uniontype name prefix is not needed, for example:

```
import UnionTName.*;
MemberRecord()
```

For example, to construct the small expression tree of Figure 2 below using the above `Exp` uniontype without importing, the following would be needed:

```
Exp.ADDop(Exp.RCONST(12), Exp.MULop(
Exp.RCONST(5), Exp.RCONST(13)))
```

If importing of `Exp` into the current scope is used, the expression becomes more concise:

```
import Exp.*;
ADDop(RCONST(12), MULop( RCONST(5),
RCONST(13)))
```

3.4 A Small Expression Tree Example

A small expression tree, of the expression 12+5*13, is depicted in Figure 2.

Using the `Exp` record constructors `ADDop`, `MULop`, `RCONST`, this tree can be constructed by the expression ADDop(RCONST(12), MULop(RCONST(5), RCONST(13)))

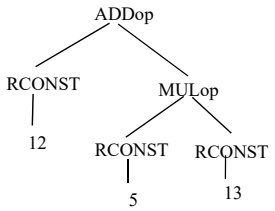

Figure 2. Abstract syntax tree of the expression 12+5*13.

3.5 Supertype Any

The predefined type `Any` is a supertype of any other MetaModelica type, i.e., all other MetaModelica types are subtypes of `Any`.

Since all other types are subtypes of `Any`, by using `Any` in a replaceable type declaration, it is possible to avoid any constraints and provide full flexibility in using any type as a type parameter in the following replaceable type declaration:

```
replaceable type TypeParam = Any
  constrainedby Any;
```

This is equivalent to the following, since the default type is used as constraining type if that is missing:

```
replaceable type TypeParam = Any;
```

The type `Any` is also present in Julia, with the same semantics that it is a supertype of all other types.

3.6 Predefined Uniontype Option for Optional Values

The predefined MetaModelica `Option` uniontype provides a type-safe way of representing the common situation where a data item is optionally present in a data structure.

The constructor `NONE()` is used to represent the case where the optional data item is not present, whereas the constructor `SOME()` is used when the data item is present in the data structure.

DOI 10.3384/ecp19157289

The following is a definition of the parameterized `Option` uniontype with a type parameter:

```
uniontype Option
 replaceable type TypeParam = Any
   constrainedby Any;
 record NONE
 end NONE;
 record SOME
   TypeParam elem;
 end SOME;
end Option;
```

For example, a `StringOption` type and a function using it are defined:

```
uniontype StringOption = Option(redeclare
  TypeParam=String);
function stringOrDefault
  input StringOption strOpt;
  input String default;
  output String str;
algorithm
  str := match strOpt
    case Option.SOME(str) then str;
    else default;
  end match;
end stringOrDefault;
```

Calling the function a few times:

```
stringOrDefault(Option.NONE(),"default")
 "default"
stringOrDefault(Option.SOME("string"),
   "default")
"string"
```

A similar predefined facility is available in Julia. Declaring a function argument or a record field as having the type `Union{T, Nothing}` allows setting it either to a value of type `T`, or to `nothing` to indicate that there is no value.

3.7 Parameterized Union Types

Parameterized union types with opaque type parameters are available. This means that only minimal information about the type parameter is needed.

There is also support for `redeclare` in cases where only information about sorting order needs to be available about the type used as type parameter. For example, this sorting order is provided by the type `Key` given by the function `keyCompare` in the `AvlSetString` package available in the OpenModelica utility library.

```
package AvlSetString
  import BaseAvlSet;
  extends BaseAvlSet;
  redeclare type Key = String;
  redeclare function extends keyStr
    algorithm
     outString := inKey;
    end keyStr;
  redeclare function extends keyCompare
    algorithm
     outResult :=
       stringCompare(inKey1, inKey2);
    end keyCompare;
end AvlSetString;
```

4 Lists and Tuples

List and tuple data types are common in many languages used for meta-programming and symbolic programming, and are available in both MetaModelica and Julia.

4.1 Lists

The following MetaModelica operations allows creation of lists and addition of new elements in front of lists in a declarative way, i.e., such lists are immutable. Extracting elements is done through pattern-matching in match-expressions.

- `list` – `list(el1,el2,el3, ...)` creates a list of elements of identical type. Examples: `list()` is the empty list, `list(2,3,4)` is a list of integers.
- `::` – the `::` operator in the expression `element::lst` adds an element in front of the list `lst` and returns the resulting list.

The types of lists and list variables can be specified as follows:

- `list` – `list<type-expr>` using angle-bracket notation is a `list` *type constructor*, e.g.:
  ```
  type RealList = list<Real>;
  ```
- Direct declaration of a variable `rlist` that denotes a list of real numbers:
  ```
  list<Real>   rlist;
  ```

A list type is a parametrized uniontype; the Option type is also such a type. The only addition is the `::` operator.

Lists are available in Julia with about the same semantics and similar but slightly different syntax.

4.2 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 can be created using the `tuple` constructor instead of the short-hand parentheses notation: `tuple(a)`
- A tuple can be seen as being returned from a function with multiple results in standard Modelica:
  ```
  (x,y,z) := foo(var, 2, 3, 5);
  ```
- Access of field values in tuples can be achieved via pattern-matching, e.g. the following will extract the three field values from a tuple value:
  ```
  (x,y,z) := tuplevalue
  ```

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;
```

Tuples are also available in Julia, with the same syntax (parenthesized list) and semantics. Tuple types can also be defined explicitly in Julia using the `Tuple` keyword:

```
Tuple{Ident,Int}
```

5 Match Expressions for Processing Complex Data

Matching on instances of structured data types such as trees is one of the central facilities in symbolic processing languages. The matching provided by the match-expression construct is very close to similar facilities in many functional languages but is also related to switch statements in C or Java. Match-expressions have two important advantages over traditional switch statements e.g. available in languages such as C or Java:

- A match-expression can appear in any of the three Modelica contexts: expressions, statements, or in equations.
- The selection in the case branches is based on pattern matching, which reduces to equality testing or switch in simple cases but is much more powerful in the general case.

Regarding allowed patterns used in match-expressions they are defined by the pattern language, see Section 5.2. For example, constants can be patterns, e.g., `"one"`, `384`, `RequirementStatus.violated`. Constructors with or without pattern variables can be patterns. The wildcard pattern _ (underscore) matches anything.

A very simple example of a match-expression is the following code fragment, which returns a number corresponding to a given input string. The pattern matching is very simple – just compare the string value of s with one of the constant pattern strings `"one"`, `"two"` or `"three"`, and if none of these matches return 0 since the wildcard pattern _ matches anything.

```
  String s;
  Real   x;
algorithm
  x := match s
    case "one"   then 1;
    case "two"   then 2;
    case "three" then 3;
    case   _     then 0;
  end match;
```

Alternatively, an else-branch, `else 0;`, can be used instead of the last wildcard pattern `case _ then 0`:

Another, more useful example, but still trivial since it only shows constants, is a match expression converting an enumeration value to a Boolean value:

```
type RequirementStatus =
  enumeration(violated, undecided, satisfied);
function RequirementStatusToBoolean
   input  RequirementStatus r;
   input  Boolean undecided = false;
   output Boolean b;
algorithm
  b = match r
   case RequirementStatus.violated  then false;
   case RequirementStatus.undecided then
                                      undecided;
   case RequirementStatus.satisfied then true;
  end match;
end RequirementStatusToBoolean;
```

The match expression in the above conversion function gives the same result as the following if-expression, but can be compiled more efficiently (Section 5) and is easier to follow:

```
  b = if r == RequirementStatus.violated
        then false
      elseif r == RequirementStatus.undecided
        then undecided
      else true;
```

The general syntactic structure of match-expressions starting with the `match` keyword is indicated by the syntax outline below. The else-branch is optional and is identical to a `case _` branch. Local equation sections contain equations, local algorithm sections contain statements. The syntax outline:

```
match <match-value-expr> <opt-local-decl>
    ...
  case <pat-expr>
    [equation | algorithm]
      <opt-equations-or-statements>
    then <expr>;
    ...
  case <pat-expr>
    [equation | algorithm]
      <opt-equations-or-statements>
    ...
  else
    [equation | algorithm]
      <opt-equations-or-statements>
end match;
```

A slightly more advanced usage of match-expressions compared to the above trivial cases is in a small expression evaluator, the function `eval`. Here we use as-binding of the result of a match to `x`, and standard Modelica dot-notation to access values, e.g. `x.rval` or `x.exp2`. The constructor pattern notation with empty parentheses, e.g., `ADDop()`, means matching with arbitrary arguments to that constructor.

```
function eval
  input  Exp  inExpression;
  output Real result;
  import Exp.*;
algorithm
 result := match x as inExpression
  case RCONST() then x.rval;
```


DOI 10.3384/ecp19157289

```
  case ADDop() then eval(x.exp1)+eval(x.exp2);
  case SUBop() then eval(x.exp1)-eval(x.exp2);
  case MULop() then eval(x.exp1)*eval(x.exp2);
  case DIVop() then eval(x.exp1)/eval(x.exp2);
  case NEGop() then -eval(x.exp);
 end match;
end eval;
```

Without the `import Exp.*` clause, constructors would need the `Exp` prefix, e.g. `Exp.RCONST()`, `Exp.ADDop`.Regarding Julia, there are several third-party libraries available for pattern matching is available with a semantics very close to the abovementioned match expression construct. Match.jl (Squire 2013) and Rematch.jl (RelationalAI 2018) use the following syntax:

```
@match item begin
    pattern1              => result1
    pattern2, if cond end => result2
   pattern3 || pattern4   => result3
   _                      => default_result
End
```

To make its semantics even closer to MetaModelica match we included the Rematch.jl Julia package in our prototype MetaModelica.jl compatibility layer and enhanced it to also include named pattern matching (Section 5.1) and `matchcontinue` semantics (pattern matching with exception handling, see Fritzson et. al 2011 for details). To implement match and `matchcontinue` semantics requires only 350 lines of Julia code. See also Section 9.

5.1 Named Pattern Matching with Pattern Variables vs Positional Matching

Named pattern matching uses named association to match/bind pattern variables to values of corresponding named arguments (e.g., record field names) of constructors.

This notation is more verbose than that for positional pattern matching but has the advantages that it is more robust against model changes such as constructor argument order, invention and maintenance of pattern variable names is avoided, and usually increased readability since the argument names are visible, especially if there are many arguments.

This example is of an `ADDop` named pattern mentioning field names `exp1` and `exp2` with pattern variables `e1` and `e2` which become bound to values during matching.

Named pattern matching is possible, i.e., the position of the pattern variable does not matter, only the field name (below `exp1` or `exp2`) which it is associated to:

```
case ADDop(exp1=e1,exp2=e2)
   then eval(e1) + eval(e2);
```

In positional pattern matching this case would appear as follows. It is more concise but dependent on argument order:

```
case ADDop(e1,e2) then eval(e1)+eval(e2);
```

5.2 Pattern Expressions

Pattern expressions are used in match expressions and can have the following forms:

- Patterns can contain literal constants of strings, integers, real numbers, Booleans, enumeration values, e.g. "`string`", `555`, `3.14`, `true`, `false`, `Sizes.medium`.
- Patterns can contain the _ wildcard which matches one item of anything.
- A pattern can be a pattern variable, i.e., an identifier, which can appear as an argument to a constructor, and which matches one item of anything.
- A pattern variable need not be declared. Its type is inferred using simple type inferencing, e.g. from the corresponding formal parameter type when it appears as an argument to a record constructor.
- A pattern variable is automatically introduced into the local scope, e.g. a case-clause, where the variable is first mentioned. Therefore, it shadows variables with the same name in outer scopes.
- A pattern variable is bound to the value it matches during pattern matching.
- The same pattern variable may occur at most once in the main part of the pattern expression, i.e., excluding the optional guard part.
- Patterns can contain calls to *record constructor* functions, *not* to other kinds of functions except constructors such as the array constructor `array`(), the array function `cat`(), the `list`() constructor or the `tuple`() constructor.
- Positional and/or named argument constructor call syntax can be used in patterns containing constructors, e.g., the positional call `FOO(1,_,2)`, is allowed; a named argument call version, e.g., `FOO(field1=1,field3=2)`, or `FOO(field1=1,field3=myvar)`, where `myvar` is a pattern variable, is also allowed. Moreover, you can mix positional and named arguments in the call pattern, with positional arguments first: `FOO(1,field3=myvar)`.
- A constructor pattern `NAME`(…) can have an unspecified argument list denoted by an empty argument list as in `FOO()`. This matches the corresponding constructor, here `FOO`, with arbitrary (zero or more) arguments.
- A constructor pattern `NAME`(…) is interpreted as implicitly filling unspecified argument patterns _ at the end of the argument list until it matches the declared number of arguments of the constructor; in the case of `array`(…) matching arbitrary (zero or more) arguments after the specified arguments. For example, a constructor `R` with three members x, y, z, would fit all of the following patterns: `R()`, `R(v1)`, `R(v1,v2)`, `R(v1,v2,v3)`.

- Patterns can contain curly-brace array constructors, which match exactly those elements mentioned, e.g., `{}`, `{3,5}`, {3,5,_}, {3,5,6}, {a,5}. The array pattern {} matches an empty array value.
- Patterns can contain the `as` binding operator, [*e.g.* state1 `as FOO(env,…)`.
- Patterns may optionally have guards, i.e., conditional expressions that are evaluated at run-time and are part of the pattern condition, i.e., if the whole matching fails including the guard, the match may try another pattern if present. *Example*: `case REAL() guard x.value > 0 then x.value`.
- Currently the MetaModelica pattern language does *not* support explicit and/or combinations of patterns, e.g. `pattern1 and pattern2`, `pattern1 or pattern2` whereas the Rematch.jl code was influenced by Scala and does support this. The and-mechanism can be achieved by embedding two or more patterns in a list of patterns, e.g. `{pattern1, pattern2}`, whereas the or-mechanism can be achieved by having two case-rules, e.g., `case pattern1 …; case pattern2 … .`

Some pattern examples:

```
"a"          // constant literal string pattern
33           // constant literal Integer pattern
3.14         // constant literal Real pattern
false        // constant literal Boolean pattern
true         // constant literal Boolean pattern
p            // pattern variable pattern, name p
Sizes.medium // literal enumeration pattern
ADDop()      // constructor pattern with zero
             // or more arbitrary arguments
ADDop(3)     // constructor pattern, first is 3,
             // followed by arbitrary args
ADDop(_,_)   // constructor pattern with 2 or
             // more arbitrary arguments
ADDop(p,_)   // constructor pattern with 2 or
     // more arbitrary arguments, the first
     // argument bound to pattern variable p
(_,_)       // tuple pattern with 2 arguments
list(_,_)   // list pattern with >=2 arguments
x :: rest   // list pattern where x matches the
 // first element and rest the rest of the list
array(_)    // array pattern with one or more
            // arbitrary arguments
array(3,4) // array pattern with the first
           // two elements being 3 and 4,
cat(1, {head}, rest) // Pattern which matches
 // both the head (first element) and
 // the rest (remaining elements) of an array
array()// array pattern, >= zero arguments
{_,_}  // array pattern, exactly two elements
{_,55} // array pattern; two elements, 2nd 55
{44}   // array pattern; one element being 44
{}     // array pattern, zero elements
{33,_} // array pattern,  two elements, 1st 33
{_,33,_,44} // array pattern with four
          // elements, the 2nd is 33, 4th is 44
```

Syntax rule:

```
pattern : expression
```

The pattern expression syntax is a subset of the general expression syntax. This is checked by semantics rules. The syntax looks slightly different in the Rematch.jl package, but all of the semantics are supported (and has some additional semantics as well).

5.3 Symbolic Differentiation Example

Symbolic differentiation of expressions is a symbolic operation that transforms expressions into differentiated expressions.

```
uniontype Exp
  record RCONST  Real e1;      end RCONST;
  record ADD Exp e1; Exp e2;   end ADD;
  record SUB Exp e1; Exp e2;   end SUB;
  record MUL Exp e1; Exp e2;   end MUL;
  record DIV Exp e1; Exp e2;   end DIV;
  record NEG Exp e1;           end NEG;
  record IDENT   String name;  end IDENT;
  record CALL Exp id; Exp[:]args;end CALL;
  record AND Exp e1; Exp e2;   end AND;
  record OR  Exp e1; Exp e2;   end OR;
  record LESS Exp e1; Exp e2;  end LESS;
 record GREATER Exp e1;Exp e2;end GREATER;
end Exp;
```

An example function `df` performs symbolic differentiation of the expression `expr` with respect to the variable `time`, returning a differentiated expression.

As previously mentioned, in the patterns _ is a reserved word that can be used as a placeholder instead of a pattern variable when the particular value in that place is not needed later as a variable value. The **as**-construct is used to bind the additional identifier to the matched value of the relevant expression.

In the following example the _ is used as a placeholder of any argument in one of the patterns, `CALL(IDENT("sin"),{_})`. This is a function call to sin with the argument list being an array of exactly one element `{_}`. The example also uses constructors with empty parentheses like `ADD()` to match for zero or more arguments with any contents.

The following well-known derivative rules are represented in the match-expression code:

- The time-derivative of a constant `RCONST()` is zero.
- The time-derivative of the time variable is one.
- The time-derivative of a time dependent variable `id` is `der(id)` but is zero if the variable is not time dependent, i.e., not in the list `tv/timevars`.
- The time-derivative of the sum `add(x.e1,x.e2)` of two expressions is the sum of the expression derivatives.
- The time-derivative of sin(x) is cos(x)*x' if x is a function of time, and x' its time derivative.

Some operators have been excluded in the `df` example below:

```
function df "Symbolic differentiation of
expression with respect to time"
  input  Exp expr;
  input  String[:] tv;
  output Exp diffexpr;
  import Exp.*;
```


DOI 10.3384/ecp19157289

```
algorithm
  diffexpr := match x as expr
   // der of constant
  case RCONST() then RCONST(0.0);
   // der of a variable
  case IDENT() then
      if x.id == "time" then RCONST(1.0)
      // der of time variable
      else if member(x.id,tv)
      // der of any variable id
        then CALL(IDENT("der"),{x.id})
      else RCONST(0.0);
  // (x.e1 + x.e2)' => x.e1' + x.e2'
  case ADD() then
    ADD(df(x.e1,tv), df(x.e2,tv));
  case SUB()then
    SUB(df(x.e1,tv), df(x.e2,tv));
 // (x.e1*x.e2)' => x.e1'*x.e2+x.e1*x.e2'
  case MUL() then
    PLUS(MUL(df(x.e1,tv),x.e2),
         MUL(x.e1, df(x.e2,tv);));
  case DIV()then
    DIV(SUB(MUL(df(x.e1,tv),x.e2),
      MUL(x.e1,df(x.e2,tv))),
      MUL(x.e2,x.e2));
  case NEG() then NEG(df(x.e1,tv);
    // sin(x.e1)' => cos(x.e1) * x.e1'
  case CALL(IDENT("sin"),{_}) then
   MUL(CALL(IDENT("cos"),{x.e2[1]}),
   df(x.e2[1],tv)); //first elem from e2
   case AND() then
     AND(df(x.e1,tv), df(x.e2,tv));
   case OR() then
     OR(df(x.e1,tv), df(x.e2,tv));
   case LESS() then
     LESS(df(x.e1,tv), df(x.e2,tv));
   case GREATER() then
     GREATER(df(x.e1,tv), df(x.e2,tv));
    // etc...
  end match;
end df;
```

6 Exception Handling

The available MetaModelica exception handling construct has the following structure:

```
  try
    // Perform something which might fail
  else
    // Perform something different
  end try;
```

This is used extensively in many of the existing MetaModelica applications. There is also a function `fail()`, which can be called to create a failure that can be caught by the next level exception handler, typically after emitting an error message.

Julia has a very similar exception handling mechanism, with a `try – catch` statement and a `throw` call to create exceptions, but additionally has named exceptions and the `finally` clause.

7 Compiler Size and Performance

The OpenModelica compiler is a very large application implemented in MetaModelica 3.0. The sizes of the main parts are shown in Table 1. It is also bootstrapped, i.e., it compiles itself (Sjölund et al, 2014).

Moreover, the new OpenModelica compiler frontend, (Pop, et al, 2019) using the new facilities of MetaModelica 3.0, has a flattening speed of between one and two orders of magnitude faster than the previous compiler frontend.

Table 1. Sizes of OpenModelica compiler phases, lines of code, including several code generators.

Compiler Phase	*Lines*
BackEnd (from flat Modelica to sorted equation systems)	106299
FrontEnd (up to flat Modelica)	152059
Intermediate representation for code generation	17368
Code generators (generated code)	356889
Code generators (template source code)	8957
Code generators template language compiler & runtime	14586
OpenModelica scripting environment	35460
Utility modules	31050
Total size (excl. generated code)	*412869*

The compilation speed for two example models is indicated in Table 2.

Table 2. Compilation speed of the OpenModelica compiler implemented in MetaModelica 3.0 for some models, using a standard desktop computer.

Example model and size	*Compile time* (s)
Hummod, 29145 equations	239 s
Engine V6 (analytic), 9016 eqs	26 s

8 New Development Environment

As previously mentioned, the new integrated OMEdit-based development environment supports algorithmic code development in MetaModelica 3.0 or Modelica 3.4, or equation-based Modelica 3.4 model development. There are four simulation arrow buttons visible in Figure 3, from the left: standard simulation, simulation with the transformational debugger for equation models, simulation with the algorithmic code debugger, and simulation with 3D graphic animation.

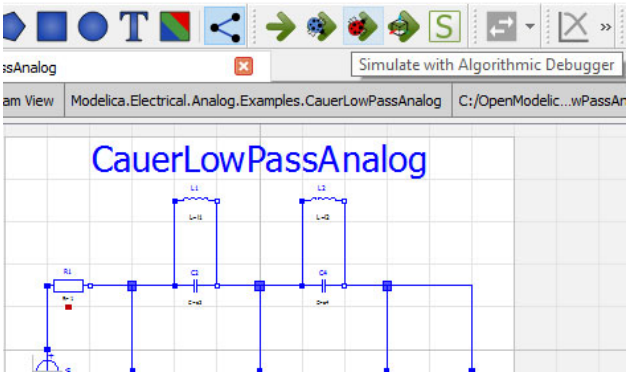

Figure 3. OMEdit with graphical view of an electrical model, as well as its four simulation+debug buttons.

8.1 Why Develop a New Environment

What was the motivation for developing a new version of the integrated environment since the Eclipse-based MDT-plugin was already available? There are basically two reasons:

- *Ease of use*. Users and developers asked for a more integrated tool, instead of needing to install the rather complex Eclipse tool for textual model debugging and MetaModelica development.
- *Performance*. Several developers were dissatisfied with the Eclipse plugin since they felt it was too slow, even though it provided useful functionality. By contrast, the new OMEdit environment is very fast, also for large applications.

8.2 Browsing and Debugging

The OMEdit-based development environment supports browsing and searching of MetaModelica packages just as the MDT Eclipse plugin. Debugging, including setting breakpoints, stepping, conditional breakpoints, attaching the debugger to an already running process, etc., is supported. A new feature is the ability to also show C function calls in the stack trace. See Figure 4, Figure 5, and Figure 6.

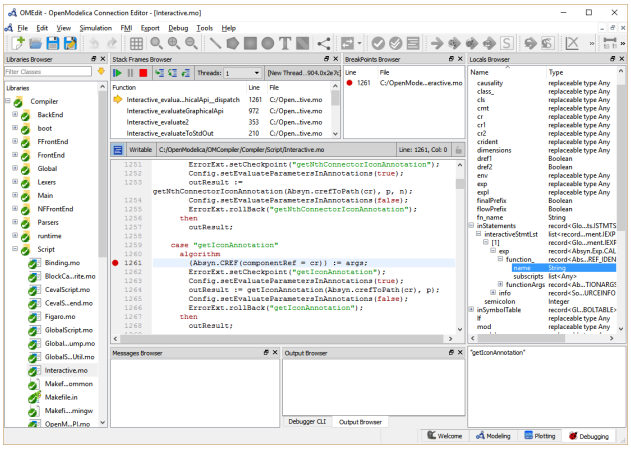

Figure 4. OMEdit during MetaModelica development. See also Figure 1 for more details.

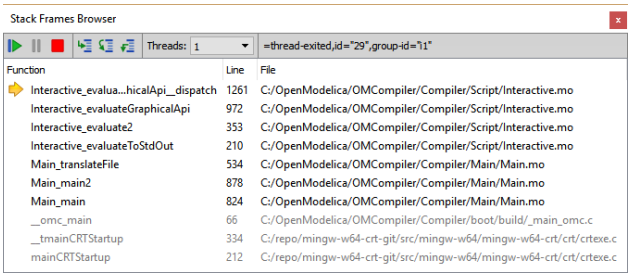

Figure 5. The function call stack trace browser showing Modelica and MetaModelica function calls at the top, C functions at the bottom.

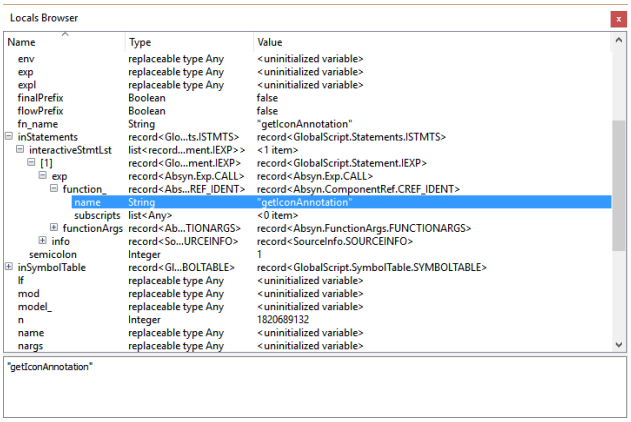

Figure 6. The local variables browser. Both standard Modelica data and MetaModelica data such as trees and lists are shown.

8.3 Separate Compilation

An efficient separate compilation mechanism for algorithmic MetaModelica or Modelica code is available, which is used routinely by the OpenModelica compiler developers to achieve rather fast turnaround time since more than two years. The compiler itself is a large application consisting of more than 250 packages, which is why separate compilation is quite important. Separate compilation of equation models is a separate topic not covered here, and is partly available using the FMI interface. The algorithmic code separate compilation mechanism works as follows:

A restriction has been introduced that all top-level packages are encapsulated and all dependencies of a module must be marked by an import statement. This improves performance in subsequent steps.

An additional useful restriction is that any public function, constant, or type may only refer to other public elements. By introducing this restriction, it is possible to create an interface file for each package that strips out protected elements and algorithm sections. Everything that remains in the resulting file is part of the interface, and loading each and every file (e.g., of the 250 OMC files) in the interface takes less than 0.6 seconds on a standard laptop.

For performance, a distinction between protected and public import elements in Modelica has been introduced. When calculating the list of interface files

DOI 10.3384/ecp19157289

that a package depends on, we must start with all the import elements in the package that is going to be compiled. For each of these packages, add the public imports to the list of packages that are going to be loaded (and do this recursively if any of those packages contain public imports). This is the list of all interfaces needed to calculate all types used in functions of the package that is going to be compiled. This set is substantially smaller than loading every single interface file.

Thus, to compile a package, the interface dependencies and the main file is loaded. For each function in the package, perform instantiation of that function and send it to the code generator. Compile each of those files with a C compiler and perform linking of the total application.

The total time including linking when updating a file without changing its public interface is 4.5 seconds for typical file (about 6000 lines of MetaModelica) or about 8 seconds for a large file (about 13000 lines). The tests were performed using an Intel Core i7 3820 @3.60GHz. There were about 250 packages in the OpenModelica compiler application used for the measurements.

Regarding Julia, the LLVM just-in-time compiler produces code directly into a binary image in memory. The disadvantage is that compilation is eventually repeated when code is loaded. However, there are some facilities for saving precompiled code to avoid recompilation.

8.4 Just-in-Time and Incremental Compilation

Julia uses LLVM for just-in-time compilation which combines performance with flexibility and interactivity. MetaModelica is currently compiled to C-code that is compiled either using the GNU C compiler or the LLVM Clang compiler. Instead, by directly generating LLVM compatible binary code it would be possible to get faster compilation and also to utilize the just-in-time capabilities of LLVM. Thus, recently we have made prototyping efforts for adapting the OpenModelica compiler backend intermediate representation (IR) to an LLVM compatible form (Andersson and Eriksson, 2018) and to interface it to the LLVM just-in-time compiler (Tinnerholm, 2019). In this way it is possible to obtain just-in-time capabilities with the associated flexibility without dependence on the Julia run-time. Additionally, there is an earlier prototype incremental compilation functionality in OpenModelica (Klinghed and Jansson, 2008).

9 Integration with Julia in Modelica Environments

As previously mentioned, Julia provides a powerful environment and a rapidly growing set of libraries for computational applications. Thus, some kind of integration with Julia seems relevant for many Modelica tools. We have identified a few levels of integration between a Modelica tool and Julia, from less to more integration:

- Level 1. Using Julia as a scripting language and making an API available for calling the Modelica tool from Julia. This have benefits of making Modelica model simulation and analysis available from Julia, e.g. for applications such as model-based control system design, e.g., (Thiele et al, 2019). Such an integration has recently been made available via the OMJulia subsystem (Lie et al, 2019) in OpenModelica
- Level 2a. Introducing an external function declaration facility for Julia functions. The benefits include making Julia functions available to Modelica modelers.
- Level 2b. Generating Julia code from the Modelica tool, i.e., adding another target language in addition to the typical C / C++. The benefits include Julia functions available to Modelica modelers as external functions and leverage some Julia run-time system functions for supporting the tool implementation.
- Level 3. Using Julia as the implementation language for the Modelica tool. This has the advantage of making the powerful Julia language and ecosystem available for tool implementation supporting both numeric and symbolic operations, and with rich libraries.

Regarding Level 3, two approaches are language embedding, i.e., embedding a Modelica-like language subset into the Julia language, or complete implementation from scratch in order to preserve all Modelica semantics.

Language embedding is a quick approach and has been chosen, e.g., by the Modia effort and is discussed in more detail in Section 10. As mentioned, an important disadvantage of such an approach is the loss of the static type checking and safe engineering practices which has been a strong guiding principle of Modelica language design. However, it might be possible to develop a static type system with static type checking for a subset of Julia. Most Julia code will not pass such a type checker, but for code that passes, this may solve the problem of safe engineering practice that is lacking for dynamic languages Some work in that direction is mentioned in (Chung et al, 2016) where a static type checker for a very small subset of Julia has been developed.

Regarding the other approach, implementation from scratch, a quicker approach is automatic translation/porting of code if the existing Modelica tool implementation language is close enough to Julia. Given the strong similarities between MetaModelica and Julia, it might be possible to auto-translate most of the OpenModelica compiler to Julia and thereby obtain a fully compliant Modelica compiler with static type checking implemented in Julia. As a first step, a

compatibility package for MetaModelica in Julia has been developed by us, including named pattern matching that was missing. One issue that was discovered is that recursive uniontypes that can be directly defined in MetaModelica are not possible in Julia. However, a solution was found by first declaring an abstract type of the uniontype which then can be referred to in the member structs. Another related issue is that Julia constants and types are declared in the order of the file, whereas in MetaModelica order does not matter. This either requires moving some MetaModelica code around if a very simple MetaModelica to Julia translator was implemented. Other than this, OpenModelica depends on a lot on external C code, which is expected to be the bulk of work to translate the entire compiler to Julia.

Performance is of great importance to OpenModelica but the MetaModelica compiler is primarily a high-level compiler and does not optimize many low-level operations due to maintenance issues of such code. Julia has a different approach where the language was designed to allow for high-performance code. An initial test showed that the Julia garbage collector is twice as fast as the Boehm garbage collector used in OpenModelica. And while the OpenModelica LLVM just-in-time compiler (Tinnerholm, 2019) is not feature-complete, it shows that LLVM just-in-time compilation such as Julia's could bring great performance benefits. If OpenModelica was ported to Julia and Modelica functions would be translated to the internal Julia AST, OpenModelica could gain performance by removing the interpreter and replacing it with running native code.

Further investigations of MetaModelica in Julia will follow, especially with regard to performance.

10 Related Work

OCaml (Minsky, et al, 2013) and Standard ML (Milner et al, 1997) are from 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. 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 (one documents what type one expects a variable should have) and negative (one ends up with a number of local variable declarations).

Another group of languages with similar constructs and pattern matching are the dynamically typed languages Lisp, Mathematica, Python, and Julia. Of these, only Julia currently seem to have good enough performance for efficient implementation of core compiler modules. Such dynamic typing is popular for prototyping but is negative from the correctness point of view since certain bugs may remain undetected for a long time and require exhaustive testing for detection. Some languages, like Lisp and Julia, provide meta-programming macros with Quote and Unquote constructs. This enables the use of concrete syntax fragments in meta-programming which may be slightly easier to use that the abstract syntax-oriented approach by the ML languages and MetaModelica, but on the other hand may be less efficiently compilable.

Several authors have used language embedding in a host language for implementing equation-based languages instead of designing a new language such as Modelica or MetaModelica. In this way the concrete and abstract syntax as well as parts of the implementation of the host language can be re-used. On the negative side, one is constrained by the host language regarding expressivity, semantics, and tool facilities (e.g. specific support for small-footprint embedded system code generation recently developed for OpenModelica). Giorgidze and Henrik Nilsson (2011) used this to embed an equation-based language in a functional language, and also used its JIT-compilation facilities for dynamically structure changing models. Erik Frisk (2017) used it for a simple diagnosis equation-based language embedded in Matlab and Python, using the available symbolic toolboxes. Hilding Elmqvist et al (2016) used language embedding of the Modia language prototype into Julia, using meta-programming macros, and also using its JIT-compilation for investigating structure changing models.

11 Conclusions

We have presented the MetaModelica 3.0 language for Modelica-style meta-programming together with its new OMEdit-based development environment. We have also done a short comparison to Julia and conclude that there are many similarities between MetaModelica and Julia. The current OpenModelica environment is the first Modelica environment that integrates meta-programming as well as graphical and textual modeling support and debugging in the same tool. The development environment provides efficient separate compilation with short turn-around time also for applications of several hundred thousand lines of code. Several facilities from the MDT Eclipse plug-in such as go to definition, type, and signature display, are planned to be made available in the new environment. A more efficient compiler frontend is almost completed, as well as a more powerful interface to the OpenModelica code generators. Moreover, further investigation of possible porting of MetaModelica to Julia is planned, which would make possible a Julia-based OpenModelica implementation.

DOI 10.3384/ecp19157289

Acknowledgements

This work has been supported by Vinnova in the ITEA OPENPROD, MODRIO, OPENCPS, and EMPHYSIS projects, and in the Vinnova RTISIM project. Support from the Swedish Government has been received from the ELLIIT project, as well as from the European Union in the H2020 INTO-CPS project. The OpenModelica development is supported by the Open Source Modelica Consortium.

References

Patrik Andersson and Simon Eriksson. *Efficient IR for the OpenModelica Compiler*. Maser Thesis report, Linköping University, 202018 | LIU-IDA/LITH-EX-A--2018/001—SE, October 2018.

Peter Aronsson, Peter Fritzson, Levon Saldamli, Peter Bunus and Kaj Nyström. Meta Programming and Function Overloading in OpenModelica. In *Proceedings of the 3rd International Modelica Conference*, Linköping, Sweden, Nov 2003.

Adeel Asghar, Sonia Tariq, Mohsen Torabzadeh-Tari, Peter Fritzson, Adrian Pop, Martin Sjölund, Parham Vasaiely, and Wladimir Schamai. An Open Source Modelica Graphic Editor Integrated with Electronic Notebooks and Interactive Simulation. In *Proc. of the 8th International Modelica Conference* 2011, pp. 739–747. Dresden, Germany, March.20-22, 2011.

Modelica Association. Modelica: A Unified Object-oriented Language for Physical Systems Modeling, Language Specification Version 3.4. May 2017. URL http://www.modelica.org/

Jeff Bezanson, Alan Edelman, Stefan Karpinski, Viral Shah. Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, Vol. 59, No. 1, pp. 65-98., 2017. http://julialang.org/publications/julia-fresh-approach-BEKS.pdf; see also: http://julialang.org/

Julialang. *Julia Language Documentation*, Release 1.02 Accessed November 14, 2018. www.julialang.org

David Broman and Jeremy Siek J. G. (2012): *Modelyze: a Gradually Typed Host Language for Embedding Equation-Based Modeling Languages*, University of California at Berkeley, No. UCB/EECS-2012-173, 2012. www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-173.html.

Benjamin Chung, Paley Li, and Jan Vitek. Static Typing Without Static Types – Typing Inheritance from the Bottom Up. In *Proc. of 1th Workshop on New Object-Oriented Languages (NOOL) 2016*, In conjunction with ACM SIGPLAN SPLASH Conference, Amsterdam, The Netherlands, October 31, 2017: http://www.it.uu.se/workshop/nool16/nool16-paper4.pdf

Hilding Elmqvist, Toivo Henningsson, and Martin Otter. Innovations for Future Modelica. In *Proc. of Modelica Conference 2017*, Prague, May 15-17, 2017.

Hilding Elmqvist, Toivo Henningsson, and Martin Otter. System Modeling and Programming in a Unified Environment based on Julia. In *Proc of ISoLA 2016*, (Eds) T. Margaria and B. Steffen, Part II, LNCS 9953, pp. 198-217, Oct. 10-14, 2016.

Erik Frisk, Mattias Krysander, and Daniel Jung. A Toolbox for Analysis and Design of Model Based Diagnosis Systems for Large Scale Models. *IFAC World Congress*. Toulouse, France, 2017. https://faultdiagnosistoolbox.github.io/ DOI: https://doi.org/10.1016/j.ifacol.2017.08.504

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

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. February 2011. URL http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-68361

Peter Fritzson. *Principles of Object Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. 1250 pages. ISBN 9781-118-859124, Wiley IEEE Press, 2014.

Peter Fritzson, Adrian Pop, Adeel Asghar, Bernhard Bachmann, Willi Braun, Robert Braun, Lena Buffoni, Francesco Casella, Rodrigo Castro, Alejandro Danós, Rüdiger Franke, Mahder Gebremedhin, Bernt Lie, Alachew Mengist, Kannan Moudgalya, Lennart Ochel, Arunkumar Palanisamy, Wladimir Schamai, Martin Sjölund, Bernhard Thiele, Volker Waurich, Per Östlund. The OpenModelica Integrated Modeling, Simulation, and Optimization Environment. In *Proceedings of the 1st American Modelica Conference*, Cambridge, MA, USA, October, 8-10, 2018. Published by LIU Electronic Press, www.ep.liu.se

George Giorgidze and Henrik Nilsson. Mixed-level Embedding and JIT Compilation for an Iteratively Staged DSL. In Julio Mariño, editor, *Proceedings of the 19th Workshop on Functional and (Constraint) Logic Programming (WFLP 2010)*, volume 6559 of Lecture Notes in Computer Science, pages 48-65, Springer-Verlag, 2011. http://www.cs.nott.ac.uk/~psznhn/Publications/wflp2010-lncs.pdf

Paul Hudak. *The Haskell School of Expression*. Cambridge University Press, 2000.

Kim Jansson and Joel Klinghed. *Incremental compilation and dynamic loading of functions in OpenModelica*. Master's thesis, Linköping University, IDA, June 2008. URN: urn:nbn:se:liu:diva-12329

Bernt Lie, Arunkumar Palanisamy, Alachew Mengist, Lena Buffoni, Martin Sjölund, Adeel Asghar, Adrian Pop, and Peter Fritzson. OMJulia: An OpenModelica API for Julia-Modelica Interaction. In Proc. of the *13th Int. Modelica Conference*, Regensburg, Germany, March 4-6, 2019.

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

Yaron Minsky, Anil Madhavepeddy, and Jason Hickey. *Real World OCaml*. O'Reilly, 2013.

Martin Otter and Hilding Elmqvist. Transformation of Differential Algebraic Array Equations to Index One Form. In *Proc. Modelica Conference*, Prague, May 15-17, 2017.

Mikael Pettersson, *Compiling Natural Semantics*. Lecture Notes in Computer Science (LNCS). Vol. 1549. 1999, Springer Verlag.

Adrian Pop, Martin Sjölund, Adeel Asghar, Peter Fritzson, Francesco Casella. Integrated Debugging of Modelica Models. *Modeling, Identication, and Control*, Vol 35, No 2, pp. 93-107, DOI: http://dx.doi.org/10.4173/mic.2014.2.3, ISSN 1890-1328, Aug 2014.

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

Adrian Pop and Peter Fritzson, MetaModelica: A Unified Equation-Based Semantical and Mathematical Modeling Language. In D. Lightfoot and C. Szyperski, editors, *Modular Programming Languages*, Vol. 4228 of Lecture Notes in Computer Science, pages 211/229. Springer Berlin / Heidelberg, 2006. DOI:10.1007/11860990_14.

Adrian Pop. *Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages*. Ph.D. Thesis. Linköping Studies in Science and Technology, Dissertation No. 1183, June 5, 2008.

Adrian Pop, Per Östlund, Francesco Casella, Martin Sjölund, Rüdiger Franke. A New OpenModelica Compiler High Performance Frontend. In Proc. of the *13th Int. Modelica Conference*, Regensburg, Germany, March 4-6, 2019.

RelationalAI. *Julia pattern matching Rematch.jl package*, 2018. https://github.com/RelationalAI-oss/Rematch.jl. Accessed Sept. 2018.

Peter van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.

Tim Sheard. *Accomplishments and Research Challenges in Meta-Programming*. Lecture Notes in Computer Science, 2196:2–.., 2001.

Tom Short. Sims - A Julia package for equation-based modeling and simulations. https://github.com/tshort/Sims.jl 2012.

Martin Sjölund, Peter Fritzson and Adrian Pop. Bootstrapping a Compiler for an Equation-Based Object-Oriented Language. DOI: 10.4173/mic.2014.1.1. *Modeling, Identification and Control*, Vol 35, No 1, pp 1-19, 2014.

Martin Sjölund. *Tools and Methods for Analysis, Debugging, and Performance Improvement of Equation-Based Models*. Ph.D. Thesis. Linköping Studies in Science and Technology, Dissertation No. 1664, June 1, 2015.

Kevin Squire. *Julia pattern matching Match.jl package*, 2013. https://github.com/kmsquire/Match.jl. Accessed Sept 2018.

Rickard Stallman, R. Pesch, S. Shebs, et al. Debugging with GDB. Free Software Foundation, 2014. URL http://www.gnu.org/software/gdb/documentation/.

Guy Steel and Rickard Gabriel. The Evolution of Lisp. In The second ACM SIGPLAN conference on History of programming languages, HOPLII. ACM, New York, NY, USA, pages 231{270, 1993. doi:10.1145/154766.155373

Bernt Lie, Arunkumar Palanisamy, Alachew Mengist, Lena Buffoni, Martin Sjölund, Adeel Asghar, Adrian Pop, and Peter Fritzson. OMJulia: An OpenModelica API for Julia-Modelica Interaction. In Proc. of the *13th Int. Modelica Conference*, Regensburg, Germany, March 4-6, 2019.

John Tinnerholm. *An LLVM backend for the OpenModelica Compiler*. Master Thesis LIU-IDA/LITH-EX-A--2019/001--SE, Dept. Computer and Information Science, Linköping University, January 2019.

Stephen Wolfram. *The Mathematica Book*, 5th Ed. Wolfram Media, Inc, 2003.

Dirk Zimmer. Equation-Based Modeling of Variable Structure Systems. PhD Dissertation, ETH Zürich. http://ecollection.library.ethz.ch/eserv/eth:1512/eth-1512-02.pdf