Compiling and Using Pattern Matching in Modelica

Kristian Stavåker, Adrian Pop, Peter Fritzson
PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, SE-581 83 Linköping, Sweden
{krsta, adrpo, petfr}@ida.liu.se

Abstract

Pattern matching is a well-known, powerful language feature found in functional programming languages. In this paper we present the implementation of pattern matching for Modelica. A pattern matching construct is useful for classification and decomposition of (possibly recursive) hierarchies of components such as the union type structures in the MetaModelica language extension. We argue that pattern matching not only is useful for language specification (as in the MetaModelica case) but also to write concise and neat functional-style programs. One useful application is in list processing (lists are currently missing from Modelica but are part of MetaModelica). Other possible applications are in the generation of models from other models, e.g. the generation of models with uncertainty equations or models with different parameters. Another application is the generation of documentation from models and checking of guidelines or certain properties of models.

Keywords: Pattern Matching, Modelica

1 Introduction

Pattern matching is a general operation that is used in many different application areas. Pattern matching is used to test whether constructs have a desired structure, to find relevant structure, to retrieve the aligning parts, and to substitute the matching part with something else.

In term pattern matching terms are matched against incomplete terms with variables and in, for instance, string pattern matching finite strings are matched against regular expressions (a typical application would be searching for substrings). Term pattern matching (which we will only consider henceforth) can be stated as: given a value *v* and patterns *p1*,…,*pN* is *v* an instance of any of the *p*'s? Language features for pattern matching (over terms) are available in all functional programming languages, for instance Haskell [3], OCaml [9], and Standard ML [10].

However, pattern matching is currently missing from Object-Oriented Equation-Based (EOO) Languages such as Modelica [2],[5], VHDL-AMS [7], and gPROMS [8]. Pattern matching features are also rare in imperative object-oriented languages even though some research has been carried out ([12],[13],[14] and [15]). In [15], for instance, the JMatch language which extends Java with pattern matching is described.

The language described in the paper [16] promotes the use of pattern matching constructs in object-oriented languages as a mean of exploring class hierarchies. One could for instance apply the visitor pattern to solve the same problem but as [17] notes this require a lot of code scaffolding and nested patterns are not supported.

The pattern matching construct for Modelica was presented in a paper on Modelica Metaprogramming extensions [4]. This paper discusses the implementation of pattern matching in the OpenModelica Compiler (OMC) [1]. We start by introducing the pattern matching construct in section 2 (syntax and semantics) followed by some programming examples in section 3. In section 4 we discuss type systems and problems with matching over class hierarchies. In section 5 we give an overview of implementation issues, followed by conclusions in section 6.

2 Pattern Matching

In this section we present the design of the pattern matching expression construct. Pattern matching expressions may occur where expressions can be used in Modelica code.

2.1 Syntax

We begin by giving the grammar rules.

```
match_keyword :
    match
  | matchcontinue

match_expression :
    match_keyword expression
    [ opt_string_comment ]
    local_element_list
    case_list case_else
    end match_keyword ";"
```

```
case_list :
    case_stmt case_list
  | case_stmt

equation_clause_case :
    equation equation_annotation_list
  | (* empty *)

case_stmt :
    case seq_pat
    [ opt_string_comment ]
    local_element_list
    equation_clause_case
    then expression ";"

case_else :
    else [ opt_string_comment ]
    local_element_list
    equation_clause_case
    then expression ";"
  | (* empty *)

local_element_list :
    local element_list
  | (* empty *)
```

The grammar rules that have been left out are rather self-describing (except the rule for patterns, `seq_pat`, which will not be covered here). An `equation_annotation_list`, for instance, is a list of equations. Only local, time-independent equations may occur inside a pattern matching expression and this must be checked by the semantic phase of the compiler. The difference between a pattern matching expression with the keyword `match` and a pattern matching expression with the keyword `matchcontinue` is in the fail semantics (see Section 2.3). The syntax can also be given (approximately) as follows.

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

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

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

2.2 Semantics

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

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

2.3 Pattern Matching Fail Semantics

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

3 Examples of Pattern Matching

As mentioned earlier a pattern matching construct is useful for language specification (meta-programming) but also as a tool to write functional-style programs. We start by giving an example of the latter usage.

```
function fac
  input Integer inExp;
  output Integer outExp;
algorithm
  outExp := matchcontinue (inExp)
    case (0) then 1;
    case (n) then if n>0 then n*fac(n-1)
             else fail();
  end matchcontinue;
end fac;
```

The above function will calculate the factorial value of an integer. If the number given as argument to the function is less than zero then the function will fail.

A fundamental data structure in MetaModelica is the union type which is a collection of records containing data, see example below.

```
uniontype UT
 record R1
   String s;
 end R1;

 record R2
   Real r;
 end R2;
end UT;
```

The pattern matching construct makes it possible to match on the different records. An example is given below.

```
function elabExp
  input Env.Env inEnv;
  input Absyn.Exp inExp;
  output Exp.Exp outExp;
  output Types.Properties outProperties;
algorithm
  (outExp,outProperties):=
  matchcontinue (inEnv,inExp)
    local
      ...
    case(_,Absyn.INTEGER(value=x))
      local Integer x;
      then (Exp.ICONST(x),Types.
             PROP(Types.T_INTEGER({})));
      ...
    case(env,Absyn.CREF(cRef = cr))
      equation
        (exp,prop) = elabCref(env,cr);
      then (exp,prop);
      ...
    case(env,Absyn.IFEXP(ifExp =
           e1,trueBranch=e2,eBranch=e3))
      local Exp.Exp e;
      equation
        (e1_1,prop1)=elabExp(env,e1);
        (e2_1,prop2)=elabExp(env,e2);
        (e3_1,prop3)=elabExp(env,e3);
        (e,prop)=elabIfexp(env,e1_1,prop1,
            e2_1,prop2,e3_1,prop3);
      then (e,prop);
  end matchcontinue;
end elabExp;
```

The function `elabExp` is used for elaborating expressions (type checking, constant evaluation, etc.). The union type `Absyn.Exp` contains a record representing an integer, a record representing a component reference, and so on. There is an environment union type, `Env.Env`, for component lookups.

Another situation where pattern matching is useful is in list processing. Again, lists are lacking from Modelica but are an important construct in MetaModelica (see the conclusions section for a small discussion). The following function selects an element that fulfills a certain condition from a list.

```
function listSelect
  input list<Type_a> inTypeALst;
  input Func_anyTypeToBool inFunc;
  output list<Type_a> outTypeALst;
  replaceable type Type_a subtypeof Any;
  partial function Func_anyTypeToBool
    input Type_a inTypeA;
    output Boolean outBoolean;
  end Func_anyTypeToBool;
algorithm
  outTypeALst:=
  matchcontinue
  (inTpeALst,inFunc)
  local
    list<Type_a> xs_1,xs; Type_a x;
    Func_anyTypeToBool cond;
  case ({},_) then {};
  case ((x :: xs),cond)
    equation
      true = cond(x);
      xs_1 = listSelect(xs, cond);
    then (x :: xs_1);
  case ((x :: xs),cond)
    equation
      false = cond(x);
      xs_1 = listSelect(xs, cond);
    then xs_1;
  end matchcontinue;
end listSelect;
```

The symbol :: is syntactic sugar for the cons operator. The function goes through the list one element at the time and if the condition is true the element is put on a new list and otherwise it is discarded. Another example of pattern matching with lists is given below. The function listThread takes two lists (of the same type) and interleaves them together.

```
function listThread
  input list<Type_a> inTypeALst1;
  input list<Type_a> inTypeALst2;
  output list<Type_a> outTypeALst;
  replaceable type Type_a subtypeof Any;
algorithm
  outTypeALst:=
  matchcontinue (inTypeALst1,inTypeALst2)
    local
      list<Type_a> r_1,c,d,ra,rb;
      Type_a fa,fb;
    case ({},{}) then {};
    case ((fa :: ra),(fb :: rb))
      equation
        r_1 = listThread(ra, rb);
        c = (fb :: r_1);
        d = (fa :: c);
      then d;
  end matchcontinue;
end listThread;
```

Yet another application for pattern matching might be to pattern match over class hierarchies. Modelica is an object-oriented language and as mentioned in the introduction and in [16] pattern matching is a powerful way

to explore a hierarchy of classes. Thus we would like to be able to write something like this:

```
record Expression
  ...
end Expression;

// Defining new expressions
record NUM
  extends Expression;
  Integer value;
end NUM;

record VAR
  extends Expression;
  Integer value;
end VAR;

record MUL
  extends Expression;
  Expression left;
  Expression right;
end MUL;

matchcontinue(inExp)
  case (NUM(x)) ...
  case (VAR(x)) ...
  case (MUL(x1,x2)) ...
end matchcontinue;
```

Here we could use the fact that `MUL` extends `Expression` when we do the pattern matching and in the static type checking. However, there are difficulties with this approach. Modelica features a structural type system thus type-equivalence is done over the structure of a record and not the name. In the implementation of pattern matching over union types, both in the implementation described in this paper (see Section 5.4) and in the RML-implementation of MetaModelica, we have augmented these type rules with features from nominal-based type systems where the name of a certain construct in important during type-checking.

Also, decomposition of records can be done via the dot-notation. No further implementation work has been done on pattern matching over record hierarchies.

3.1 Generating Uncertainty Equations

One application of pattern matching is the generation of models with uncertainty equations [18] from ordinary models. This has applications in sensitivity analysis. Basically pattern matching is employed to walk the meta-model of a specific model, and generate a new model in which variables have multiple values in a certain given interval.

4 Discussion on type systems

Modelica features a structural type system [5]. Another class of type systems is nominative type systems. In a structural type system two types are equal if they have the same structure and in a nominative type system this is determined by explicit declarations or the name of the types. Consider the following two records:

```
record REC1
  Integer int1, int2;
end REC1;

record REC2
  Integer int1, int2;
end REC2;
```

In a structural type system these two types would be considered equal since they have the same components. In a nominative type system, however, they would not be equal since they do not have the same names. Also in a nominal type system a type is a subtype of another type only if it is explicitly declared to be so (nominal subtyping). Consider the following three records.

```
record A
  Integer B, C;
end A;

record E1
  Integer B, C, D;
end E1;

record E2
  extends A;
  Integer D;
end E2;
```

In a structural type system record E1 would be a subtype of record A while in a nominative type system this would not be the case. Record E2, however, would be considered to be a subtype of record A in a nominative type system since an inheritance relation is explicitly declared. Java is an example of a language that uses nominative typing while C, C++ and C# use both nominative and structural (sub)-typing [19].

The typing rules in Modelica have to be augmented with nominal type system rules when typing pattern matching constructs. This is rather easy to enforce as we know that the records appearing in pattern matching should be part of a uniontype. When checking if a record is a subtype of another record and any of them appear in an uniontype then the subtyping rule will succeed only if they have the same name (they are equivalent) or if there is a explicit inheritance relation between them.

5 Pattern Matching Implementation

Since a pattern matching expression may contain complex nested patterns and partial overlaps between cases it should be compiled into a simpler, less complex form. Thus, a pattern matching expression is compiled into intermediate form (typically if-elseif-else nodes).

5.1 Overview

The pattern matching construct has been implemented in OMC using an algorithm described in [6]. Here a pattern is viewed as an alternation and repetition-free regular expression over atomic values, constructor names and wildcards. The algorithm first transforms a matchcontinue expression into a Deterministic Finite Automata (DFA) with subpatterns on the arcs. This DFA is then transformed into if-elseif-else nodes. The main goal of the algorithm is to unify overlapping patterns into common branches in the DFA in order to reduce code replication. This algorithm will also try to construct branches to already existing states in order to reduce code replication further. The end result is no nested patterns and no overlap between different if-cases.

The algorithm is composed of four steps: Preprocessing, Generating the DFA, Merging of equivalent states and Generating Intermediate Code. The preprocessing step takes all the match rules and produces a matrix of (preprocessed) patterns and a vector of final states (one for each row of patterns). In the next step the DFA is generated from the matrix and the vector of final states. In the following step equivalent states are merged and finally, in the last step, the intermediate code is generated.

We give a small example to illustrate the intuitive idea behind the algorithm (we use RML [6] style syntax).

```
case xs
   of C(1)   => A1
    | C(2)   => A2
    | C2()   => A3
```

The corresponding matrix and right-hand side vector:

```
| xs=C(ys=1) |    | A1 |
| xs=C(ys=2) |,   | A2 |
| xs=C2()    |    | A3 |
```

We select the first column (the only column). The constructor C matches the first two cases and the constructor C2 matches the last case. Since C2() does not contain any subpattern we are done on this "branch" and we reach the final state. We must continue to match on C's subpatterns, however, and we introduce a new variable ys. The variable ys is a *pattern-variable*, such a variable will be introduced for every sub-pattern.

```
case xs
  of C(ys)  => …
   | C2()   => A3
```

The rest of the matrix and vector:

```
| ys=1 |    | A1 |
| ys=2 |,   | A2 |
```

We match the rest of the matrix and vector and we get the result:

```
case xs
    of C(ys) =>
       ( case ys
            of 1 => A1
             | 2 => A2)
     | C2()      => A3
```

Note that in the real algorithm a DFA would first be created (with a state for each case and right-hand side and arcs for C, C2, '1' and '2'). This DFA would then be transformed into simple-cases.

5.2 OMC implementation

The specific OpenModelica translation path for Modelica code with matchcontinue constructs is presented in Figure 1. The matchcontinue expression has been added to the abstract syntax, `Absyn`. The pattern matching algorithm is invoked on the matchcontinue expression in the `Inst` module. The main function of the pattern matching algorithm is `PatternM.matchMain` which is given in a light version below.

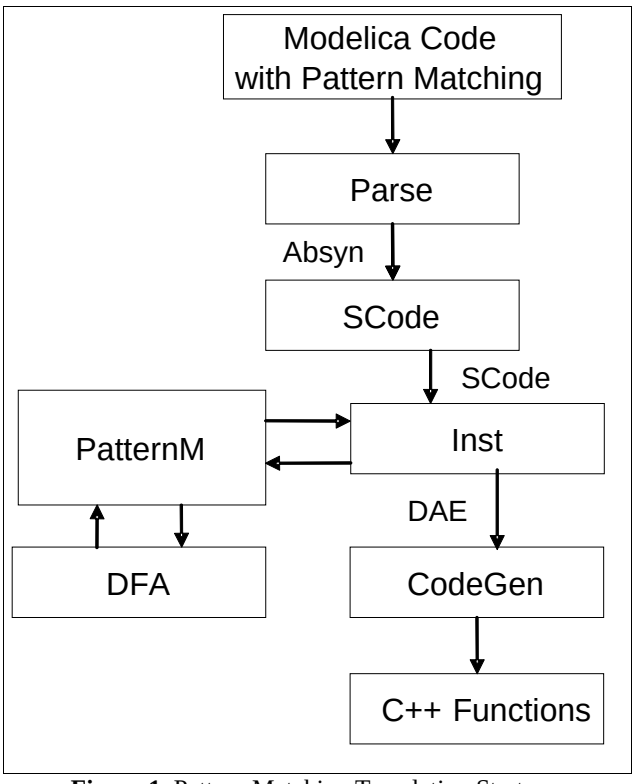

Figure 1. Pattern Matching Translation Strategy.

```
function matchMain
  input Absyn.Exp matchCont;
  input Env.Cache cache;
  input Env.Env env;
  output Env.Cache outCache;
  output Absyn.Exp outExpr;
```

```
algorithm
   (outCache,outExpr) :=
   matchcontinue (matchCont,cache,env)
   ...
case(localMatchCont,localCache,localEnv)
  local
  ...
  equation
    (localCache,...,rhList,patMat,...) =
      ASTtoMatrixForm(localMatchCont,
      localCache,localEnv);
    ...
       (startState,...)=matchFunc
      (patMat,rhList,STATE(...));
    ...
    dfaRec=DFA.DFArec(...,startState,
    ...);
    ...
    (localCache,expr) =
      DFA.fromDFAtoIfNodes(dfaRec,...,
      localCache,localEnv,...);
  then (localCache,expr);
end matchcontinue;
end matchMain;
```

The first function to be called in `matchMain is AST-ToMatrixForm` which creates a matrix out of the patterns as well as a list of right-hand sides (the code in a case clause except the actual pattern). This corresponds to step 1 of the above described algorithm. The list of right-hand side will actually only contain identifiers, and not all code in a right-hand side, so that the match algorithm won't have to pass along a lot of extra code. The code in the right-hand sides is saved in another list and is later added.

After this the function `matchFunc` is called with the matrix of patterns, the right-hand side list and a start state. This function will single out a column, create a branch and a new state for all matching patterns in the column and then call itself recursively on each new state and a modified version of the matrix, as described in [6]. The function (roughly) distinguishes between three cases:

- All of the patterns in the uppermost matrix row are wildcards.
- All of the patterns in the uppermost matrix row are wildcards or constants.
- At least one of the patterns in the uppermost matrix row is a constructor call.

However, due to the fail semantics of a matchcontinue expression we cannot simply discard all cases below a row with only wildcards as is done in [6]. This is due to the fact that a case-clause with only wildcards may fail and then an attempt to match the subsequent case-clause should be carried out.

Finally, the created DFA is transformed into if-else-elseif nodes (intermediate code) in the function `fromD-FAtoIfNodes`. This corresponds to step 3 and 4 of the algorithm described above. The pattern matching algorithm returns a value block expression containing the if-else-elseif nodes (see section 5.2). C++ code is then generated for the value block expression in the Codegen module.

5.3 Example

We first give an example of the compilation of a matchcontinue expression over simply types. In the next section we discuss the compilation of pattern matching over more complex types (union types, lists, etc.).

```
function func
  input Integer i1;
  input Integer i2;
  output Integer x1;
algorithm
  x1 :=
  matchcontinue (i1,i2)
    local
      Integer x;
    case (x as 1,2)
      local
      equation
        false = (x == 1);
      then 1;
    case (_,_) then 5;
  end matchcontinue;
 end func;
```

The code above is first compiled into intermediate form as seen in figure 1. The following C++-code is then generated from the intermediate code (note that the code is somewhat simplified):

```
{
 modelica_integer x;
 modelica_integer LASTRIGHTHANDSIDE__;
 integer_array BOOLVAR__; /* [2] */
 alloc_integer_array(&BOOLVAR__, 2, 1, 1);
  while (1) {
    try {
        state1:
        if ((i1 == 1) && (BOOLVAR__[1]
                || BOOLVAR__[2])) {
          state2:
          if ((i2 == 2) && BOOLVAR__[1]) {
            goto finalstate1;
          }
          else {
              state3:
              if (BOOLVAR__[2]) {
                goto finalstate2;
              }
          }
        }
        else {
            goto state3;
        }
        break;
        finalstate1:
        LASTRIGHTHANDSIDE__ = 1;
        x = i1;
        if (x == 1) {
          throw 1;
```

```
        }
        x1 = 1;
        break;
        finalstate2:
        LASTRIGHTHANDSIDE__ = 2;
        x1 = 5;
        break;
        }
        catch(int i) {
        BOOLVAR__[LASTRIGHTHANDSIDE__]=0;
        }
      }
  }
```

Each state label corresponds to a state in the DFA (which was the intermediate result of the pattern matching algorithm) and each if-case corresponds to a branch. See figure 2 for the generated DFA.

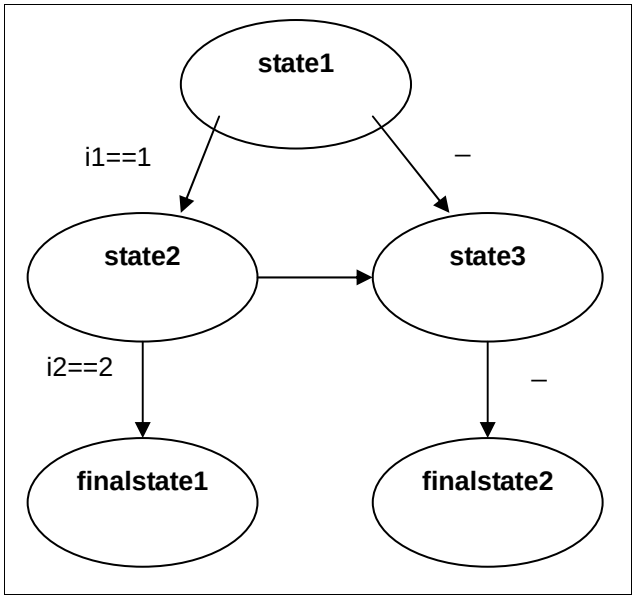

Figure 2. Code Example Generated DFA.

Note that if a case-clause fails then the next case-clause will be matched, since we have a matchcontinue expression. There is an array (BOOLVAR__) with an entry for each final state in the DFA. If a fail occurs an exception will be thrown and the catch-clause at the bottom will be executed. The catch-clause will set the array entry of the case-clause that failed to zero so that when the pattern matching algorithm restarts (notice the while(1) loop) this case-clause will not be entered again.

5.4 Pattern matching over union types, lists, tuples and option types

The remaining MetaModelica constructs (that are lacking from Modelica) are currently being added to OMC: lists, union types, option types and tuples. Compilation of MetaModelica code should only be performed if a special compile flag is set. Lists and union types were introduced in section 3. An option type can be seen as a union type with two records, SOME and NONE, where SOME may contain data and NONE is an empty record. A tuple type can be viewed as an unnamed record. See [4] for more details about MetaModelica.

We briefly discuss pattern matching over variables holding these types. Consider first an example with union types given below.

```
uniontype UT
record REC1
Integer field1;
Integer field2;
end REC1;

record REC2 ... end REC2;
end UT;

matchcontinue (x)
case (REC1(1,2)) ...
case (REC1(1,_)) ...
  ...
end matchcontinue;
```

The example above will result in the following intermediate code.

```
if (getHeaderNum(x) == 0) then
   Integer $x1 = getVal(x,1);
   Integer $x2 = getVal(x,2);
   if ($x1 == 1) then
      if ($x2 == 2) then
      ...
      elseif (true) then
      ...
      end if;
   end if;
elseif (...)
 ...
end if;
```

Note that static type checking is performed by the compiler to make sure that REC1 is a member of the type of variable x and that it contains two integer fields etc.. Union types are represented as boxed-values, with a header and subsequent fields, in C++. Each record in a union type is represented by a number (an enumeration). Since REC1 is the first record in the union type it is represented by 0. The function getHeaderNum is a builtin function that retrieves the header of variable x. The function getVal is also a builtin function that retrieves a data field (given by an offset) from the variable x.

Lists are compiled in a similar fashion.

```
matchcontinue (x)
case (1 :: var) ...
...
end matchcontinue;

=>

if (true) then
Integer $x1 = car(x,1);
```

```
list<Integer> $x2 = cdr(x);
      if ($x1 == 1) then
      ...
      elseif (...)
      ...
      end if;
end if;
```

The symbol :: is the cons constructor. The functions car and cdr are builtin functions for fetching the car and cdr parts of a list. Lists are also implemented as boxed values in the generated C++ code so this can be done in a straightforward way. An example of pattern matching over tuples is given below.

```
matchcontinue (x)
case ((5,false)) ...
case ((5,true)) ...
...
end matchcontinue;

=>

if (true) then
Integer $x1 = getVal(x,1);
Boolean $x2 = getVal(x,2);
      if ($x1 == 5) then
      ...
      elseif (...)
      ...
      end if;
end if;
```

Tuples are, just as union types and lists, implemented as boxed values in C++. The builtin function getVal takes an index and offsets into a boxed value in order to obtain the correct field.

Finally, option types are dealt with in a similar manner as union types.

Note that the reason why we need a run-time type check of union types is that a union type variable may hold any of several record types, which one can only be determined at run-time. When it comes to lists and tuples only one type can exist in a matchcontinue column, if this is violated it will be detected by the static type checker leading to a compile-time error.

5.5 Value block Expression

The value block expression allows equations and algorithm statements to be nested within another equation or algorithm statement. A value block expression contains a declaration part, a statements or equations part and a return expression. The return value of the value block is the value of the evaluated return expression. A value block has been added to OMC mainly because of its use as an intermediate data structure for the pattern matching expression.

6 Conclusions

We have presented the design and implementation of pattern matching for Modelica. We believe that adding this language feature to Modelica will result in a more powerful and complete language. Pattern matching is useful for traversing hierarchies of components, for writing functional-style programs, traversing lists, etc.. Pattern matching would be most useful if MetaModelica constructs such as lists and union type were merged with Modelica. If these constructs were to be included in Modelica, however, several type-related issues must be dealt with. Another possibility is to be able to pattern match over record-hierarchies (as shown in one of the examples). Even if decomposition of records can be done in a straightforward way through the dot-notation, checking the structure of a hierarchical record would imply a lot of if-statements which would be error-prone to be written.

7 Acknowledgements

This work has been supported by the Swedish Foundation for Strategic Research (SSF), in the RISE and VISIMOD projects, and by Vinnova in the Safe and Secure Modeling and Simulation project.

References

[1] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Software Development Environment. Simulation News Europe, 44/45, Dec 2005. http://www.ida.liu.se/projects/OpenModelica

[2] Peter Fritzson. Principles of Object-Oriented Modeling and Simulation with Modelica 2.1, 940 pp., Wiley-IEEE Press, 2004. See also: http://www.mathcore.com/drmodelica/

[3] Paul Hudak. The Haskell School of Expression. Cambridge University Press, 2000.

[4] Peter Fritzson, 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.

[5] The Modelica Association. The Modelica Language Specification Version 3.0, September 2007. http://www.modelica.org.

[6] Mikael Pettersson. Compiling Natural Semantics. PhD thesis, Linköping Studies in Science and Technology, 1995.

[7] Christen E. and K. Bakalar. VHDL-AMS-a hardware description language for analog and-mixed-signal applications, In 36th Design Automation Conference, June 1999

[8] Oh Min and C.C. Pantelides (1996) "A Modeling and Simulation Language for Combined Lumped and Distributed Parameter System." Computers & Chemical Engineering, vol 20: 6-7. pp. 611-633 1996.

[9] Xavier Leroy et al., The Objective Caml system. Documentation and user's manual, 2007, http://caml.inria.fr/pub/docs/manual-ocaml

[10] Robin Milner, Mads Tofte, Robert Harper and David MacQueen, The Definition of Standard ML, Revised Edition, MIT University Press, May 1997, ISBN: 0-262-63181-4

[11] Peter Fritzson. Modelica Meta-Programming and Symbolic Transformations, MetaModelica Programming guide, Version June 2007

[12] P.E. Moreau, C. Ringeissen, M. Vittek: A Pattern Matching Compiler for Multiple TargetLanguages. In: In Proc. of Compiler Construction (CC), volume 2622 of LNCS. (2003) 61–76

[13] M. Oderksy, P. Wadler: Pizza into Java: Translating theory into practice. In: Proc. of Principles of Programming Languages (POPL). (1997)

[14] M. Zenger, M. Odersky: Extensible Algebraic Datatypes with Defaults. In: Proc. of Int. Conference on Functional Programming (ICFP). (2001)

[15] J. Liu, A.C. Myers: JMatch: Iterable Abstract Pattern Matching for Java. In: Proc. of the 5th Int. Symposium on Practical Aspects of Declarative Languages (PADL). (2003) 110–127

[16] Burak Emir, Martin Odersky, and John Williams, Matching Objects With Patterns, LAMP-REPORT-2006-006,EPFL, Lausanne, Switzerland, Language Computer Corporation, Richardson Texas

[17] Martin Odersky, Raising Your Abstraction: In Defense of Pattern Matching, June 29, 2006. http://www.artima.com/weblogs/viewpost.jsp?thread=166742 [Last referenced on January 19, 2008]

[18] European standard ENV 13005.

[19] Benjamin C. Pierce, Types and Programming Languages. The MIT Press Massachusetts Institute of Technology Cambridge, Massachusetts 02142 http://mitpress.mit.edu ISBN 0-262-16209-1