Exception Handling for Modelica

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

Abstract

Any mature modeling and simulation language should provide support for error recovery. Errors might always appear in the runtime of such languages and the developer should be able to specify alternatives when failures happen. In this paper we present the design and implementation of exception handling in Modelica. To our knowledge this is the first approach of integrating equation-based object-oriented languages (EOO) with exception handling.

Keywords: Exception handling, Modelica.

1 Introduction

According to the terminology defined in IEEE Standard 100 [9], we define an *error* to be something that is made by humans. Caused by an error, a *fault* (also *bug* or *defect*) exists in an artifact, e.g. a model. If a fault is executed, this results in a *failure*, making it possible to detect that something has gone wrong.

Approaches to statically prevent and localize faults in equation-based object-oriented modeling languages are presented in [16] and [17]. However, in this paper we focus on language mechanisms for dynamically handling certain classes of faults and exceptional conditions within the application itself. This is known as exception handling. An exception is a condition that changes the normal flow of control in a program.

Language features for exception handling are available for most modern programming languages, e.g. object oriented languages such as Java [15], C++ [14], and functional languages such as Haskell [3], OCaml [12], and Standard ML [13].

However, exception handling is currently missing from Object-Oriented Equation-Based (EOO) Languages like Modelica [2][6], VHDL-AMS [10], gPROMS [11].

A short sketch of the syntax of exception handling for Modelica was presented in a paper on Modelica Metaprogramming extensions [5], but the design was incomplete, not implemented, and no further work was done at that time.

The design of exception handling capabilities in Modelica is currently work in progress. The following constructs are being proposed:

- A **`try...catch`** statement or expression.
- A **`throw`** `(...)` call for raising exceptions.

We have tried to keep the design of syntax and semantics of exception handling in Modelica as close as possible to existing language constructs from C++ and Java, while being consistent with Modelica syntax style.

2 Applications of Exceptions

In this section we provide examples of exception handling usefulness. There are three contexts in which exceptions can be thrown and caught: expression level, algorithm level and equation level.

```
import Modelica.Exceptions=Exn;

function log
 input Real x;
 output Real y;
algorithm
 y :=
 if x <= 0
 then
  throw (Exn.InvalidArgumentException(
         message="Logarithm is undefined
                  for ..."))
 else
   Modelica.Math.log(x);
end log;
```

Function `log` defined above will throw an exception if it is provided with an invalid argument. This is not only useful for mathematical functions, but also for functions (i.e. like the ones in `Modelica.Utilities` package) that deal with errors due to the operating system. A common for all tools standard hierarchy of exceptions could be defined in the Modelica Standard Library for all the exceptions categories needed. Depending on the simulation runtime implementation (i.e. language of choice) of the Modelica tool exceptions

could be translated from Modelica to the runtime and back.

A model that uses the try-catch construct in the expression and equation contexts is presented below:

```
model Test
 // try to read a value from file
 // and if it fails just give it
 // a default value.
 parameter Real p=
       try
        readRealParameter("file.txt","p")
       catch(Exn.IOException e)
        0
       end try;
 Real x;
 Real y;
equation
  try
    y = log(x);
  catch(Exn.InvalidArgumentException e)
    // terminate the simulation with
    // a message on what went wrong
    terminate(e.message);
  end try;
end Test;
```

In this model exception handling in expressions and equations are shown. In the case of exception handling in equations the example just terminates the simulation with an exception.

As one may have noticed the exceptions can be thrown during:

- Compilation time for expressions or functions that are evaluated at compile time
- Simulation time, due to exceptions raised into the solver, functions, expressions or equations.

All the exceptions raised during compile time are reported to the user. The exceptions which are caught are reported as warnings and the un-caught ones are reported as errors.

3 Exception Handling

In this section we present the design of the exception handling constructs. The grammar of the try-catch constructs is given below. The grammar follows the style from the Modelica Specification [6] and uses constructs defined there. Different try clauses for each of the expression, statements and equations contexts are defined.

```
exception_declaration:
  type_specifier IDENT
  ["(" exception_arguments ")"]

exception_arguments:
   expression
   [ "," exception_arguments ]
 | named_arguments

named_arguments:
  named_argument [ "," named_arguments ]

named_argument:
  IDENT "=" expression

name:
  IDENT [ "." name ]

throw_clause:
  throw ["(" name
  [ "(" exception_arguments ")"] ")" ]

try_clause_expression:
  try
    expression
  ( else_catch_clause_expression
    | catch_clause_expression
      { catch_clause_expresion }
      [ else_catch_clause_expression ] )
  end try

catch_clause_expression:
  catch "(" exception declaration ")"
    expression

else_catch_clause_expression:
  elsecatch
    expression

try_clause_algorithm:
  try
    { statement ";" }
    ( else_catch_clause_algorithm
      | catch_clause_algorithm
        { catch_clause_algorithm }
        [ else_catch_clause_algorithm ] )
  end try

catch_clause_algorithm:
  catch "(" exception declaration ")"
    { statement ";" }

else_catch_clause_algorithm
  elsecatch
    { statement ";" }

try_clause_equation
  try
    { equation ";" }
  ( else_catch_clause_equation
    | catch_clause_equation
      { catch_clause_equation }
      [ else_catch_clause_equation ] )
  end try

catch_clause_equation:
  catch "(" exception_declaration ")"
    { equation ";" }

else_catch_clause_expression:
  elsecatch
    { equation ";" }
```

Throwing via `throw;` without any parameter can only appear inside the catch clause and will throw the currently caught exception. This constraint is not specified

in the above grammar to keep it simple. Of course, it could be also checked by the semantics phase.

The try-catch clauses shown here are part of the various contexts rules in Modelica grammar: expressions, algorithm and equation.

3.1 Exception Handling for Statements

The statement variant has approximately the following syntax:

```
try
  <statements1>
catch(<exception_declaration>)
  <statements2>
end try;
```

The semantics of a try-catch for statements is as follows: An exception generated from a failure during the execution of `statements1` will lead to the execution of `statements2` if the exception matches the catch clause.

3.2 Exception Handling for Expressions

The syntax of the expression variant is as follows:

```
try
  <expression1>
catch(<exception_declaration>)
  <expression2>
end try;
```

The semantics of a try-catch for expressions is as follows: An exception generated from a failure while executing `expression1` will lead to the execution of `expression2` if the exception matches the catch clause.

3.3 Exception Handling for EOO

What does it mean to have exception handling for equation-based models? For example, if an uncaught exception, e.g. division by zero, occurs in any of the expressions or statements executed during the solution of the equation-system generated from the model, the catch could handle this, e.g. by simulating an alternative model (providing alternate equations), or stopping the simulation in a graceful way, e.g. by an error-message to the user. *The number of equations within the try construct must be the same as the number of equations in the catch part. This restriction is needed because models must be balanced. Of course, the restriction does not apply for the catch parts that only terminates the simulation and reports an error.*

The syntax of the equation variant is as follows:

```
try
  <equations1>
catch(<exception_declaration>)
  <equations2> | <terminate(...)>
end try;
```

The semantics of a try-catch for equations is as follows: If a failure generating an exception occurs during the solution of the equations in the set of equations denoted `equations1`, then if the catch matches the raised exception, then instead the `equations2` set is solved.

The source of the exception can be in the expressions and functions called in `equations1`, which are evaluated during the solving process. Certain exceptions might originate from the solver. In that case, a few selected solver exceptions need to be standardized and predefined.

The semantics of try-catch for equations is similar to the one for if-equations, with the difference that the event triggering the catch block is when an exception is thrown.

There could be several semantics for try-catch in equation section and they are discussed in Section 8.

3.4 Exception Handling and external functions

The compiler should be able to check the exceptions in order to:

- Report an error if the catch part tries to catch an exception that will never be thrown.
- Report exceptions that are not caught anywhere
- Generate efficient code for exceptions

The compiler can find automatically at compilation time what exceptions are thrown from models and functions defined in Modelica. However, the compiler must be provided with additional help when it comes to external functions. Therefore, when declaring external functions, the exceptions that might be thrown by them have to be declared too.

We could model this additional information in two ways: directly in the grammar or as annotations.

Directly in the grammar as part of the `element_list` (check the Modelica grammar for the element list specification) of the function or model:

```
throws_declaration:
   throws name { "," name } ";"
```

Is not really needed to specify in the grammar the possible exceptions to be thrown, we could use annotations instead:

```
annotation(throws={name1, name2, ... };
```

Names used above are constructed according to `name` grammar rule specified in the beginning of this section.

In the literature this feature of the compiler (or the language) is called *Checked exceptions* [18].

4 Transforming matchcontinue Fail Semantics

The current MetaModelica language extension has a simple fail semantics: fail exceptions can be thrown explicitly (via a `fail()` call) or implicitly (e.g., via a failure due to no patterns matching in a called function), and be caught/handled within the subsequent case(s) in the `matchcontinue` construct matching the same pattern.

The `matchcontinue` construct can be transformed into a match-expression that does not have the continue semantics after a failure, however requiring that the fail exception is caught in the same case branch.

Example:

```
matchcontinue x local ...
case Plus(a,b) equation   // raise
    ...generateFailureException...
case Plus(a,b) equation   // Catch
    handleFailure(a,b)
case _ handle_All_Inclusive_case();
end matchcontinue
```

can be transformed into the following:

```
match x local ...
case Plus(a,b) equation
  try
    ...generateFailureException...
  catch(Fail fail)
    handleFailure(a,b)
  end try;
case _ handle_All_Inclusive_case();
end match;
```

This transformation will be supported by a refactoring tool to transform existing code based on `matchcontinue` constructs into faster and clenrer code based on the `match` construct combined with exception handling. Such transformation will speed up the OpenModelica compiler, by removing many uses of matchcontinue with repeated matching due to overlapping patterns.

5 Exception Values

In this section we discuss different ways of representing exception values in Modelica. In general exceptions are values of a user defined type. Certain exceptions, such as `DivisionByZero` or `ArrayIndexOutOfBounds` are predefined. The user should be able to define exceptions hierarchically (i.e. packages of exceptions) and use inheritance to add extra information (components) to existing exceptions, thus creating specialized exceptions.

5.1 Exceptions as Types

We can model exceptions as a built-in Modelica type Exception. A pseudo-class declaration of such a type and its usage would look like:

```
type Exception
  // the value of the exception is
  // a string, accessed directly
  StringType 'value'
end Exception;

// Defining a new exception
type E1
  extends Exception;
end E1;

// Instantiate new exception
E1 e1 = "exception E1";
// Raise new exception
throw e1;

// Adding more information to an exception
type E2
  extends E1;
  parameter String moreInfo;
end E2;

// Instantiate the exception
E2 e2(moreInfo="E2 add") = "exception E2";

 // Throw exception
throw(e2);

try
  ...
catch(E2 e2)
  // here you can access the
  // e2 value directly
  // but you cannot access e2.moreInfo

catch(E1 e1)
  // here you can access the
  // value of e1 directly
end try;
```

Because we extend a basic type, it is possible to add more information to the exception, but this information cannot be accessed via dot notation.

5.2 Exceptions as Records

Another way to model exceptions is as Modelica records.

```
record Exception
  parameter String message;
end Exception;

// defining a new exception
record E1
  extends Exception(message="E1");
  parameter String moreInfo;
end E1;

// instantiate new exception
E1 e1(moreInfo="More Info");

// raise new exception
throw(e1);
```

```
// Try and catch
try
  ...
catch (E1 e1)
  // here you can access e.message
  // and e.moreInfo
catch (Exception e)
  // here you can access e.message
end try;
```

Modeling exceptions as records has many of the desired properties that a user might want. The problems we see here are that:

- Is not very intuitive to throw and catch arbitrary records.
- The hierarchical structure is partly lost during flattening, which means that for the records used in the throw/try-catch constructs this information should be preserved.
- The inheritance hierarchy is flattened for records and one would like to keep it intact to be able to catch exceptions starting from very specific (at the bottom of the inheritance hierarchy) to more general (at the top of the inheritance hierarchy)

We think that a better approach is with a new restricted Modelica class called `exception`.

5.3 New Restricted Class: exception

We believe that the best way to model exceptions in Modelica is by extending the language with a new restricted class called exception. Moreover, similar design choices have been made in Java or Standard ML, with their predefined exception types. In Java one can only throw objects of the `java.lang.Throwable` and its superclass `java.lang.Exception`. The C++ language allows throwing of values of any type. In Standard ML and OCaml exceptions values and their type need to be defined using a special syntax.

Exceptions can be represented in Modelica as a new restricted class in the following way:

```
exception E1
  parameter String message;
end E1;

E1 e1(message="More Info");
throw(e1); // raise new exception

// defining a new exception
exception E2
  extends E1(message="E2");
  parameter String moreInfo;
end E2;

// instantiate new exception
E2 e2(moreInfo="More Info");
throw(e2); // raise new exception

try
  ...
catch(E2 e2)
  // here you can access e.message
  // and e.moreInfo
catch(E1 e1)
  // here you can access e.message
end try;
```

Having a specific restricted class for exceptions would have the following advantages:

- Throwing and catching only values of restricted class exception is more intuitive than using records.
- Both the structural hierarchy and the inheritance hierarchy of the exceptions can be kept during flattening and translated to C++, Java, Standard ML or OCaml code more easily.
- The type checking of throw and try-catch constructs would be more specific and straightforward.

6 Typing Exceptions

Modelica features a structural type system, which means that two structures can be in the subtype relationship even if they have no explicit inheritance specified between them.

The type checking procedure for exceptions has to be different than for all the other constructs, namely:

- Only restricted classes of type exception can be thrown.
- When elaborating declarations of restricted class exception the subtype relationship applies only if there is *specific inheritance relation* between exceptions. This is needed because the exceptions have to be matched by name and have to be ordered so that the most specific case (supertype) is first and the least specific (subtype) is last in a catch clause.
- When translation declarations of restricted class exception there will be *no flattening of the inheritance hierarchy*.
- When elaborating catch clauses the compiler has to: i) *match the exception by name*, ii) *reorder the catch clauses in the inverse order of the inheritance relation* between exceptions or give an error if the less specific exceptions are matched before the more specific ones.
- The compiler has to check if an exception specified in the catch clause will actually be thrown from the try body or not. If such exception is not thrown the compiler can either discard the catch clause or issue a warning/error at that specific point.

With these new rules the typing of exception declarations, exception values and catch clauses can be achieved. After the translation, the runtime system and the language in which was implemented (C++, Java,

Standard ML) will provide the rest of the checking for exceptions.

7 Implementation

In this section we briefly present the OpenModelica implementation of exception handling. When referring to Exception Hierarchy we mean both the structural hierarchy and the inheritance hierarchy.

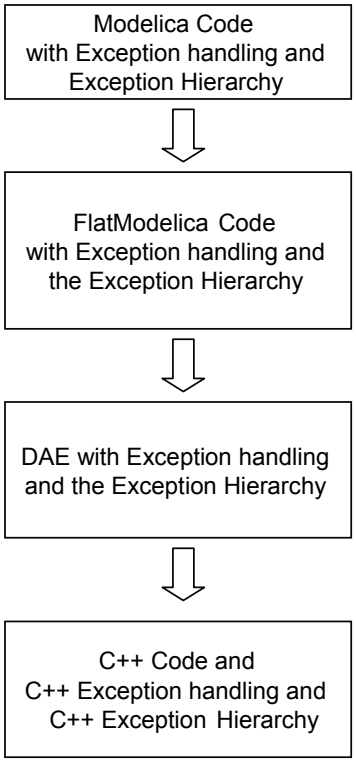

Figure 1. Exception handling translation strategy.

7.1 Overview

The general translation of Modelica with exception handling follows the path described in **Error! Reference source not found.**. The exception handler and the exception hierarchy are passed through the compiler via the intermediate representations of each phase until the C++ code is generated (or any other language code used in the backends of different Modelica compilers).

The specific OpenModelica translation path for Modelica code with exception handling is presented in Figure 2.

Exception handling in OpenModelica required the following extensions:

- The parser was extended with the proposed exception handling grammar.
- Each intermediate representation of the OpenModelica compiler was augmented with support for exceptions.

Both the structural and the inheritance hierarchy of the exceptions are passed through the OpenModelica compiler until C++ code is generated.

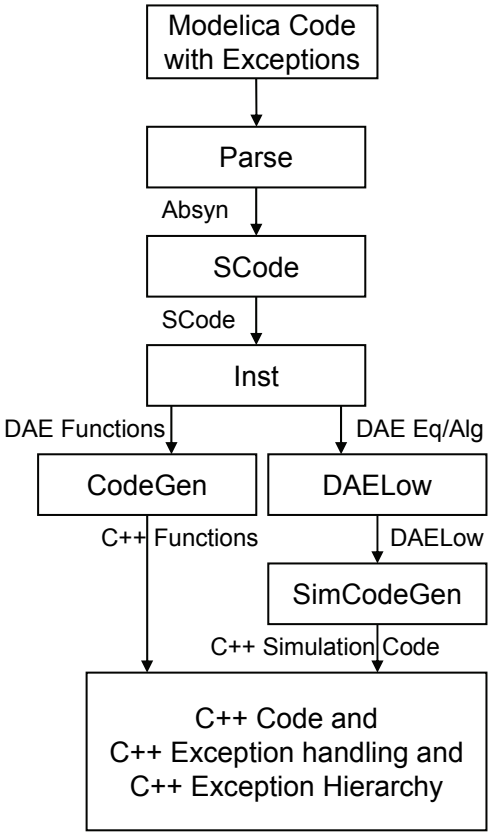

Figure 2. OpenModelica implementation.

7.2 Translation of Exception values

The translation from the internal representation to C++ code is straightforward: a Modelica exception maps to a C++ class. For example, the following Modelica code with exceptions:

```
exception E
  parameter String message;
end E;

exception E1
  extends E(message="E1");
  parameter Integer id = 1;
end E1;
```

is translated into the following C++ code:

```
class E
{
  public:
  modelica_string message;
  E(modelica_string message_modification)
  {
    message = message_modification;
  }
  E()
  { message = ""; }
}
```

```
class E1 : public E
{
  public:
  modelica_integer id;
  E1(modelica_string message_modification,
     modelica_integer id_modification)
  {
    message = message_modification;
    id = id_modification;
  }
  E1()
  {
    message = "E1";
    id = 1;
  }
}
```

The following Modelica code for exception instantiation and exception throwing:

```
E  e;  throw(e);
E1 e1; throw(e1);

E1 e2(message="E2", id=2);
throw(e2);

E1 e3(message="E3");
throw(e3);
```

is translated to the following C++ code:

```
E  *e  = new E();  throw e;
E1 *e1 = new E1(); throw e1;

E1 *e2 = new E1("E2", 2);
throw e2;

E1 *e3 = new E1();
e3->message = "E3";
throw e3;
```

Is also possible to represent exception values in C++ as objects allocated on the stack, i.e.: `E1 e2("E2", 2);`.

7.3 Translation of Exception handling

The C++ exception handling code follows the Modelica code. The table below defines the translation procedure for Modelica including the MetaModelica extensions.

Modelica Expressions	**C++**

`x :=`
`try`
`  exp1`
`catch(E e)`
`  exp2`
`end try;`
	`modelica_type temp;`
	
	`try`
	`{`
	`  temp = exp1;`
	`}`
	`catch(E *e)`
	`{`
	`  temp = exp2;`
	`}`
	`x = temp;`

Modelica Statements	**C++**

`try`
`  <statements>`
`catch(E e)`
`  <statements>`
`end try;`
	`try`
	`{`
	`  // Modelica`
	`  // corresponding`
	`  // C++ statements`
	`}`
	`catch(E *e)`
	`{`
	`  // Modelica`
	`  // corresponding`
	`  // C++ statements`
	`}`

Modelica Equations	**C++**

`try`
`  <eqnsA>`
`catch (Ex1 e1)`
`  <eqnsB>`
`end try;`
 	
`try`
`  <eqnsC>`
`catch (Ex2 e2)`
`  <eqnsD>`
`end try;`
	`event1=false;`
	`event2=false;`
	
	`while time < stopTime`
	`{`
	`  try{`
	`  call SOLVER for problem:`
	`  if event1`
	`  then`
	`  eqnsB;`
	`  else`
	`  eqnsA`
	`  end if;`
	
	`  if event2`
	`  eqnsD;`
	`  else`
	`  eqnsC;`
	`  end if;`
	`  }`
	`  catch(Ex1 *e1)`
	`  {`
	`  discard posible`
	`  calculated current`
	`  step values;`
	`  reinit the solver`
	`  with previous step`
	`  values;`
	`  event1 = true;`
	`  }`
	`  catch(Ex2 *e2)`
	`  {`
	`  discard posible`
	`  calculated current`
	`  step values;`
	`  reinit the solver`
	`  with previous step`
	`  values;`
	`  event2 = true;`
	`  }`
	`}`

8 Further Discussion

During the design and implementation of exception handling we have encountered various issues which we will present in this section. The exception handling in

expressions and algorithm sections are straightforward. However when extending exception handling for equation sections there are several questions which influence the design choices that come to mind:

Questions: *Is the exception handling necessary for equation sections? If yes, what are the semantics that would bring the most usefulness to the language?*

Answers: We believe that exception handling is necessary in the equation sections at least to give more useful errors to the user (i.e. with `terminate(message)` in the catch clause) or to provide an alternative for gracefully continuing the simulation. Right now in Modelica there is no way to tell where a simulation failed. There are assert statements that provide some kind of lower level checking but they do not function very well in the context of external functions. As example where alternative equations for simulation might be needed we can think of the same system in different level of detail. Where the detailed system can fail due to complexity and numerical problem the simulation can be continued with the less complex system.

Semantics of try-catch in equation sections

Several semantics can be employed to deal with try-catch clauses in equation sections:

1. Terminate the simulation with a message (as we show in application section)
2. Continue the simulation with the alternative equations from the catch clause activated and the ones from the try-body disabled. When the exception occurs the calculated values in that solver step are discarded and the solver is called again with previous values and the alternative from the catch clause.
3. Signal the user that an exception occurred and restart the simulation from the beginning with the catch-clause equations activated.
4. When an exception occurs, discard the values calculated in the current step and activate the alternative equations from catch-clause. However, at the next step try again the equations from the try-body. This will make the catch-clause equation active only for the steps where an error might occur.

We think that the most useful design for exception handling in equation section is the one that has both features 1 and 2 active.

9 Conclusions

We have presented the design and the implementation of exception handling for Modelica. We strongly believe in the need for a well designed exception handling in Modelica. By adding exception handling constructs to the language we get a more complete language and provide the developer with means to better control exceptions. There are several issues that have to be considered when designing and implementing these constructs which we have discussed in this paper.

10 Acknowledgements

This work has been supported by Swedish Foundation for Strategic Research (SSF), in the RISE and VISIMOD projects, 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://ww.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] Kenneth C. Louden: Programming Languages, Principles and Practice. 2:nd edition, Thomson Brooks/Cole, 2003, (ISBN 0-534-95341-7)

[5] 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.

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

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

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

[9] IEEE Standards Information Network. IEEE 100 The Authoritative Dictionary of IEEE Standards Terms. IEEE Press, New York, USA, 2000.

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

[11] 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.

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

[13] 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

[14] Bjarne Stroustrup: The C++ Programming Language (Special Edition). Addison Wesley. Reading Mass. USA. 2000. ISBN 0-201-70073-5. 1029 pages. Hardcover.

[15] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java™ Language Specification, Third Edition, ISBN-13: 978-0321246783, Prentice Hall, June 2005.

[16] Peter Bunus. Debugging Techniques for Equation-Based Languages. Ph.D. Thesis No. 873, Linköping University, 2004

[17] David Broman. Safety, Security, and Semantic Aspects of Equation-Based Object-Oriented Languages and Environments. Licentiate Thesis, Thesis No. 1337, Linköping University, December, 2007.

[18] Exception Handling: http://en.wikipedia.org/wiki/Exception_handling