

Linköping Studies in Science and Technology

Dissertation No. 1183

Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages

by

Adrian Pop



Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköping 2008

Integrated Model-driven Development Environments for Equation-based Object-oriented Languages

by

Adrian Pop

June 2008

ISBN 978-91-7393-895-2

Thesis No. 1183

ISSN 0345-7524

ABSTRACT

Integrated development environments are essential for efficient realization of complex industrial products, typically consisting of both software and hardware components. Powerful equation-based object-oriented (EOO) languages such as Modelica are successfully used for modeling and virtual prototyping increasingly complex physical systems and components, whereas software modeling approaches like UML, especially in the form of domain specific language subsets, are increasingly used for software systems modeling.

A research hypothesis investigated to some extent in this thesis is if EOO languages can be successfully generalized also to support software modeling, thus addressing whole product modeling, and if integrated environments for such a generalized EOO language tool support can be created and effectively used on real-sized applications.

However, creating advanced development environments is still a resource-consuming error-prone process that is largely manual. One rather successful approach is to have a general framework kernel, and use meta-modeling and meta-programming techniques to provide tool support for specific languages. Thus, the main goal of this research is the development of a meta-modeling approach and its associated meta-programming methods for the synthesis of model-driven product development environments that includes support for modeling and simulation. Such environments include components like model editors, compilers, debuggers and simulators. This thesis presents several contributions towards this vision in the context of EOO languages, primarily the Modelica language.

Existing state-of-the art tools supporting EOO languages typically do not satisfy all user requirements with regards to analysis, management, querying, transformation, and configuration of models. Moreover, tools such as model-compilers tend to become large and monolithic. If instead it would be possible to model desired tool extensions with meta-modeling and meta-programming, within the application models themselves, the kernel tool could be made smaller, and better extensibility, modularity and flexibility could be achieved.

We argue that such user requirements could be satisfied if the equation-based object-oriented languages are extended with meta-modeling and meta-programming. This thesis presents a new language that unifies EOO languages with term pattern matching and transformation typically found in functional and logic programming languages. The development, implementation, and performance of the unified language are also presented.

The increased ease of use, the high abstraction, and the expressivity of the unified language are very attractive properties. However, these properties come with the drawback that programming and modeling errors are often hard to find. To overcome these issues, several methods and integrated frameworks for run-time debugging of the unified language have been designed, analyzed, implemented, and evaluated on non-trivial industrial applications.

To fully support development using the unified language, an integrated model-driven development environment based on the Eclipse platform is proposed, designed, implemented, and used extensively. The development environment integrates advanced textual modeling, code browsing, debugging, etc. Graphical modeling is also supported by the development environment

based on a proposed ModelicaML Modelica/UML/SysML profile. Finally, serialization, composition, and transformation operations on models are investigated.

This work has been supported by the National Computer Science Graduate School (CUGS), the ProViking Graduate School, the Swedish Foundation for Strategic Research (SSF) financed research on Integrational Software Engineering (RISE), VISIMOD and Engineering and Computational Design (SECD) projects; the Vinnova financed Semantic Web for Products (SWEBPROD) and Safe and Secure Modeling and Simulation projects. Also, we acknowledge the cooperation with Reasoning on the Web with Rules and Semantics (REWERSE) "Network of Excellence" (NoE) funded by the EU Commission and Switzerland within the "6th Framework Programme" (FP6), Information Society Technologies (IST). We also acknowledge support from the Swedish Science Council (VR) in the project on High-Level Debugging of Equation-Based System Modeling & Simulation Languages and from MathCore Engineering AB.

Acknowledgements

*A thesis cannot be finished; it has to be abandoned.
Finally, the deadline for this thesis has come!*

I would like to thank the following people and organizations (in no particular order or classification) which were important to me:

- Supervisors (Peter Fritzson, Uwe Abmann).
- Opponent (Hans Vangheluwe) and Committee (Görel Hedin, Petter Krus, Tommi Karhela)
- PELAB (Bodil Mattsson Kihlström, Kristian Sandahl, Christoph Kessler, Mariam Kamkar, Mikhail Chalabine, Olof Johansson, David Broman, Kristian Stavåker, Håkan Lundval, Andreas Borg, Emma Larsdotter Nilsson, Mattias Eriksson, Levon Saldalmli, Kaj Nyström, Martin Fransson, Anders Sandholm, Andrzej Bednarski, John Wilander, Jon Edvardsson, Jesper Andersson, Mikael Pettersson, etc.).
- Master thesis students: Simon Björklén, Emil Carlsson, Dynamic Loading Team (Kim Jansson, Joel Klinghed), Refactoring Team (Kristoffer Norling, Mikael Blom), MDT Team (Elmir Jagudin, Andreas Remar, David Akhvlediani, Vasile Băluță), OMNotebook Team (Ingemar Axelsson, Anders Fernström, Henrik Eriksson, Henrik Magnusson).
- MathCore (Katalin Bunuș, Peter Aronsson, Lucian Popescu, Daniel Hedberg, Björn Zachrisson, Vadim Engelson, Jan Brugård, etc.).
- IDA (Lillemor Wallgren, Inger Emanuelsson, Petru Eleș, Gunilla Mellheden, Britt-Inger Karlsson, Inger Norén, etc.), DIG (Lisbeth Linge, Tommy Olsson, Henrik Edlund, Andreas Lange), TUS, ESLAB.
- Family (Flore, ȚiȚi, Paul & Liana, Teodor & Letiția, ...) and Friends (Peter & Katalin Bunuș, Sorin Manolache, Călin Curescu & Njideka Andreea Udechukwu, Traian & Ruxandra Pop, Alexandru Andrei & Diana Szentiványi, Claudiu & Aurelia Duma, Ioan & Simona Chisalița, Șerban Stelian, Cristian & Cristina Tomoiagă, Adrian & Simona Ponoran, Dicu Ștefan, Ioan & Adela Pleșa, Andreea & Sorin Marian, Horia Bochiș, Ilie Savga, and many more).
- Thesis reviewers (Peter Fritzson, Hans Vangheluwe, Görel Hedin, Petter Krus, Tommi Karhela, Jörn Guy Süß, Kristian Stavåker, Paul Pop)
- All others that I might forgot to mention.

Adrian Pop

Linköping, June 5, 2008

Table of Contents

Part I Motivation, Introduction, Background and Related Work	1
Chapter 1 Introduction	3
1.1 Research Objective (Motivation).....	4
1.2 Contributions	5
1.3 Thesis Structure	6
1.4 Publications	8
Chapter 2 Background and Related Work.....	11
2.1 Introduction	11
2.1.1 Systems, Models, Meta-Models, and Meta-Programs	11
2.1.2 Meta-Modeling and Meta-Programming Approaches	12
2.2 The Modelica Language	14
2.2.1 An Example Modelica Model.....	17
2.2.2 Modelica as a Component Language.....	18
2.3 Modelica Environments.....	19
2.3.1 OpenModelica.....	19
2.3.2 MathModelica, Dymola, SimulationX.....	20
2.4 Related Equation-based languages: gProms, VHDL-AMS and the χ language	23
2.5 Natural Semantics and the Relational Meta-Language (RML)	24
2.5.1 An Example of Natural Semantics and RML	25
2.5.2 Specification of Syntax	27
2.5.3 Integrated Environment for RML	27
2.6 The eXtensible Markup Language (XML)	28
2.7 System Modeling Language (SysML).....	30
2.7.1 SysML Block Definitions	32
2.8 Component Models for Invasive Software Composition.....	32
2.9 Integrated Product Design and Development	35
Part II Extending EOO Languages for Safe Symbolic Processing.....	37
Chapter 3 Extending Equation-Based Object-Oriented Languages.....	39
3.1 Introduction	39
3.1.1 Evaluator for the Exp1 Language in the Unified Language....	40
3.1.2 Examples of Pattern Matching.....	42
3.1.3 Language Design	45
3.2 Equations	45
3.2.1 Mathematical Equations.....	46
3.2.2 Conditional Equations and Events	46
3.2.3 Single-Assignment Equations	47
3.2.4 Pattern Equations in Match Expressions.....	47
3.3 High-level Data Structures	49
3.3.1 Union-types.....	49

3.3.2	Lists, Tuples and Option Types.....	50
3.4	Solution of Equations.....	51
3.5	Pattern Matching.....	52
3.5.1	Syntax.....	53
3.5.2	Semantics	54
3.5.3	Discussion on type systems.....	55
3.6	Exception Handling	56
3.6.1	Applications of Exceptions	56
3.6.2	Exception Handling Syntax and Semantics.....	58
3.6.3	Exception Values.....	62
3.6.4	Typing Exceptions.....	64
3.6.5	Further Discussion.....	65
3.7	Related Work	66
3.8	Conclusions and Future Work.....	67
Chapter 4 Efficient Implementation of Meta-Programming EOO Languages		69
4.1	Introduction.....	69
4.2	MetaModelica Compiler Prototype.....	69
4.2.1	Performance Evaluation of the MetaModelica Compiler Prototype	70
4.3	OpenModelica Bootstrapping	72
4.3.1	OpenModelica Compiler Overview	72
4.4	High-level Data Structures Implementation.....	75
4.5	Pattern Matching Implementation.....	77
4.5.1	Implementation Details	78
4.6	Exception Handling Implementation	84
4.6.1	Translation of Exception Values	86
4.6.2	Translation of Exception Handling	88
4.7	Garbage Collection	89
4.7.1	Layout of Data in Memory	90
4.7.2	Performance Measurements	91
4.8	Conclusions.....	93
Part III Debugging of Equation-based Object Oriented Languages.....		95
Chapter 5 Portable Debugging of EOO Meta-Programs		97
5.1	Introduction.....	97
5.2	Debugging Method – Code Instrumentation.....	97
5.2.1	Early Instrumentation.....	98
5.2.2	Late Instrumentation	99
5.3	Type Reconstruction	99
5.4	Performance Evaluation.....	100
5.4.1	The Test Machine.....	100
5.4.2	The Test Files.....	100
5.4.3	Compilation Performance.....	102
5.4.4	Run-time Performance	102

5.5	Tracing and Profiling	103
5.5.1	Tracing	103
5.5.2	Profiling	104
5.6	The Eclipse-based Debugging Environment	104
5.6.1	Starting the Modelica Debugging Perspective	105
5.6.2	Setting the Debug Configuration	106
5.6.3	Setting/Deleting Breakpoints	107
5.6.4	The Debugging Session and the Debug Perspective	108
5.7	Conclusions	110
Chapter 6	Run-time Debugging of EOO Languages	111
6.1	Introduction	111
6.2	Debugging Techniques for EOO Languages	111
6.3	Proposed Debugging Method	112
6.3.1	Run-time Debugging Method	113
6.4	The Run-time Debugging Framework	115
6.4.1	Translation in the Debugging Framework	115
6.4.2	Debugging Framework Overview	118
6.4.3	Debugging Framework Components	118
6.4.4	Implementation Status	119
6.5	Conclusions and Future Work	120
Chapter 7	Debugging Natural Semantics Specifications	121
7.1	Introduction	121
7.2	Related Work	122
7.3	The rml2c Compiler and the Runtime System	122
7.4	Debugger Design and Implementation	124
7.5	Overview of the RML Integrated Environment	125
7.6	Design Decisions	126
7.6.1	Debugging Instrumentation	126
7.6.2	External Program Database	126
7.6.3	External Data Value Browser	126
7.6.4	Why not an Interpreter?	127
7.7	Instrumentation Function	127
7.8	Type Reconstruction in the Runtime System	128
7.9	Debugger Implementation	129
7.9.1	The rml2c Compiler Addition	129
7.9.2	The Debugging Runtime System	129
7.9.3	The Data Value Browser	130
7.9.4	The Post-Mortem Analysis Tool	130
7.10	Debugger Functionality	130
7.10.1	Starting the RML Debugging Subprocess	131
7.10.2	Setting/Deleting Breakpoints	132
7.10.3	Stepping and Running	133
7.10.4	Examining Data	133
7.10.5	Additional Commands	136
7.11	The Data Value Browser	136

7.12	The Post-Mortem Analysis Tool.....	138
7.13	Performance Evaluation.....	139
7.13.1	Code Growth	139
7.13.2	The Execution Time	140
7.13.3	Stack Consumption	140
7.13.4	Number of Relation Calls.....	141
7.14	Conclusions and Future Work.....	141
Part IV Advanced Integrated Environments		143
Chapter 8 Modelica Development Tooling (MDT)		145
8.1	Introduction.....	145
8.1.1	Integrated Interactive Programming Environments	145
8.1.2	The Eclipse Framework.....	147
8.1.3	Eclipse Platform Architecture	147
8.1.4	OpenModelica MDT Eclipse Plugin	148
8.2	OpenModelica Environment Architecture	149
8.3	Modelica Development Tooling (MDT) Eclipse Plugin.....	150
8.3.1	Using the Modelica Perspective	151
8.3.2	Creating a Project.....	151
8.3.3	Creating a Package	151
8.3.4	Creating a Class.....	151
8.3.5	Syntax Checking	153
8.3.6	Code Completion.....	153
8.3.7	Automatic Indentation.....	154
8.4	The OpenModelica Debugger Integrated in Eclipse	156
8.5	Simulation and Plotting from MDT	156
8.6	Conclusions.....	157
Chapter 9 Parsing-Unparsing and Refactoring.....		159
9.1	Introduction.....	159
9.2	Comments and Indentation	160
9.3	Refactorings	160
9.3.1	The Principle of Minimal Replacement	160
9.3.2	Some Examples of Refactorings	161
9.3.3	Representing Comments and User-Defined Indentation	161
9.4	Implementation	162
9.4.1	Base Program representation.....	163
9.4.2	The Parser.....	163
9.4.3	The Scanner.....	163
9.4.4	The New Unparser	163
9.5	Refactoring Process	164
9.5.1	Example of Function Name Refactoring.....	164
9.5.2	Calculation of the Additional Overhead.....	167
9.5.3	Unparsers/Prettyprinters versus Indenters.....	167
9.6	Further Discussion	169
9.7	Related Work	170

9.8	Conclusions	171
9.9	Appendix	171
Chapter 10	UML and Modelica System Modeling with ModelicaML	175
10.1	Introduction	175
10.2	SysML vs. Modelica.....	176
10.3	ModelicaML: a UML profile for Modelica	177
10.3.1	Modelica Class Diagrams	178
10.4	The ModelicaML Integrated Design Environment.....	184
10.4.1	Integrated Design and Development Environment	185
10.4.2	The ModelicaML GMF Model	186
10.4.3	Modeling with Requirements.....	188
10.5	Representing Requirements in Modelica.....	189
10.5.1	Using Modelica Annotations	189
10.5.2	Creating a new Restricted Class: requirement	189
10.6	Conclusion and Future Work.....	190
10.7	Appendix	191
Chapter 11	An Integrated Framework for Model-driven Product Design and Development Using Modelica	193
11.1	Introduction	193
11.2	Architecture overview	195
11.3	Detailed framework description	196
11.3.1	ModelicaXML	196
11.3.2	Modelica Database (ModelicaDB).....	197
11.3.3	FMDesign	198
11.3.4	The Selection and Configuration Tool.....	199
11.3.5	The Automatic Model Generator Tool.....	200
11.4	Conclusions and Future Work	200
11.5	Appendix	202
Part V	Meta-programming and Composition of EOO Languages	205
Chapter 12	ModelicaXML: A ModelicaXML Representation with Applications	207
12.1	Introduction	207
12.2	Related Work.....	209
12.3	Modelica XML Representation	209
12.3.1	ModelicaXML Example	209
12.3.2	ModelicaXML Schema (DTD/XML-Schema)	212
12.4	ModelicaXML and XML Tools	217
12.4.1	The Stylesheet Language for Transformation (XSLT)	217
12.4.2	The Query Language for XML (XQuery).....	218
12.4.3	Document Object Model (DOM).....	219
12.5	Towards an Ontology for the Modelica Language	220
12.5.1	The Semantic Web Languages.....	220
12.5.2	The roadmap to a Modelica representation using Semantic Web Languages	223

12.6	Conclusions and Future work	225
Chapter 13	Composition of XML dialects: A ModelicaXML case study	227
13.1	Introduction.....	227
13.2	Background	228
13.2.1	Modelica and ModelicaXML	228
13.2.2	The Compost Framework.....	230
13.3	COMPOST extension for Modelica.....	233
13.3.1	Overview	233
13.3.2	Modelica Box Hierarchy	234
13.3.3	Modelica Hook Hierarchy	235
13.3.4	Examples of Composition and Transformation Programs.....	237
13.4	Conclusions and Future work	240
13.5	Appendix.....	240
Part VI	Conclusions and Future Work.....	243
Chapter 14	Conclusions and Future Work	245
14.1	Conclusions.....	245
14.2	Future Work Directions	246
Bibliography	249

Table of Figures

Figure 1-1. Thesis structure.	7
Figure 2-1. The Object Management Group (OMG) 4-Layered Model Driven Architecture (MDA).	13
Figure 2-2. Meta-Modeling and Meta-Programming dimensions.	14
Figure 2-3. Hierarchical model of an industrial robot, including components such as motors, bearings, control software, etc. At the lowest (class) level, equations are typically found.	16
Figure 2-4. Number of rabbits – prey animals, and foxes – predators, as a function of time simulated from the predator-prey LotkaVolterra model.	17
Figure 2-5. Connecting two components that have electrical pins.	18
Figure 2-6. OMShell.	20
Figure 2-7. OMNotebook.	20
Figure 2-8. Modelica Development Tooling (MDT).	20
Figure 2-9. MathModelica modeling and simulation environment. (courtesy of MathCore AB).	21
Figure 2-10. Dymola Modeling and Simulation Environment (courtesy of Dynasim AB).	22
Figure 2-11. SimulationX modeling and simulation environment (courtesy of ITI GmbH).	22
Figure 2-12. SOSDT Eclipse Plugin for RML Development.	27
Figure 2-13. SysML diagram taxonomy.	30
Figure 2-14. SysML block definitions.	31
Figure 2-15. Black-box vs. Gray-box (invasive) composition. Instead of generating glue code, composers invasively change the components.	33
Figure 2-16. Invasive composition applied to hooks result in transformation of the underlying abstract syntax tree.	34
Figure 2-17. Integrated model-driven product design and development framework.	35
Figure 3-1. A discrete-time variable z changes value only at event instants, whereas continuous-time variables like y may change both between and at events.	47
Figure 3-2. Abstract syntax tree of the expression $12+5*13$	49
Figure 4-1. MetaModelica Compiler Prototype – compilation phases.	70
Figure 4-2. The stages of translation and execution of a MetaModelica model.	73
Figure 4-3. OpenModelica compiler packages and their connection.	74
Figure 4-4. Pattern Matching Translation Strategy.	79
Figure 4-5. Code Example Generated DFA.	82
Figure 4-6. Exception handling translation strategy.	85
Figure 4-7. OpenModelica implementation.	86
Figure 4-8. Garbage Collection time (s) vs. Execution time (s).	90
Figure 4-9. Garbage Collection time (s).	91
Figure 5-1. Early vs. Late Debugging Instrumentation in MetaModelica compiler.	98
Figure 5-2. Variable value display during debugging using type reconstruction.	100

Figure 5-3. Advanced debugging functionality in MDT.....	105
Figure 5-4. Accessing the debug configuration dialog.....	106
Figure 5-5. Creating the Debug Configuration.	106
Figure 5-6. Specifying the executable to be run in debug mode.	107
Figure 5-7. Setting/deleting breakpoints.	107
Figure 5-8. Starting the debugging session.	108
Figure 5-9. Eclipse will ask if the user wants to switch to the debugging perspective.	108
Figure 5-10. The debugging perspective.	109
Figure 5-11. Switching between perspectives.	109
Figure 6-1. Debugging approach overview.	113
Figure 6-2. Translation stages from Modelica code to executing simulation.	115
Figure 6-3. Translation stages from Modelica code to executing simulation with additional debugging steps.	117
Figure 6-4. Run-time debugging framework overview.	118
Figure 7-1. The <code>rml2c</code> compiler phases.	123
Figure 7-2. Tool coupling within the RML integrated environment with debugging.	125
Figure 7-3. Using breakpoints.	131
Figure 7-4. Stepping and running.	132
Figure 7-5. Examining data.	134
Figure 7-6. Additional debugging commands.	135
Figure 7-7. Browser for variable values showing the current execution point (bottom) and the variable value (top).	137
Figure 7-8. When datatype constructors are selected, the bottom part presents their source code definitions for easy understanding of the displayed values.	138
Figure 8-1. The architecture of Eclipse, with possible plugin positions marked.	148
Figure 8-2. The architecture of the OpenModelica environment.	149
Figure 8-3. The client-server architecture of the OpenModelica environment.	150
Figure 8-4. Creating a new package.	151
Figure 8-5. Creating a new class.	152
Figure 8-6. Syntax checking.	152
Figure 8-7. Code completion using a popup menu after a dot	153
Figure 8-8. Code completion showing a popup function signature after typing a left parenthesis.	154
Figure 8-9. Example of code before indentation.	154
Figure 8-10. Example of code after automatic indentation.	155
Figure 8-11. Plot of the Influenza model.	157
Figure 9-1. AST of the Example.mo file.	165
Figure 9-2. Syntax checking.	169
Figure 10-1. ModelicaML diagrams overview.	177
Figure 10-2. ModelicaML class definitions.	179
Figure 10-3. ModelicaML Internal Class vs. Modelica Connection Diagram.	180
Figure 10-4. Package hierarchy modeling.	181
Figure 10-5. Equation modeling example with a Modelica Class Diagram.	182

Figure 10-6. Simulation diagram example.....	183
Figure 10-7. ModelicaML Eclipse based design environment with a Class diagram.	186
Figure 10-8. ModelicaML GMF Model (Requirements).....	187
Figure 10-9. Modeling with Requirement Diagrams.	188
Figure 10-10. Modeling with requirements (Requirements palette).	191
Figure 10-11. Modeling with requirements (Connections).....	192
Figure 11-1. Design framework for product development.	195
Figure 11-2. Modelica and the corresponding ModelicaXML representation.	197
Figure 11-3. FMDesign – a tool for conceptual design of products.	198
Figure 11-4. FMDesign information model.	202
Figure 11-5. ModelicaDB meta-model.	203
Figure 12-1. The program (root) element of the ModelicaXML Schema.	212
Figure 12-2. The definition element from the ModelicaXML Schema.	213
Figure 12-3. The component element from the ModelicaXML Schema.	214
Figure 12-4. The equation element from the ModelicaXML Schema.	215
Figure 12-5. The algorithm element from the ModelicaXML Schema.	216
Figure 12-6. The expressions from ModelicaXML schema.	216
Figure 12-7. The Semantic Web Layers.	220
Figure 13-1. The layers of COMPOST.	231
Figure 13-2. The XML composition. System Architecture Overview.....	234
Figure 13-3. The Modelica Box Hierarchy defines a set of templates for each language structure.....	235
Figure 13-4. The Modelica Hook Hierarchy.....	236

Index of tables

Table 4-1. Execution time in seconds. The – sign represents out of memory.	71
Table 4-2. Garbage Collection Performance.....	92
Table 5-1. Compilation performance (no debugging vs. early vs. late instrumentation).....	102
Table 5-2. Running performance of script RRLargeModel2.mos.	103
Table 5-3. Running performance of script BouncingBall.mos.	103
Table 5-4. The impact of tracing on execution time.	103
Table 7-1. RML premise types. These constructs are swept for variables to be registered with the debugging runtime system.	127
Table 7-2. Size (#lines) without and with instrumentation.	140
Table 7-3. Running time without and with debugging.	140
Table 7-4. Used stack without and with debugging.....	140
Table 7-5. Number of performed relation calls.....	141

Part I

Motivation, Introduction, Background and Related Work

Chapter 1

Introduction

Motto:

Models..., models everywhere.

Meta-models model models

Meta-MetaModels models Meta-Models.

Attempt at a Definition of the Term "meta-model" (www.metamodel.com):

*A meta-model is a precise definition of the constructs
and rules needed for creating semantic models.*

Integrated development environments are essential for efficient realization of complex industrial products, typically consisting of both software and hardware components. Powerful equation-based object-oriented (EOO) languages such as Modelica are successfully used for modeling and virtual prototyping increasingly complex physical systems and components, whereas software modeling approaches like UML, especially in the form of domain specific language subsets, are increasingly used for software systems modeling.

A research hypothesis investigated to some extent in this thesis is if EOO languages can be successfully generalized also to support software modeling, thus addressing whole product modeling, and if integrated environments for such a generalized EOO language tool support can be created and effectively used on real-sized applications.

However, creating advanced development environments is still a resource-consuming error-prone process that is largely manual. One rather successful approach is to have a general framework kernel, and use meta-modeling and meta-programming techniques to provide tool support for specific languages. Thus, the main goal of this research is the development of a meta-modeling approach and its associated meta-programming methods for the synthesis of model-driven product development environments that includes support for modeling and simulation. Such environments include components like model editors, compilers, debuggers and simulators. This thesis presents several contributions towards this vision in the context of EOO languages, primarily the Modelica language.

1.1 Research Objective (Motivation)

Current state-of-the art equation-based object-oriented languages are supported by tools that have fixed features and are hard to extend. The modeling community needs better tools to support creation, querying, manipulation, composition and simulation of models in equation-based object-oriented languages.

The current state-of-the art tools supporting EOO languages do not satisfy all the different requirements users expect, for example the following:

- Creation, query, manipulation, composition and management of models.
- Query of model equations for: optimization purposes, parallelization, model checking, simulation with different solvers, etc.
- Model configuration for simulation purposes: initial state, initialization via xml files or databases.
- Simulation features: running a simulation and displaying a result, running more simulations in parallel, possibility to handle simulation failures and continue the simulation on a different path, possibility to generate only specific data within a simulation, possibility to manipulate simulation data for export to another tool.
- Model transformation and refactoring: export to a different tool, improve the current model or library but retain the semantics, model composition and invasive model composition.
- Continuous partial differential equations (PDEs) transformed into: Discretized, finite difference, Discretized, Finite Elements (FEM), Discretized, finite volume.

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

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

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

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

An alternative idea is to add features to the modeling language such that for example a model package can contain model analysis and translation features that therefore are not required in the model compiler. An example is a PDEs

discretization scheme that could be expressed in the modeling language itself as part of a PDE package instead of being added internally to the model compiler.

The direct questions arising from the research objective are:

- Can we deliver a new language that allows people to build their own solution to their problems without having to go via tool vendors?
- What is expected from such a language?
- What properties should the language have based on the requirements for it? This includes language primitives, type system, semantics, etc.
- Can such a language combined with a general tool be better than a special-purpose tool?
- What are the steps to design and develop such a language?
- What methods and tools should support the debugging of the new language?
- How can we construct advanced interactive development environments that support such a language?

1.2 Contributions

The integrated model-driven environments and the new MetaModelica language presented in this thesis provide efficient and effective methods for designing and developing complex product models. Methods and tools for debugging, management, serialization, and composition of models are additional contributions.

The research contributions of the thesis are:

- The design, implementation, and evaluation of a new general executable mathematical modeling and semantics meta-modeling language called MetaModelica. The MetaModelica language extends the existing Modelica language with support for meta-modeling, meta-programming, and exception handling facilities.
- The design, implementation and evaluation of advanced portable debugging methods and frameworks for runtime debugging of MetaModelica and semantic specifications.
- The design, implementation and evaluation of several integrated model-driven environments supporting creation, development, refactoring, debugging, management, composition, serialization, and graphical representation of models in EOO languages. Additionally, an integrated model-driven product design and development environment based on EOO languages is also contributed.
- Alternative representation of Modelica EOO models based on XML and UML/SysML are investigated and evaluated. Transformation and invasive composition of EOO models has also been investigated.

1.3 Thesis Structure

In this section we give a short overview of each of the parts in the thesis. At the end of this section we also present visually, in Figure 1-1, an overview of the structure of this thesis.

The thesis consists of six main parts:

- **Part I** presents the thesis motivation, its introduction, the background and related work.
- **Part II** focuses on the design and implementation of an general-purpose unified EOO language called MetaModelica
- **Part III** introduces our work with regards to run-time debugging of meta-programs, equation based languages and semantic specifications.
- **Part IV** presents the design and implementation of several integrated development environments for EOO languages.
- **Part V** presents contributions to serialization, invasive composition and transformation of EOO models.
- **Part VI** concludes the thesis and gives future work directions.

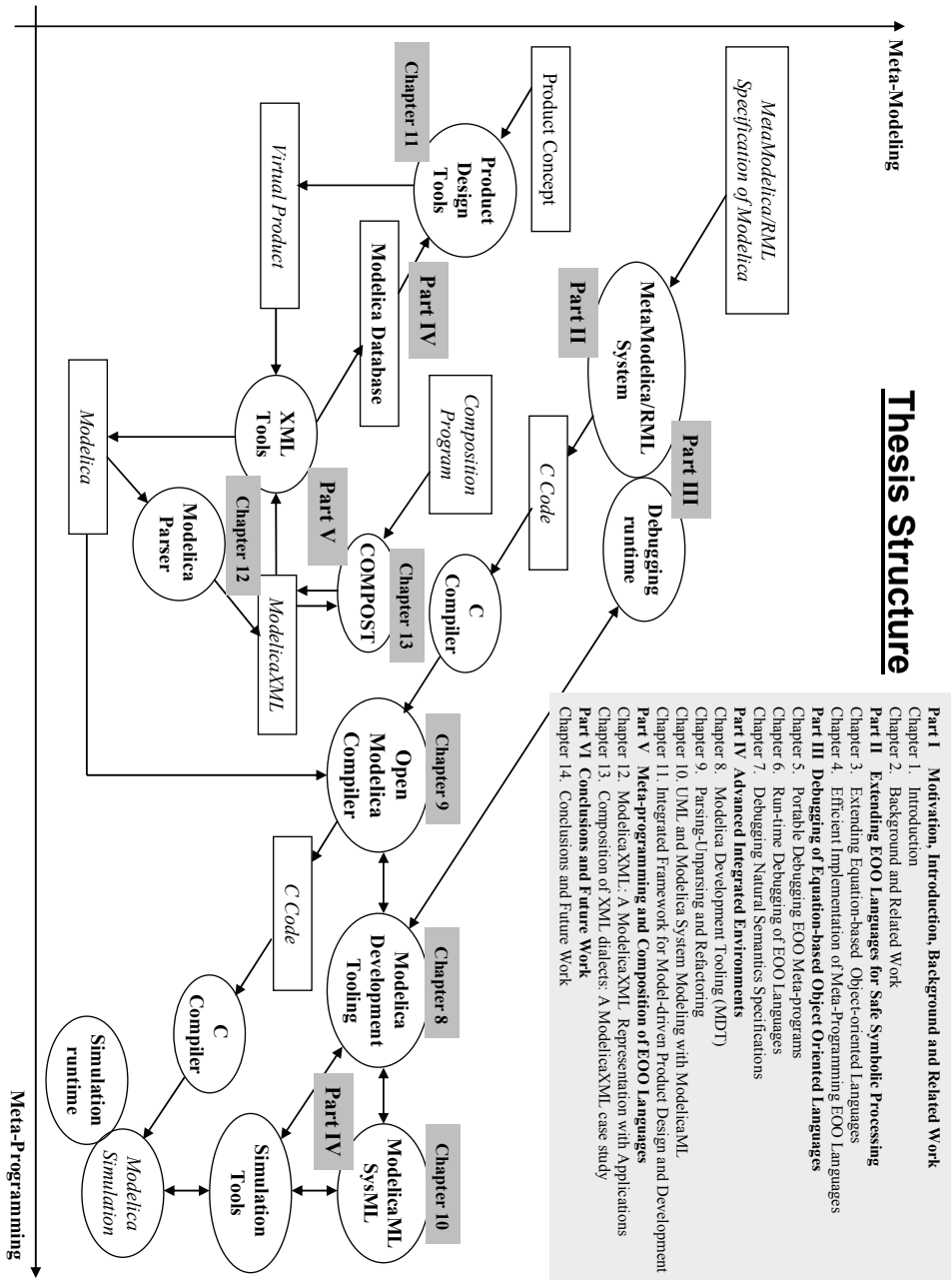


Figure 1-1. Thesis structure.

1.4 Publications

This thesis is partially based on the following publications:

1. Adrian Pop, Kristian Stavåker, Peter Fritzson: *Exception Handling for Modelica*, 6th International Modelica Conference, March 03-04, 2008, Bielefeld, Germany
2. Peter Fritzson, Adrian Pop, Kristoffer Norling, Mikael Blom: *Comment- and Indentation Preserving Refactoring and Unparsing for Modelica*, 6th International Modelica Conference, March 03-04, 2008, Bielefeld, Germany
3. Kristian Stavåker, Adrian Pop, Peter Fritzson: *Compiling and Using Pattern Matching in Modelica*, 6th International Modelica Conference, March 03-04, 2008, Bielefeld, Germany
4. Jörn Guy Süß, Peter Fritzson, Adrian Pop, Luke Wildman: *Towards Integrated Model-Driven Testing of SCADA Systems Using the Eclipse Modeling Framework and Modelica*, 19th Australian Software Engineering Conference (ASWEC 2008), March 26-28, 2008, Perth, Western Australia
5. Adrian Pop, David Akhvlediani, Peter Fritzson: *Integrated UML and Modelica System Modeling with ModelicaML in Eclipse*, The 11th IASTED International Conference on Software Engineering and Applications (SEA 2007), November 19-21, 2007, Cambridge, MA, USA
6. Adrian Pop, Peter Fritzson: *Towards Run-time Debugging of Equation-based Object-oriented Languages*, The 48th Conference on Simulation and Modeling (SIMS 2007), October 30-31, 2007, Goteborg, Sweden
7. Adrian Pop, Vasile Băluță, Peter Fritzson: *Eclipse Support for Design and Requirements Engineering Based on ModelicaML*, The 48th Conference on Simulation and Modeling (SIMS 2007), October 30-31, 2007, Goteborg, Sweden
8. Adrian Pop, David Akhvlediani, Peter Fritzson: *Towards Unified System Modeling with the ModelicaML UML Profile*, EOOLT'2007 - 1st International Workshop on Equation-Based Object-Oriented Languages and Tools, part of ECOOP'2007 - 21st European Conference on Object-Oriented Programming, July 29-August 3, 2007, Berlin, Germany
9. Peter Fritzson, Peter Aronsson, Adrian Pop, Håkan Lundvall, Kaj Nyström, Levon Saldamli, David Broman, Anders Sandholm: *OpenModelica - A Free Open-Source Environment for System Modeling, Simulation, and Teaching*, IEEE International Symposium on Computer-Aided Control Systems Design, October 4-6, 2006, Munich, Germany

10. Elmir Jagudin, Andreas Remar, Adrian Pop, Peter Fritzson: *OpenModelica MDT Eclipse plugin for Modelica Development, Code Browsing, and Simulation*, the 47th Conference on Simulation and Modeling (SIMS2006), September, 28-29, 2006, Helsinki, Finland
11. Adrian Pop, Peter Fritzson: *MetaModelica: A Unified Equation-Based Semantical and Mathematical Modeling Language*, Joint Modular Languages Conference 2006 (JMLC2006), September, 13-15th, 2006, Jesus College, Oxford, England. Also in Lecture Notes in Computer Science, volume 4228, p: 211-229.
12. Adrian Pop, Peter Fritzson, Andreas Remar, Elmir Jagudin, David Akhvlediani: *OpenModelica Development Environment with Eclipse Integration for Browsing, Modeling, and Debugging*, 5th International Modelica Conference (Modelica2006), September, 4-5th, 2006, Vienna, Austria.
13. Olof Johansson, Adrian Pop, Peter Fritzson: *Engineering Design Tool Standards and Interfacing Possibilities to Modelica Simulation Tools*, 5th International Modelica Conference (Modelica2006), September, 4-5th, 2006, Vienna, Austria.
14. Adrian Pop, Peter Fritzson: *An Eclipse-based Integrated Environment for Developing Executable Structural Operational Semantics Specifications*, Structural Operational Semantics 2006 (SoS'2006), a Satellite Workshop of The 17th International Conference on Concurrency Theory (CONCUR'2006), August 26, 2006, Bonn, Germany.
15. Adrian Pop: *Contributions to Meta-Modeling Tools and Methods*, Licentiate Thesis No. 1162, Linköping University, June 3, 2005
16. Adrian Pop, Peter Fritzson: *Debugging Natural Semantics Specifications*, Sixth International Symposium on Automated and Analysis-Driven Debugging (AADEBUG2005), September 19-21, 2005, Monterey, California. Published in the ACM SIGSOFT/SIGPLAN.
17. Adrian Pop, Peter Fritzson: *A Portable Debugger for Algorithmic Modelica Code*, the 4th International Modelica Conference (Modelica2005), March 7-9, 2005, Hamburg, Germany. Published in the local proceedings and online.
18. Olof Johansson, Adrian Pop, Peter Fritzson: *ModelicaDB - A Tool for Searching, Analysing, Crossreferencing and Checking of Modelica Libraries*, the 4th International Modelica Conference (Modelica2005), March 7-9, 2005, Hamburg, Germany. Published in the local proceedings and online.
19. Peter Fritzson, Adrian Pop, Peter Aronsson: *Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica*, the 4th

- International Modelica Conference (Modelica2005), March 7-9, 2005, Hamburg, Germany. Published in the local proceedings and online.
20. Ilie Savga, Adrian Pop, Peter Fritzson: *Deriving a Component Model from a Language Specification: An Example Using Natural Semantics*, Internal Report, December, 2005.
 21. Adrian Pop, Peter Fritzson: *The Modelica Standard Library as an Ontology for Modeling and Simulation of Physical Systems*, Internal Report, August, 2004.
 22. Adrian Pop, Ilie Savga, Uwe Abmann, Peter Fritzson: *Composition of XML dialects: A ModelicaXML case study*, Software Composition Workshop (SC2004) , affiliated with European Joint Conferences on Theory and Practice of Software (ETAPS'04) , March 27 - April 4, 2004, Barcelona, Spain. The paper can be found in Electronic Notes in Theoretical Computer Science Volume 114, 17 January 2005, Pages 137-152, Proceedings of the Software Composition Workshop (SC 2004)
 23. Olof Johansson, Adrian Pop, Peter Fritzson: *A functionality coverage analysis of industrially used ontology languages*, Model Driven Architecture: Foundations and Applications (MDAFA2004), June 10-11, 2004, Linköping, Sweden
 24. Adrian Pop, Olof Johansson, Peter Fritzson: *An integrated framework for model-driven design and development using Modelica*, SIMS 2004, the 45th Conference on Simulation and Modeling, September 23-24, 2004, Copenhagen, Denmark. Complete proceedings can be found at: ScanSims.org
 25. Adrian Pop, Peter Fritzson: *ModelicaXML: A Modelica XML Representation with Applications*, Modelica 2003, The 3rd International Modelica Conference, November 3-4, Linköping, Sweden

Chapter 2

Background and Related Work

2.1 Introduction

The research work in this thesis is cross-cutting several research fields, which we introduce in this section. Here we give a more detailed presentation of the specific background and related work of the several areas in which we address problems.

2.1.1 Systems, Models, Meta-Models, and Meta-Programs

Understanding existing *systems* or building new ones is a complex process. When dealing with this complexity people try to break large systems into manageable pieces. In order to experiment with systems people create *models* that can answer questions about specific system properties. As a simple example of a system we can take a fish; our mental model of a fish is our internal mind representation, experiences, and beliefs about this system. In other words, a model is an abstraction of a system which mirrors parts or all its characteristics we are interested in. Models are created for various reasons from proving that a particular system can be built to understanding complex existing systems. Modeling – the process of model creation – is often followed by simulation performed on the created models. A simulation can be regarded as an experiment applied on a model.

Meta-modeling is still a modeling activity but its aim is to create *meta-models*. A meta-model is one level of abstraction higher than its described models.

- If a model MM is used to describe a model M , then MM is called the *meta-model* of M .
- Alternatively one can consider a meta-model as the description of the syntax and/or meaning (semantics) of concepts that are used in the underlying level to construct models (model families).

The usefulness of meta-models highly depends on the purpose for which they are created and what they attempt to describe. In general, a meta-model can be regarded as:

- A schema for data (here data can mean anything from information to programs, models, meta-models, etc) that needs to be exchanged, stored, or transformed.
- A language that is used to describe a specific process or methodology.
- A language for expressing (additional) meaning (semantics) or syntax of existing information, e.g. information present on the World Wide Web (WWW).

Thus, meta-models are ways to express and share some kind of knowledge that helps in the design and management of models.

When the models are programs, the programs that manipulate them are called *meta-programs* and the process of their creation is denoted as *meta-programming*. As examples of meta-programs we can include program generators, interpreters, compilers, static analyzers, and type checkers. In general meta-programs do not act on the source code directly but on a representation (model) of the source code, such as *abstract syntax trees*. The abstract syntax trees together with the meta-program that manipulates them can be regarded as a meta-model.

One can make a distinction between *general purpose modeling* and *domain specific modeling*, for example physical systems modeling. General purpose modeling is concerned with expressing and representing any kind of knowledge, while domain specific modeling is targeted to specific domains. Lately, approaches that use general purpose modeling languages (meta-metamodels) to define domain specific modeling languages (meta-models) together with their environments have started to emerge. The meta-metamodeling methodology is used to specify such approaches.

Combining different models that use different formalisms and different levels of abstraction to represent aspects of the same system is highly desirable. Computer aided *multi-paradigm modeling* is a new emerging field that is trying to define a domain independent framework along several dimensions such as multiple levels of abstraction, multi-formalism modeling, meta-modeling, etc.

2.1.2 Meta-Modeling and Meta-Programming Approaches

Hardly anyone can speak of general purpose modeling without mentioning the Unified Modeling Language (UML) (OMG [115]). UML is by far the most used specification language used for modeling. UML together with the Meta-Object Facility (MOF) (OMG [112]) forms the bases for the Model-Driven Architecture (MDA) (OMG [113]) which aims at unifying the design, development, and integration of system modeling. The architecture has four layers, called M0 to M3 presented in Figure 2-1 and below:

- M3 is the meta-metamodel which is an instance of itself.
- M2 is the level where the UML meta-model is defined. The concepts used by the designer, such as Class, Attribute, etc., are defined at this level.
- M1 is the level where the UML models and domain-specific extensions of the UML language reside.
- M0 is the level where the actual user objects reside (the world).

An instance at a certain level is always an instance of something defined at one level higher. An actual object at M0 is an instance of a class defined at M1. The classes defined in UML models at M1 are instances of the Class concept defined at M2. The UML meta-model itself is an instance of M3. Other meta-models that define other modeling languages are also instances of M3.

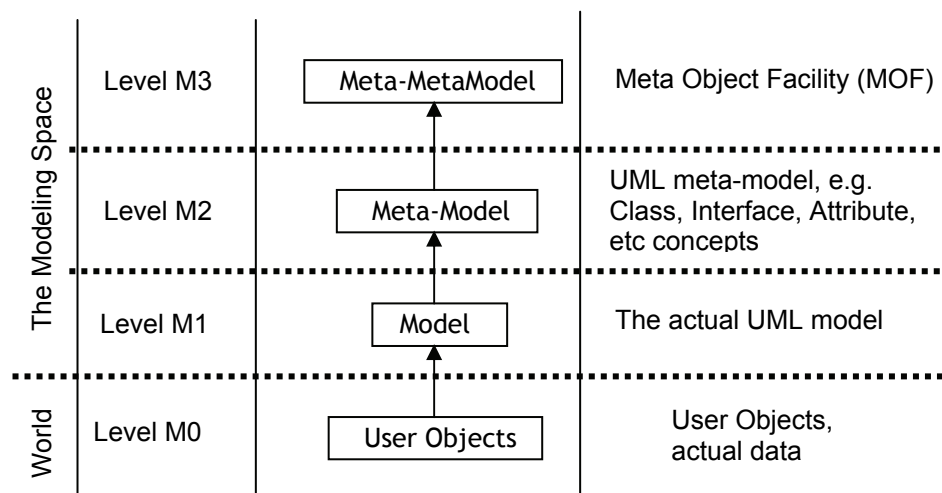


Figure 2-1. The Object Management Group (OMG) 4-Layered Model Driven Architecture (MDA).

Within the MDA framework, UML Profiles are used to tailor the general UML language to specific areas (domain specific modeling).

Modeling environment configuration approaches similar to the UML Profiles, are present within the Generic Modeling Environment (GME) (Ledeczi et al. 2001 [82], Ledeczi et al. 2001 [83]) which is a configurable toolkit for creating domain-specific modeling and program synthesis environments. Here, the configuration is accomplished through meta-models specifying the modeling paradigm (modeling language) of the application domain.

Computer-aided Multi-paradigm Modeling and Simulation (CaMpaM) (Lacoste-Julien et al. 2004 [79], Lara et al. 2003 [80]) supported by tools such as the ATOM³ environment (A Tool for Multi-formalism and Meta-Modeling) (Vangheluwe and Lara 2004 [170]) is aiming at combining several dimensions of modeling (levels of abstractions, multi-formalisms and meta-modeling) in order to configure environments tailored for specific domains.

We have already described what meta-modeling and meta-programming are. From another point of view meta-modeling and meta-programming are orthogonal solutions to system modeling (Figure 2-2) that can be combined to achieve model definition and transformation at several abstraction levels.

By using meta-programming it is possible to achieve transformation between models or meta-models. The meta-models one level up can be used to enforce the correctness of the transformation. Translation and transformation between models are highly desirable as new models appear and solutions to system modeling require different modeling languages and formalisms together with their environments.

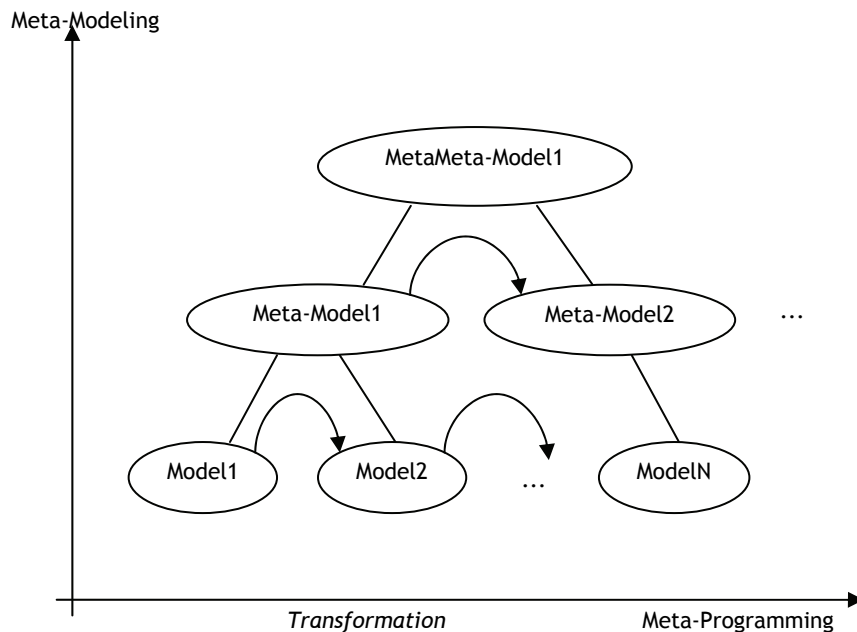


Figure 2-2. Meta-Modeling and Meta-Programming dimensions.

2.2 The Modelica Language

Starting 1989, our group developed an equation-based specification language for mathematical modeling called ObjectMath (Fritzson et al. 1995 [53], Viklund et al. 1992 [173]), using Mathematica as a basis and a frontend, but adding object orientation and efficient code generation. Following this path, in 1996 our group joined efforts with several other groups in object-oriented mathematical modeling to start a design-group for developing an internationally viable declarative mathematical modeling language. This language, called *Modelica*, has been designed by the Modelica Design Group, initially consisting mostly of the developers of a number of different equation-based object-oriented modeling

languages like Allan, Dymola, NMF, ObjectMath, Omola, SIDOPS+, Smile, as well as other modeling and simulation experts. In February 2000, a non-profit organization named “Modelica Association” was founded in Linköping, Sweden, for further development and promotion of Modelica. Modelica (Elmqvist et al. 1999 [35], Fritzson 2004 [44], Fritzson and Engelson 1998 [50], Modelica.Association 1996-2008 [99], Tiller 2001 [152]) is an object-oriented modeling language for declarative equation-based mathematical modeling of large and heterogeneous physical systems. For modeling with Modelica, commercial software products such as MathModelica (MathCore [91]) or Dymola (Dynasim 2005 [27]) have been developed. Also open-source implementations like the OpenModelica system (Fritzson et al. 2002 [46], PELAB 2002-2008 [118]) are available.

The Modelica language has been designed to allow tools to automatically generate efficient simulation code with the main objective of facilitating exchange of models, model libraries, and simulation specifications. The definition of simulation models is expressed in a declarative manner, modularly and hierarchically. Various formalisms can be expressed in the more general Modelica formalism. In this respect Modelica has a multi-domain modeling capability which gives the user the possibility to combine electrical, mechanical, hydraulic, thermodynamic, etc., model components within the same application model. Compared to most other modeling languages available today, Modelica offers several important advantages from the simulation practitioner’s point of view:

- *Object-oriented mathematical modeling.* This technique makes it possible to create model components, which are employed to support hierarchical structuring, reuse, and evolution of large and complex models covering multiple technology domains. *A general type system* that unifies object-orientation, multiple inheritance, and generics templates within a single class construct. This facilitates reuse of components and evolution of models.
- *Acausal modeling* based on ordinary differential equations (ODE) and differential algebraic equations (DAE) together with discrete equations forming a hybrid DAE.. There is also ongoing research to include partial differential equations (PDE) in the language syntax and semantics (Saldamli et al. 2002 [142]), (Saldamli 2002 [140], Saldamli et al. 2005 [141]).
- *Multi-domain modeling* capability, which gives the user the possibility to combine electrical, mechanical, thermodynamic, hydraulic etc., model components within the same application model.
- *A strong software component model*, with constructs for creating and connecting components. Thus the language is ideally suited as an architectural description language for complex physical systems, and to some extent for software systems.
- *Visual drag & drop and connect composition of models* from components present in different libraries targeted to different domains (electrical, mechanical, etc).

The language is strongly typed and declarative. See (Modelica.Association 1996-2008 [99]), (Modelica-Association 2005 [101]), (Tiller 2001 [153]), and (Fritzson 2004 [44]) for a complete description of the language and its functionality from the perspective of the motivations and design goals of the researchers who developed it. Shorter overviews of the language are available in (Elmqvist et al. 1999 [35]), (Fritzson and Engelson 1998 [50]), and (Fritzson and Bunus 2002 [49]).

The Modelica component model includes the following three items: a) components, b) a connection mechanism, and c) a component framework. *Components* are connected via the *connection mechanism* realized by the Modelica system, which can be visualized in connection diagrams. The *component framework* realizes components and connections, and ensures that communication works over via the connections.

For systems composed of *acausal* components with behavior specified by equations, the direction of data flow, i.e., the *causality* is initially unspecified for connections between those components. Instead the causality is automatically deduced by the compiler when needed. Components have well-defined *interfaces* consisting of ports, also known as *connectors*, to the external world. A component may internally consist of other connected components, i.e., *hierarchical* modeling is possible. Figure 2-3 shows a hierarchical component-based modeling of an industry robot.

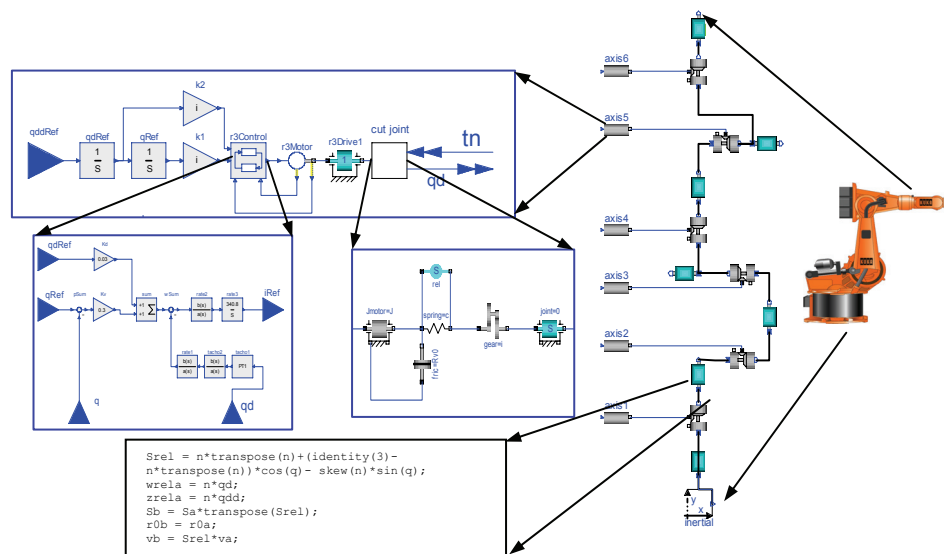


Figure 2-3. Hierarchical model of an industrial robot, including components such as motors, bearings, control software, etc. At the lowest (class) level, equations are typically found.

2.2.1 An Example Modelica Model

The following is an example Lotka Volterra Modelica model containing two differential equations relating the sizes of rabbit and fox populations which are represented by the variables `rabbits` and `foxes`: The model was independently developed by Alfred J Lotka (1925) and Vito Volterra (1926): The rabbits multiply (by breeding); the foxes eat rabbits. Eventually there are enough foxes eating rabbits causing a decrease in the rabbit population, etc., causing cyclic population sizes. The model is simulated and the sizes of the rabbit and fox populations are plotted in Figure 2-4 as a function of time.

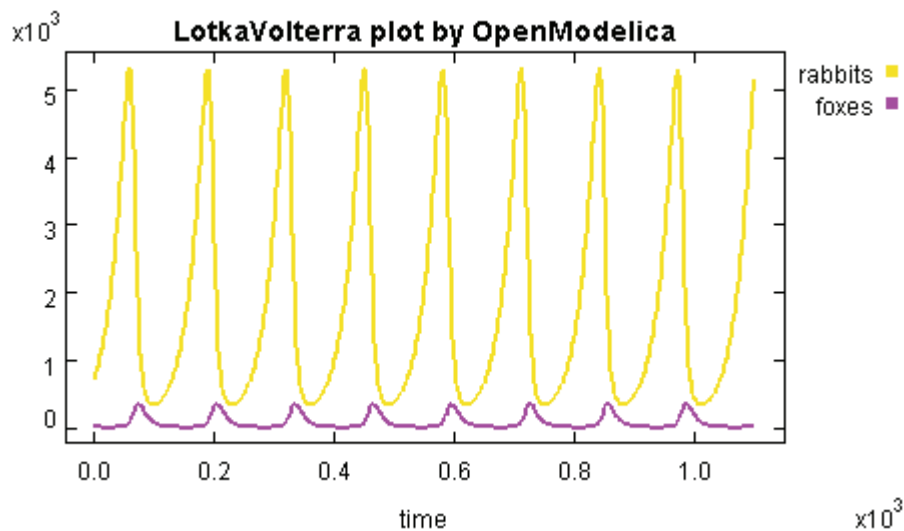


Figure 2-4. Number of rabbits – prey animals, and foxes – predators, as a function of time simulated from the predator-prey LotkaVolterra model.

The notation `der(rabbits)` means time derivative of the `rabbits` (population) variable.

```

model LotkaVolterra
  parameter Real g_r = 0.04 "Natural growth rate for rabbits";
  parameter Real d_rf = 5e-5 "Death rate of rabbits due to
    foxes";
  parameter Real d_f = 0.09 "Natural death rate for foxes";
  parameter Real g_fr = 0.1 "Efficiency in growing foxes from
    rabbits";
  Real rabbits(start=700) "Rabbits with start population 700";
  Real foxes(start=10) "Foxes, with start population 10";
equation
  der(rabbits) = g_r*rabbits - d_rf*rabbits*foxes;
  der(foxes) = g_fr*d_rf*rabbits*foxes - d_f*foxes;
end LotkaVolterra;

```

2.2.2 Modelica as a Component Language

Modelica offers quite a powerful software component model that is on par with hardware component systems in flexibility and potential for reuse. The key to this increased flexibility is the fact that Modelica classes are based on equations, i.e., acausal connections for which the direction of data flow across the connection is not fixed. Components are connected via the connection mechanism, which can be visualized in connection diagrams. The component framework realizes components and connections, and ensures that communication works and constraints are maintained over the connections. For systems composed of acausal components the direction of data flow, i.e., the causality is automatically deduced by the compiler at composition time.

Two types of coupling can be established by connections depending on whether the variables in the connected connectors are non-flow (default), or declared using the `flow` prefix:

1. Equality coupling, for non-flow variables, according to Kirchhoff's first law.
2. Sum-to-zero coupling, for flow variables, according to Kirchhoff's current law.

For example, the keyword `flow` for the variable `i` of type `Current` in the `Pin` connector class indicates that all currents in connected pins are summed to zero, according to Kirchhoff's current law.

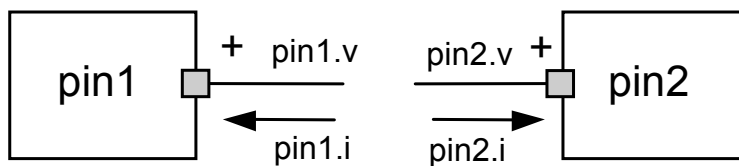


Figure 2-5. Connecting two components that have electrical pins.

Connection equations are used to connect instances of connection classes. A connection equation `connect(pin1, pin2)`, with `pin1` and `pin2` of connector class `Pin`, connects the two pins (Figure 2-5) so that they form one node. This produces two equations, namely:

$$\begin{aligned} \text{pin1.v} &= \text{pin2.v} \\ \text{pin1.i} + \text{pin2.i} &= 0 \end{aligned}$$

The first equation says that the voltages of the connected wire ends are the same. The second equation corresponds to Kirchhoff's second law, saying that the currents sum to zero at a node (assuming positive value while flowing into the component). The sum-to-zero equations are generated when the prefix `flow` is used. Similar laws apply to flows in piping networks and to forces and torques in mechanical systems.

2.3 Modelica Environments

For modeling with Modelica, commercial software products such as MathModelica (MathCore [91]) (Figure 2-9), Dymola (Dynasim 2005 [27]) or SimulationX (ITI.GmbH 2008 [71]) have been developed. Also open-source implementations like the OpenModelica system (Fritzson et al. 2002 [46], Fritzson et al. 2005 [47], PELAB 2002-2008 [118]) are available.

2.3.1 OpenModelica

The OpenModelica environment is a complete Modelica modeling, compilation and simulation environment based on free software distributed in binary and source code form. The components of the OpenModelica environment are:

- *OpenModelica Interactive Compiler (OMC)* is the core component of the environment. OMC provides advanced interactive functionality for model management: loading, instantiation, query, checking and simulation. The OMC functionality is available via command line scripting or - when run as a server - via the CORBA (OMG [111]) (or socket) interface. The other environment components presented below are using OMC as a server to access its functionality.
- *OMShell* is an interactive command handler that provides very basic functionality for loading and simulation of models.
- *OMNotebook* adds interactive notebook functionality (similar to the Mathematica environment) to the environment. OMNotebook documents blend together evaluation cells with explanation text. The evaluation cells can be executed directly in the notebook and their results incorporated. The OMNotebook component is very useful for teaching, model explanation and documentation because all the information regarding a model (including simulation results) can be included in the same document.
- *Modelica Development Tooling (MDT)* is an Eclipse plug-in that integrates the OpenModelica compiler with Eclipse. MDT, together with the OpenModelica compiler, provides an environment for working with Modelica and MetaModelica projects. Advanced textual (code browsing, syntax highlighting, syntax checking, code completion and assistance, automatic code indentation, etc) and UML/SysML editing features for developing models are available. The environment also provides debugging features.

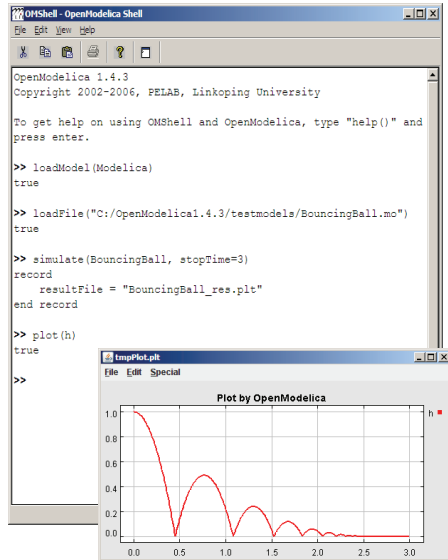


Figure 2-6. OMShell

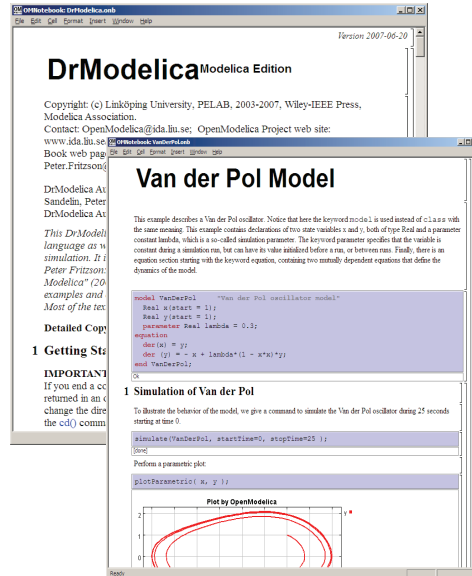


Figure 2-7. OMNotebook

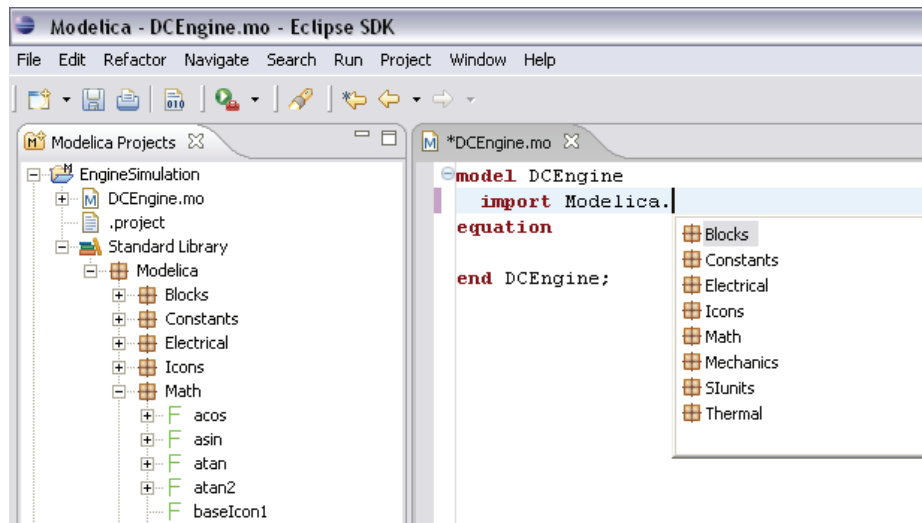


Figure 2-8. Modelica Development Tooling (MDT).

2.3.2 MathModelica, Dymola, SimulationX

MathModelica is an integrated problem-solving environment (PSE) for full system modeling and simulation (Fritzson 2006 [45]). The environment integrates

Modelica-based modeling and simulation with graphic design, advanced scripting facilities, integration of code and documentation, and symbolic formula manipulation provided via Mathematica (Wolfram 2008 [175]). The MathModelica environment is based on the OpenModelica compiler (OMC) but also provides additional commercial capabilities like graphical editor and simulation center.

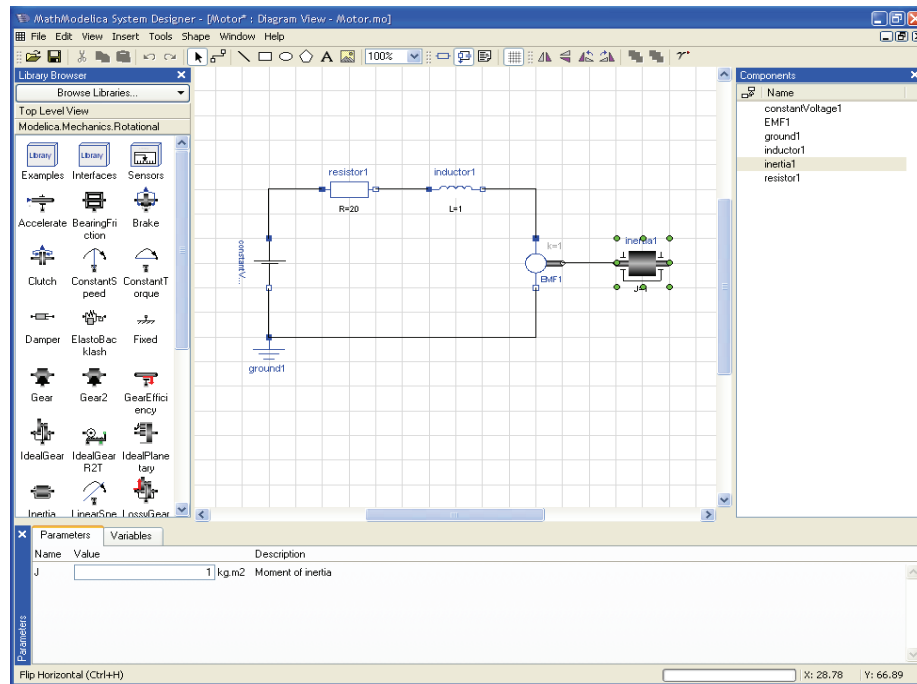


Figure 2-9. MathModelica modeling and simulation environment. (courtesy of MathCore AB)

Dymola (Dynamic Modeling Laboratory) described by (Elmqvist et al. 2003 [34]) is probably one of the most well known multi-domain modeling and simulation environments that supports the Modelica language.

The environment allows the analysis of complex systems that incorporate mechanical, hydraulic, electrical, and thermal components as well as control systems. Dymola does not feature any debugging techniques for possible structural and numerical errors.

For dynamic debugging the simulation environment offers the possibility of logging discrete events. This functionality is useful in tracking down errors in the discrete part of hybrid system models.

The analysis facilities of Dymola concentrate more on profiling. Details of execution times for each block are available. Numeric model instabilities have to be detected in Dymola by directly examining the simulation results.

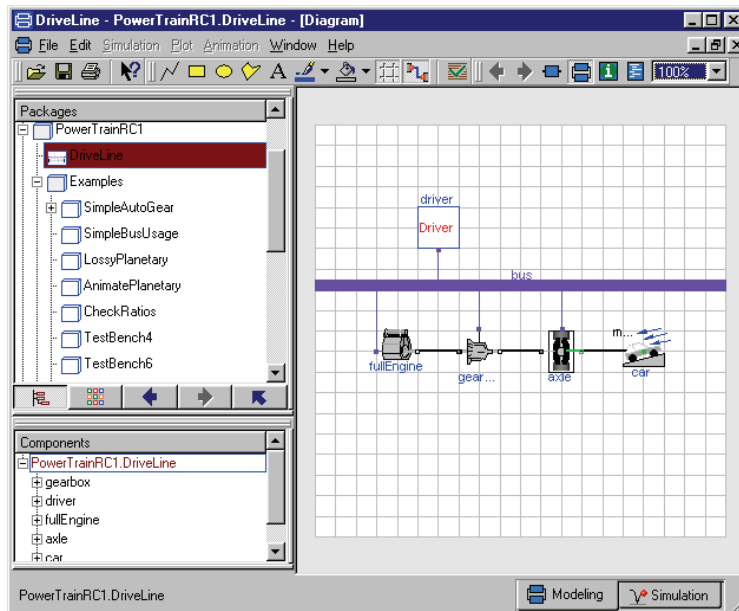


Figure 2-10. Dymola Modeling and Simulation Environment (courtesy of Dynasim AB).

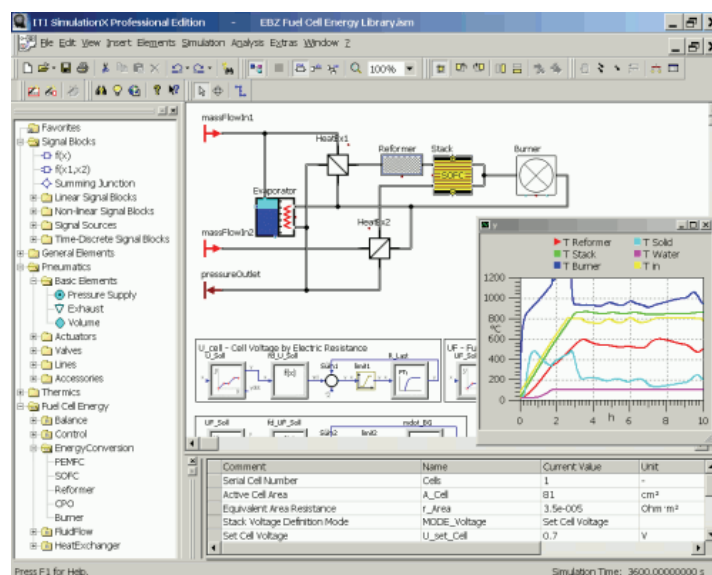


Figure 2-11. SimulationX modeling and simulation environment (courtesy of ITI GmbH)

SimulationX is a software environment for valuation of the interaction of all components of technical systems. SimulationX provides a CAE tool for modeling,

simulation and analyzing of physical effects – with ready-to-use model libraries for 1D mechanics, 3D multibody systems, power transmission, hydraulics, pneumatics, thermodynamics, electrics, electrical drives, magnetics as well as controls – post processing included.

2.4 Related Equation-based languages: gProms, VHDL-AMS and the χ language

In the area of mathematical modeling the most important general de-facto standards for different dynamic simulation modes are:

- Continuous: Matlab/Simulink, MatrixX/SystemBuild, Scilab/Scicos for general systems, SPICE and its derivatives for electrical circuits, ADAMS, DADS/Motion, SimPack for multi-body mechanical systems.
- Discrete: general-purpose simulators based on the discrete-event GPSS line, VHDL- and Verilog simulators in digital electronics, etc.
- Hybrid (discrete + continuous): Modelica/Dymola, AnyLogic, VHDL-AMS and Verilog-AMS simulators (not only for electronics but also for multi-physics problems).

The insufficient power and generality of the former modeling languages stimulated the development of Modelica (as a true object-oriented, multi-physics language) and VHDL-AMS/Verilog-AMS (multi-physics but strongly influenced by electronics).

The rapid increase in new requirements to handle the dynamics of highly complex, heterogeneous systems requires enhanced efforts in developing new language features (based on existing languages!). Especially the efficient simulation of hardware-software systems and model structural dynamics are yet unsolved problems. In electronics and telecommunications, therefore, the development of SystemC-AMS has been launched but these attempts are far from the multi-physics and multi-domain applications which are addressed by Modelica.

gProms (general Process Modeling Systems) (Min and Pantelides 1996 [98]) provides a set of advanced tools for supporting model development and maintenance. Several techniques are provided for model validation, dynamic optimization, optimal experiment design, and life cycle modeling, but unfortunately gProms lacks support for debugging simulation models when structural or numerical failures occur.

VHDL-AMS (Christen and Bakalar 1999 [22]) is the IEEE-endorsed standard modeling language (standard 1076.1) created to provide a general-purpose, easily exchangeable and open language for modern analog-mixed-signal designs. Models can be exchanged between all simulation tools that adhere to the VHDL-AMS standard. Advantages of VHDL-AMS are:

- Model Exchangeability. Free exchange of information between VHDL-AMS simulation tools.
- Multi-level modeling. Different levels of abstraction of model behavior.
- Multi-domain modeling. Offers solutions in different application domains.
- Mixed-signal modeling. Supports analog, digital, and mixed signal modeling.
- Multiple modeling styles. Behavioral, dataflow, structural modeling methods.

The χ language (Fábián 1999 [37]) is a hybrid specification formalism, suitable for the description of discrete-event, continuous-time, and hybrid systems. It is a concurrent language, where the discrete-event part is based on Communicating Sequential Processes (Hoare 1985 [65]) and the continuous-time part is based on Differential Algebraic Equations (DAEs). Models written in the χ language can be executed by the χ simulator.

2.5 Natural Semantics and the Relational Meta-Language (RML)

Concerning *specification languages for programming language semantics*, compiler generators based on denotational semantics (Pettersson and Fritzson 1992 [123]) (Ringström et al. 1994 [137]), have been investigated and developed with some success. However this formalism has certain usability problems, and Operational Semantics/Natural Semantics has become the dominant formalism in common language semantics specification literature.

Therefore a meta-language and compiler generator called RML (Relational Meta Language) (Fritzson 1998 [43], PELAB 1994-2008 [117], Pettersson 1995 [120], Pettersson 1999 [122]) for Natural Semantics was developed, which we have used extensively for full-scale specifications of languages like Java 1.2 (Holmén 2000 [66]), C subset with pointer arithmetic, functional, and equation-based object-oriented languages (Modelica). Generated implementations are comparable in performance to hand implementations. However, it turned out that development environment support is needed also for specification languages. Recent developments include a debugger for Natural Semantics specifications (Pop and Fritzson 2005 [127]) and (Chapter 7).

Natural Semantics (Kahn 1988 [75]) is a specification formalism that is used to specify the semantics of programming languages, i.e., type systems, dynamic semantics, translational semantics, static semantics (Despeyroux 1984 [25], Glesner and Zimmermann 2004 [55]), etc. Natural Semantics is an operational semantics derived from the Plotkin (Plotkin 1981 [125]) structural operational semantics combined with the sequent calculus for natural deduction. There are few systems implemented that compile or interpret Natural Semantics.

One of these systems is Centaur (Borrás et al. 1988 [15]) with its implementation of Natural Semantics called Typol (Despeyroux 1984 [25], Despeyroux 1988 [26]). This system is translating the Natural Semantics inference rules to Prolog.

The Relational Meta-Language (RML) is an efficient implementation of Natural Semantics, with a performance of the generated code that is several orders of magnitude better than Typol. The RML language is compiled to highly efficient C code by the `rml2c` compiler. In this way large parts of a compiler can be automatically generated from their Natural Semantics specifications. RML is successfully used for specifying and generating practically usable compilers from Natural Semantics for Java, Modelica, MiniML (Clément et al. 1986 [23]), Mini-Freja (Pettersson 1995 [120]) and other languages.

2.5.1 An Example of Natural Semantics and RML

As a simple example of using Natural Semantics and the Relational Meta-Language (RML) we present a trivial expression (Exp1) language and its specification in Natural Semantics and RML. A specification in Natural Semantics has two parts:

- Declarations of syntactic and semantic objects involved.
- Groups of inference rules which can be grouped together into relations.

In our example language we have expressions built from numbers. The abstract syntax of this language is declared in the following way:

integers:

$$v \in Int$$

expressions (abstract syntax):

$$e \in Exp ::= v \mid e1 + e2 \mid e1 - e2 \mid e1 * e2 \mid e1 / e2 \mid -e$$

The inference rules for our language are bundled together in a judgment $e \Rightarrow v$ in the following way (here we do not present similar rules for the other operators.):

$$(1) \quad v \Rightarrow v$$

$$(2) \quad \frac{e1 \Rightarrow v1 \quad e2 \Rightarrow v2 \quad v1 + v2 \Rightarrow v3}{e1 + e2 \Rightarrow v3}$$

RML modules have two parts, an interface comprising datatype declarations (abstract syntax) and signatures of relations (judgments) that operate on such datatypes, followed by the declarations of the actual relations which group together rules and axioms. In RML, the Natural Semantics specification shown above is represented as follows:

```

module Exp1:

  (* Abstract syntax of the language Exp1 *)
  datatype Exp = RCONST of real
                | ADD    of Exp * Exp
                | SUB    of Exp * Exp
                | MUL    of Exp * Exp
                | DIV    of Exp * Exp
                | NEG    of Exp

  relation eval: Exp => real
end

  (* Evaluation semantics of Exp1 *)
  relation eval: Exp => real =

    (* Evaluation of a real node is the real number itself *)
    axiom eval(RCONST(rval)) => rval

    (* Evaluation of an addition node ADD is v3, if v3 is
       the result of adding the evaluated results of its
       children e1 and e2. *)
    rule eval(e1) => v1 & eval(e2) => v2 & v1 + v2 => v3
      -----
      eval( ADD(e1, e2) ) => v3

    rule eval(e1) => v1 & eval(e2) => v2 & v1 - v2 => v3
      -----
      eval( SUB(e1, e2) ) => v3

    rule eval(e1) => v1 & eval(e2) => v2 & v1 * v2 => v3
      -----
      eval( MUL(e1, e2) ) => v3

    rule eval(e1) => v1 & eval(e2) => v2 & v1 / v2 => v3
      -----
      eval( DIV(e1, e2) ) => v3

    rule eval(e) => v & -v => vneg
      -----
      eval( NEG(e) ) => vneg

  end (* eval *)

```

A proof-theoretic interpretation can be assigned to this specification. We interpret inference rules as recipes for constructing proofs. We wish to prove that there is a value v such that $1 + 2 \Rightarrow v$ holds for this specification. To prove this proposition we need an inference rule that has a conclusion, which can be instantiated (matched) to the proposition. The only proposition that matches is the second proposition (2), which is instantiated as follows:

$$\frac{1 \Rightarrow v1 \quad 2 \Rightarrow v2 \quad v1 + v2 \Rightarrow v}{1 + 2 \Rightarrow v}$$

To continue the proof, we need to apply the first proposition (axiom) several times, and we soon reach the conclusion. One can observe that debugging of Natural Semantics comprise proof-tree understanding.

2.5.2 Specification of Syntax

Regarding the specification of lexical and syntatic rules for a new language, we use external tools such as Lex, Yacc, Flex, Bison, etc., to generate those modules. The parser builds abstract syntax by calling RML-defined constructors. The abstract syntax is then passed from the parser to the RML-generated modules. We currently use the same approach for languages defined using MetaModelica.

2.5.3 Integrated Environment for RML

The SOSDT (Structural Operational Semantics Development Tooling) is an integrated environment for RML (Figure 2-12).

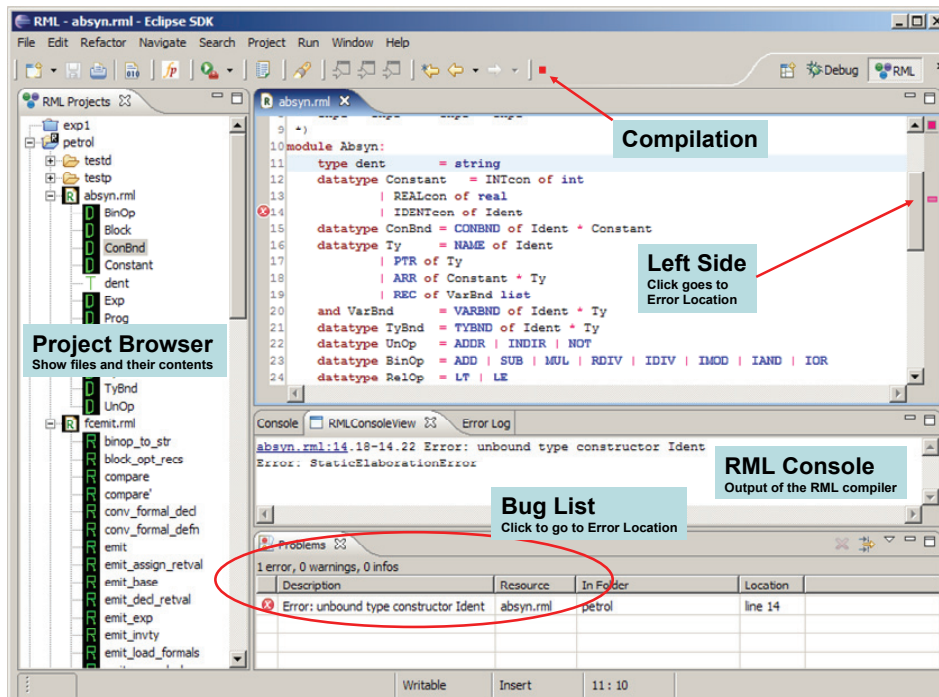


Figure 2-12. SOSDT Eclipse Plugin for RML Development.

The SOSDT environment (Pop and Fritzson 2006 [129]) includes support for browsing, code completion through menus or popups, code checking, automatic indentation, and debugging of specifications.

2.6 The eXtensible Markup Language (XML)

The Extensible Markup Language (XML) (W3C [158]) is a standard recommended by the World Wide Web Consortium (W3C). XML is a simple and flexible text format derived from Standardized Generalized Markup Language (SGML) (W3C [163]). The XML language is widely used for information exchange over the Internet. The tools one can use for parsing, querying, transforming or validating XML documents have reached a mature state. Such tools exist both as open-source projects and commercial software products.

A small example of an XML document is shown below:

```
<?xml version="1.0"?>
<!DOCTYPE persons SYSTEM "persons.dtd">
<persons>
  <person job="programmer">
    <name>John Doe</name>
    <email>
      grigore@none.ro
    </email>
  </person>
  ...
  <person job="manager">
    <comment>Classified</comment>
  </person>
</persons>
```

An XML document is simply a text in which the information is marked up using tags. The tags are the names enclosed in angle brackets. For easy identification we show *elements* in **bold** face and *attribute* names in *italics* throughout the XML example. The information delimited by **<persons>** and **</persons>** tags is an XML element. As we can see, it can contain other elements called **<person>** that nests additional elements within itself.

The attributes are specified after the tag as an unordered name/value list of name="value" items. In our example, the attribute *job* with the value "programmer".

The first line states that this is an XML document. The second line expresses that an XML parser must validate the contents of the elements against the Document Type Definition (DTD) (W3C [158]) file, here named "persons.dtd". The DTD provides constraints for the contents much like grammars used for programming languages.

There are two criteria to be met in order for an XML document to be valid. First, all the elements have to be properly nested and must have a start/end tag. Second, all the contents of all elements must obey their DTD grammar specifications.

We will define a DTD for the above example:

```
<!-- the person.dtd file -->
<!ENTITY % person-job-attribute
    "job(programmer|manager) #REQUIRED">
<!ELEMENT persons (person*)>
<!ELEMENT person ((name+, email*) | comment+)>
<!ATTLIST person
    project CDATA #IMPLIED
    &person-job-attribute;
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT comment (#PCDATA)>
```

The above DTD defines one entity, four elements, and one attribute list containing two attributes. The entities are underlined, **bold** is used for elements, and attributes are specified in *italics*.

The entity (ENTITY) declaration defines person-job-attribute as a text value that can be used anywhere inside the DTD and the XML document. The XML parser will replace the entity with its defined text where it is used. The principal element (ELEMENT) declared in DTD is **persons** and has zero or more elements **person** nested inside. The special characters inside the element definitions are "*" meaning: zero or more, "|" meaning: selection – either left side or right side, "+" meaning: one or more.

The attribute (ATTLIST) list defines two attributes for the **person** element: *project* and *job*.

The *project* attribute can contain character data (CDATA) and is optional (#IMPLIED). The *job* attribute can only have one of the two values, either "programmer" or "manager".

There is another XML document structure standard, called XML-Schema (W3C [167]), which is similar to DTD but is encoded in XML. This standard reconstructs all the capabilities of the DTD and extends them with: namespaces, context sensitivity, the possibility to define several root elements in the same schema, integrity constraints, regular expressions, sub-typing, etc. Tools for transforming XML-Schema representations from/to a DTD representation are available. We use the DTD variant in this example only because it is clearer than the too verbose XML-Schema.

One can consult the World Wide Web Consortium website (W3C [158], W3C [167]) for more information regarding XML, DTD and XML-Schema.

2.7 System Modeling Language (SysML)

The Unified Modeling Language (UML) has been created to assist software development processes by providing means to capture software system structure and behavior. This eventually evolved into the main standard for Model Driven Development.

The System Modeling Language (SysML) (OMG [114]) is a graphical modeling language for systems engineering applications. SysML was developed and submitted by systems engineering experts, and adopted by the OMG in 2006. SysML is built on top of UML2.0 and tailored to the needs of system engineers by supporting specification, analysis, design, verification and validation of a broad range of systems and system-of-systems.

The main goal behind SysML is to unify and replace different document-centric approaches in the system engineering field with a single systems modeling language. A single model-centric approach improves communication, assists to manage complex system design and allows its early validation and verification.

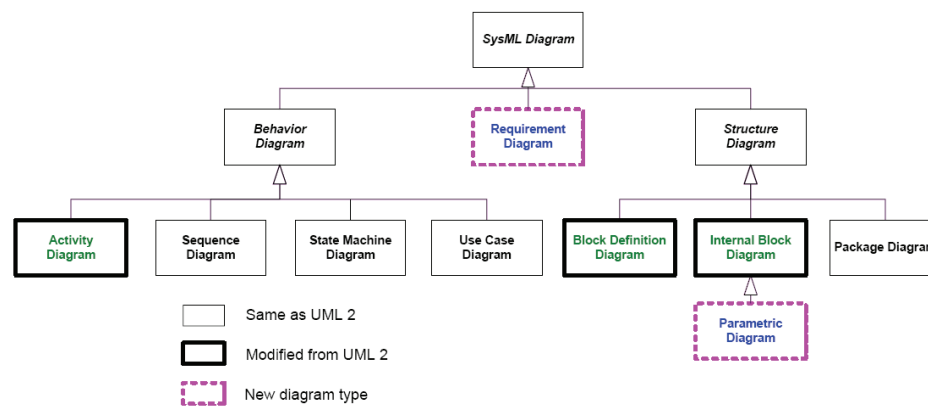


Figure 2-13. SysML diagram taxonomy.

The taxonomy of SysML diagrams is presented in Figure 2-13. The following major extensions compared to UML are made in SysML:

- *Requirements diagrams* support requirements presentation in tabular or in graphical notation, allows composition of requirements and supports traceability, verification and “fulfillment of requirements”. This is a new type of a diagram added to capture system requirements.
- *Block diagrams* extend the Composite Structure diagram of UML2.0. The purpose of this diagram is to capture system components, their parts and connections between parts. Connections are handled by means of connecting ports which may contain data, material, or energy flows.

- *Parametric diagrams* help perform engineering analysis such as performance analysis. Parametric diagrams contain constraint elements, which define mathematical equations, linked to properties of model elements.
- *Activity diagrams* show system behavior as data and control flows. Activity diagrams are similar to Extended Functional Flow Block Diagrams (EFFBDs), which are already widely used by system engineers. Activity decomposition is supported by SysML.
- *Allocations* are used to define mappings between model elements: For example, a certain Activity may be allocated to a Block, which implies that activity will be performed by the block.

For a full description of SysML see (SysML, 2006) (OMG [114]).

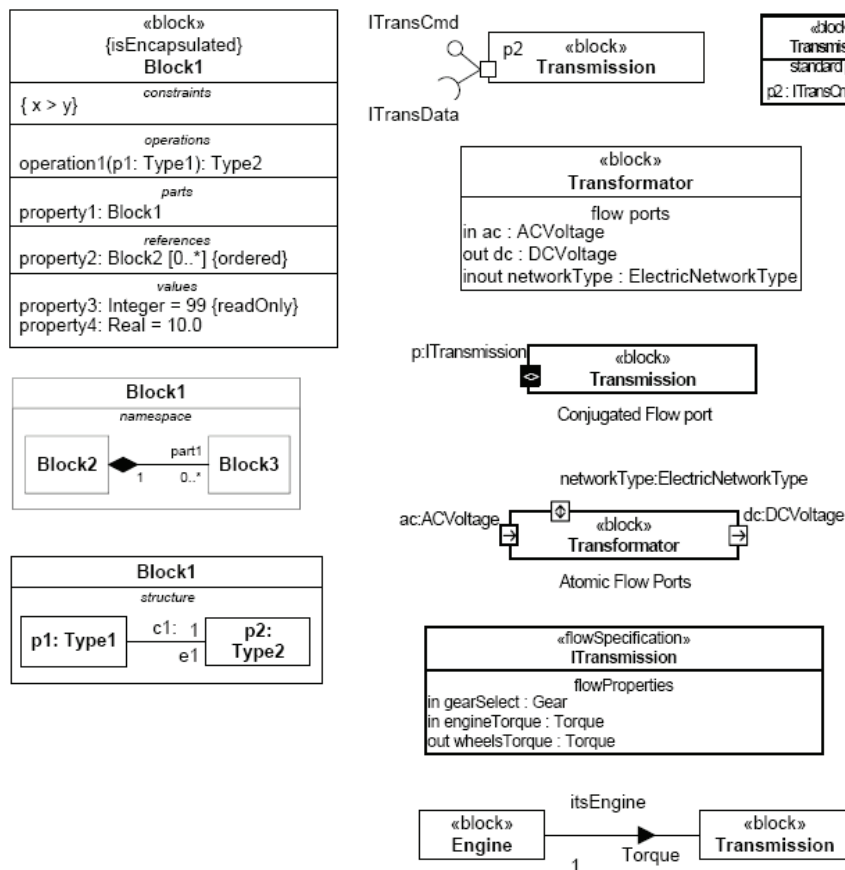


Figure 2-14. SysML block definitions.

2.7.1 SysML Block Definitions

SysML block definitions are shown in Figure 2-14. A SysML block can include properties to specify block parts, values, and references to other blocks. A separate compartment is dedicated for each of these features. To describe the behavior of a block the “Operations” compartment is reused from UML and it lists operations that describe certain behavior. SysML defines a special form of compartment for constraint definitions owned by a block. The use of the “Constraint” compartment is optional. A “Namespace” compartment may appear if nested block definitions exist for a block. A “Structure” compartment may appear to show internal parts and connections between parts within a block definition.

SysML defines two types of ports: standard ports and flow ports. Standard ports, which are reused from UML, are service-oriented ports required or provided by a block. Flow ports specify interaction points through which items may flow between blocks, and between blocks and environment. A flow port definition may include single item specification or complex flow specification through the FlowSpecification interface; flow ports define what “can” flow between the block and its environment. Flow direction can be specified for a flow port in SysML. SysML also defines a notion of Item flows that specify “what” does flow in a particular usage context.

2.8 Component Models for Invasive Software Composition

The idea that software should be built from existing components appeared in the software community at the end of the 60s, first formulated by Douglas McIlroy (McIlroy 1968 [96]) and had considerable influence in the software industry.

The most important result of dividing software into relatively independent and adaptable parts is the increased reusability in software development. "Reuse is the use of existing software components in a new context, either elsewhere in the same system or in another system" (Marciniak 1994 [90]). Programmers want a methodology that defines how to reintegrate previously created software into a new context of development, to create software systems from existing software rather than building them from scratch.

Software components are the basic units for software composition. They are designed to be composed; that is, their structure and behavior shall follow specific rules. "A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard." (Heineman and Councill 2001 [64]).

A component model defines the external appearance of components that build a system. The component model defines the functionality of the components to be used in composition by explicitly describing component interfaces. A well-designed

component model provides support for several important properties of its components, such as:

1. *Substitution*: one component can be replaced by another that fulfills at least the same syntactic or semantic conditions.
2. *Adaptation*: the ability to customize and configure components for different reuse contexts.
3. *Extension*: when new system requirements appear, the extension of existing components should be possible.

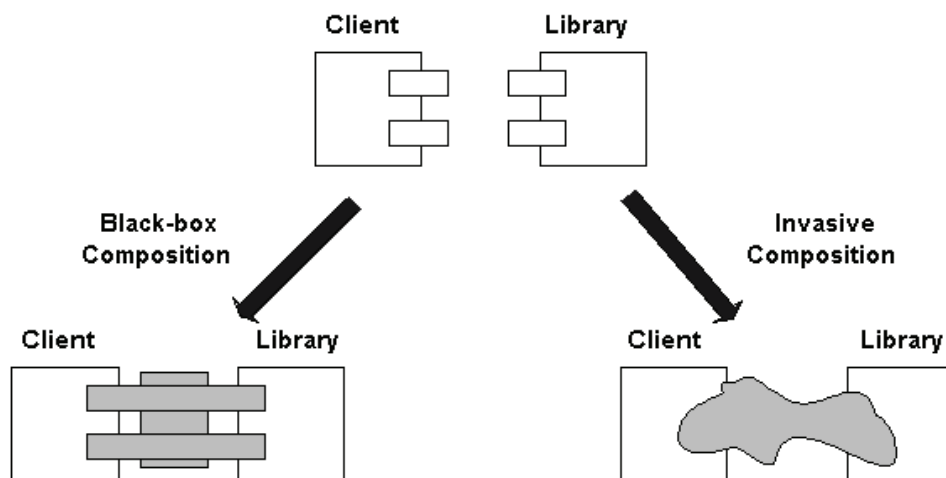


Figure 2-15. Black-box vs. Gray-box (invasive) composition. Instead of generating glue code, composers invasively change the components.

A component model is the core of a component system. In a typical component system, the component model describes components as *black boxes*, i.e., encapsulated binary code components with completely hidden implementations. The black-box composition method includes various transformations, like adaptation and glue code generation, which essentially compose black boxes without changing their actual content.

However, in Chapter 13 of this thesis we consider components containing *fragments*, i.e., pieces of code. As in black-box systems, the contents of the components are hidden under a composition interface. This method is different from black-box composition because the composition operators can invasively change the component fragments at predefined points of variability. This reuse abstraction is called *grey-box* composition and the composition of grey-box components is denoted as *invasive software composition* (see Figure 2-15).

Invasive software composition is a composition technology based on parameterization and extension of *grey-box* components (Aßmann 2003 [5]). For a

terminological distinction, we call invasive components *fragment boxes*; the variability points *hooks*, and the invasive composition operators *composers*. A typical fragment box consists of a set of fragments and an invasive composition interface, defined by hooks. Hooks can be of two types: *declared hooks*, defined by the programmer using some kind of markup and *implicit hooks* defined by the language structure.

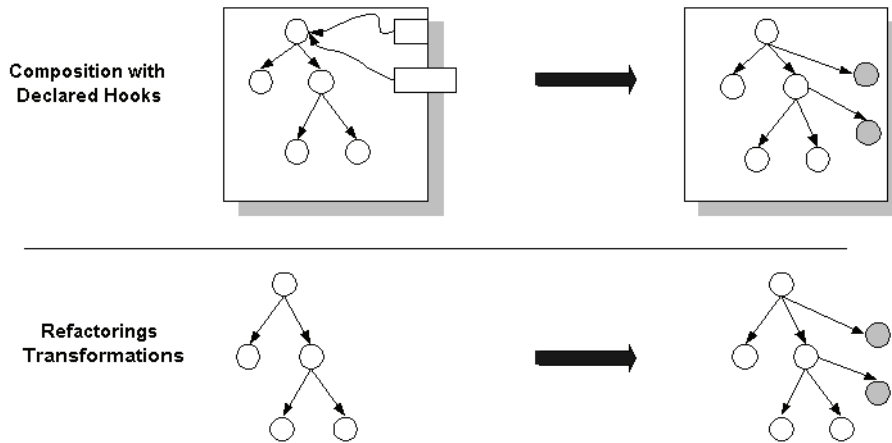


Figure 2-16. Invasive composition applied to hooks result in transformation of the underlying abstract syntax tree.

Since the composers of an invasive composition program manipulate fragment components, i.e., some other programs, an invasive composition implies meta-programming. The changes resulting from composition on fragment boxes apply directly to the corresponding abstract syntax tree by attaching and removing fragments as presented in Figure 2-16.

The COMPOST system (Aßmann and Ludwig 2005 [7]) provides invasive software composition of Java (Aßmann 2003 [5]) and ModelicaXML components (Chapter 12), (Pop and Fritzson 2003 [126]). The composition library supports generics (Musser and Stepanov 1988 [104]), mixin-ins (Bracha and Cook 1990 [17]), connectors (Aßmann et al. 2000 [6]), aspects (Kiczales et al. 1997 [78]) and views (Aßmann 2003 [5]) by invasively transforming language components.

Automatic derivation of a component model from language specification in Natural Semantics is presented in (Savga et al. 2004 [144]).

Using the Extensible Markup Language (XML) (W3C [158]), and the XML Schema (W3C [167]) to model abstract syntax trees (Attali et al. 2001 [8], Attali et al. 2001 [9], Badros 2000 [11], Schonger et al. 2002 [145]) of programming languages is becoming an interesting alternative for having easy access to the structure of programs (in our case models) without the need for a specific parser. We used this approach when designing and defining the meta-model for the Modelica language presented in this thesis. In order to compose and transform models defined by our meta-model we employ invasive software composition

(Aßmann 2003 [5]), which is a grey-box component composition. To drive the composition we have designed a component model for our meta-model within the COMPOST system.

2.9 Integrated Product Design and Development

In the area of model-driven product design using modeling and simulation we focus on the integration of the Modelica language with conceptual modeling tools based on Function-Means tree decomposition (Andreasen 1980 [3]).

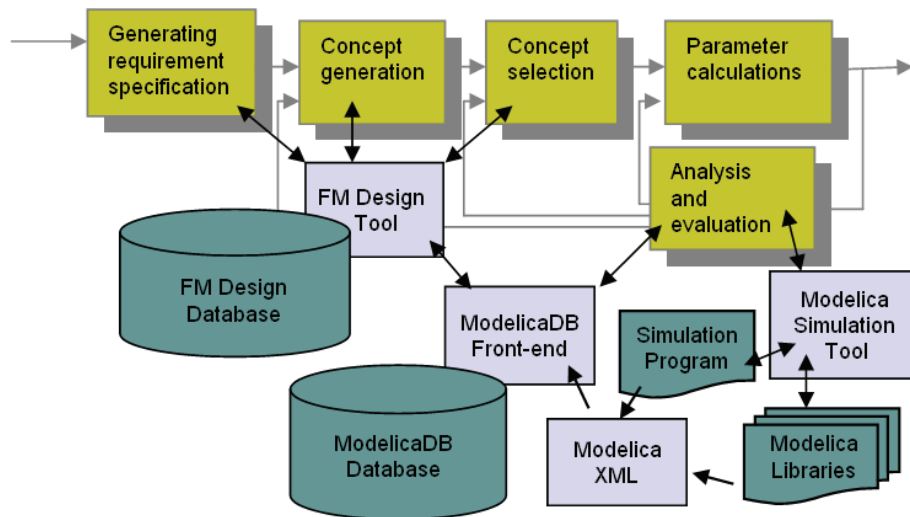


Figure 2-17. Integrated model-driven product design and development framework.

Designing products is a complex process. Highly integrated tools are essential to helping a designer to work efficiently. Designing a product includes early design phase product concept modeling and evaluation, physical modeling and simulation and finally the physical product realization (Figure 2-17). For physical modeling and simulation available tools provide advanced functionality. However, the integration of such tools with conceptual modeling tools is a resource consuming process that today requires large amounts of manual, and error prone work. Also, the number of physical models available to the designer in the product concept design phase is typically quite large. This has an impact on the selection of the best set of component choices for detailed product concept simulation.

To address these issues we have developed a framework (Chapter 11) for product development based on an XML meta-model (Chapter 12), (Pop and Fritzson 2003 [126]) of Modelica and its representation in a Modelica Database (Johansson et al. 2005 [74], Pop et al. 2004 [132]). The product concept design of

the product development process is based on Function-Means tree decomposition and is implemented in the FMDesign component (Figure 2-17).

To provide flexibility of the product design framework we have addressed the composition and transformation of Modelica models in the COMPOST framework (Chapter 13), (Pop et al. 2004 [133]).

Our framework for model-driven product design and development has similarities with Schemebuilder (Bracewell and D.A.Bradley 1993 [16]). The Modelith framework (Johansson et al. 2002 [72], Larsson et al. 2002 [81]) also employs an XML-based model representation for transformation and exchange in physical system modeling.

However, our work is more oriented towards the design of advanced complex products that require systems engineering, and targeted to the simulation modeling language Modelica. The Modelica language has a more expressive power in modeling dynamic systems and system architectures, than many of the tools for systems engineering that are currently used. Also, meta-modeling and invasive software composition methods are considered for automatic model composition and configuration. Tight integration of conceptual modeling tools with modeling and simulation tools is proposed. For details on Systems Engineering, the reader is referred to the International Council on Systems Engineering Website (INCOSSE 1990-2008 [70]).

Part II

Extending EOO Languages for Safe Symbolic Processing

Chapter 3

Extending Equation-Based Object-Oriented Languages

For a long time, one of the major research goals in the computer science research community has been to raise the level of abstraction and expressive power of specification languages/programming languages. Many specification languages and formalisms have been invented, but unfortunately very few of those are practically useful, due to limited computer support for these languages and/or inefficient implementations. Thus, one important goal is executable specification languages of high abstraction power and with high performance, good enough for practical usage and comparable in execution speed to hand implementations of applications in low-level languages such as C or C++. In the background chapter we described our work in creating efficient executable specification languages for two application domains. The first area is formal specification of programming language semantics, whereas the second is formal specification of complex systems for which an object-oriented mathematical modeling language called Modelica was developed, including architectural support for components and connectors. Based on these efforts, we designed a unified equation-based mathematical modeling language that can handle modeling of items as diverse as programming languages, computer algebra, event-driven systems, and continuous-time physical systems. The key unifying feature is the notion of equation. In this chapter we describe the design and implementation of the unified language. A prototype compiler implementation is already up and running, and used for substantial applications.

3.1 Introduction

About sixteen years ago, our research group has selected two application domains for research on high-level specification languages:

- Specification languages for programming language semantics. Much work has been done in this area, but there is still no standard class of compiler-

compiler tools around, as successful as parser generators based on grammars in BNF form like lex (flex), yacc (bison), ANTLR, etc.

- Equation-based specification languages for mathematical modeling of complex (physical) systems.

The purpose of our work is to unify the languages developed in these two domains into a new language. The main goal of this work is the design and development of a *general executable mathematical modeling and semantics meta-modeling language*. This language should have a clean semantics as in the case of Modelica and Natural Semantics (RML), and should be compiled to code of high performance. This language will allow expressing mathematical models but also meta-models and meta-programs that specify composition of models, transformation of models, model constraints, etc. This language is based on Modelica extended with several new language constructs that allow program language specification. The unified language is called MetaModelica. MetaModelica – a Unified Equation-Based Modeling Language

The idea to define a unified equation-based mathematical and semantical modeling language started from the development of the OpenModelica compiler (Fritzson et al. 2002 [46]). The entire compiler was generated from a Natural Semantics specification written in RML. The open source OpenModelica compiler has its users in the Modelica community which have detailed knowledge of Modelica but very little knowledge of RML and Natural Semantics. In order to allow people from the Modelica community to contribute to the OpenModelica compiler we retargeted the development language from RML to MetaModelica, which is based on the Modelica language with several extensions. We already translated the OpenModelica compiler from RML to the MetaModelica using an automated translator (Carlsson 2005 [21]) implemented in RML. We also developed a compiler which can handle the entire OpenModelica compiler specification (~140000 lines of code) defined in MetaModelica. An evaluation of the performance of the compiler and the generated code is presented in Chapter 4.

The basic idea behind the unified language is to use equations as the unifying feature. Most declarative formalisms, including functional languages, support some kind of limited equations even though people often do not regard these as equations, e.g. single-assignment equations.

Using the meta-programming facilities, common tasks like generation, composition, and querying of Modelica models can be automated.

The MetaModelica language inherits all the strong component capabilities of Modelica. Components can be reused in different contexts because the causality is not fixed in equations and is up to the compiler to decide it.

3.1.1 Evaluator for the Exp1 Language in the Unified Language

Below we give a very simple example of the meta-modeling and meta-programming capabilities of the MetaModelica language. The semantics of the

operations in the small expression language `Exp1` follows below, expressed as an interpretative language specification in `MetaModelica` in a style close to `Natural` and/or `Operational Semantics`, see `Exp1` specified in `RML` in Section 2.5.1. Such specifications typically consist of a number of functions, each of which contains a match expression with one or more cases, also called rules. In this simple example there is only one function, here called `eval`, since we specify expression evaluation.

```

function eval
  input   Exp  in_exp;
  output Real out_real;
algorithm
  out_real :=
  match in_exp
  local Real v1,v2,v3;  Exp e1,e2;
  case RCONST(v1) then v1;
  case ADD(e1,e2) equation
    v1 = eval(e1);  v2 = eval(e2); v3 = v1 + v2; then v3;
  case SUB(e1,e2) equation
    v1 = eval(e1);  v2 = eval(e2); v3 = v1 - v2; then v3;
  case MUL(e1,e2) equation
    v1 = eval(e1);  v2 = eval(e2); v3 = v1 * v2; then v3;
  case DIV(e1,e2) equation
    v1 = eval(e1);  v2 = eval(e2); v3 = v1 / v2; then v3;
  case NEG(e1) equation
    v1 = eval(e1); v2 = -v1; then v2;
  end match;
end eval;

```

As usual in `Modelica` the equations are not directional, e.g. the two equations `v1 = eval(e1)` and `eval(e1) = v1` are equivalent. The compiler will select one of the forms based on input/output parameters and data dependencies.

There are some design considerations behind the above match-expression construct that may need some motivation.

- Why do we have local variable declarations within the match-expression? The main reason is clear and understandable semantics. In all three usage contexts (equations, statements, expressions) it should be easy to understand for the user and for the compiler which variables are unknowns (i.e., unbound local variables) in pattern expressions or in local equations. Another reason for declaring the types of local variables is better documentation of the code – the modeler/programmer is relieved of the burden of doing manual type-inference to understand the code.
- Why the `then` keyword before the returned value? The code becomes easier to read if there is a keyword before the returned value-expression. Note that most functional languages use the `in` keyword instead in this context, which

is less intuitive, and would conflict with the array element membership meaning of the Modelica in keyword.

3.1.2 Examples of Pattern Matching

A pattern matching construct is useful not only 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 := match (inExp)
    case (0) then 1;
    case (n) then
      if n>0
      then n*fac(n-1)
      else fail();
    end match;
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) := match (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(ifExpel, 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 match;
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 (i.e., variable or constant), 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. Lists do not exist in Modelica but are an important construct in MetaModelica. The following function selects an element that fulfills a certain condition from a list. The `matchcontinue` construct is used in this case instead of `match`. The `matchcontinue` construct uses local backtracking to select the correct case:

- The first case that matches the given value is selected and evaluated (marked case (a) in the example);
- If during the execution of case (a) the equation `true = cond(x);` fails because `cond(x)` returns `false` all the variables bound previously become un-bound and the next case marked case (b) is selected for execution.
- If case (b) fails too then the entire `listSelect` function fails.

```

function listSelect
  input list<Type_a>          inTypeALst;
  input Func_anyTypeToBool inFunc;
  output list<Type_a>        outTypeALst;
public
  replaceable type Type_a constrainedby 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) // case (a)
        equation
            true = cond(x);
            xs_1 = listSelect(xs, cond);
        then (x :: xs_1);
    case ((x :: xs),cond) // case (b)
        equation
            false = cond(x);
            xs_1 = listSelect(xs, cond);
        then xs_1;
    end matchcontinue;
end listSelect;

```

The symbol `::` is just 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 constrainedby 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 is walking over class hierarchies. Modelica is an object-oriented language and one can use pattern matching to explore a hierarchy of classes as presented in (Emir et al. 2007 [36]). 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. A discussion about these difficulties is given in the pattern-matching section of this chapter (section 3.5).

3.1.3 Language Design

In the next sections we present the MetaModelica language design. The equations types of the unified language are presented together with pattern-matching features and exception handling.

3.2 Equations

The following sections presents the kinds of equations already present in Modelica and detail the addition of the equations that support the definition of semantic specifications.

3.2.1 Mathematical Equations

Mathematical models almost always contain equations. There are basically four main kinds of mathematical equations in Modelica which we exemplify below expressed in traditional mathematical syntax.

Differential equations contain time derivatives such as $\frac{dx}{dt}$, usually denoted \dot{x} :

$$\dot{x} = a \cdot x + 3 \quad (1)$$

Algebraic equations do not include any differentiated variables:

$$x^2 + y^2 = L^2 \quad (2)$$

Partial differential equations also contain derivatives with respect to other variables than time:

$$\frac{\partial a}{\partial t} = \frac{\partial^2 a}{\partial z^2} \quad (3)$$

Difference equations express relations between variables, e.g. at different points in time:

$$x(t+1) = 3x(t) + 2 \quad (4)$$

3.2.2 Conditional Equations and Events

Behavior can develop continuously over time or as discrete changes at certain points in time, usually called events. It is possible to express events and discrete behavior solely based on conditional equations. An event in Modelica is something that happens that has the following four properties:

- A point in time that is instantaneous, i.e., has zero duration.
- An event condition that switches from false to true for the event to happen.
- A set of variables that are associated with the event, i.e., are referenced or explicitly changed by equations associated with the event.
- Some behavior associated with the event, expressed as conditional equations that become active or are deactivated at the event. Instantaneous equations are a special case of conditional equations that are active only at events.

Modelica has several constructs to express conditional equations, e.g. if-then-else equations for conditional equations that are active during certain time durations, or when-equations for instantaneous equations.

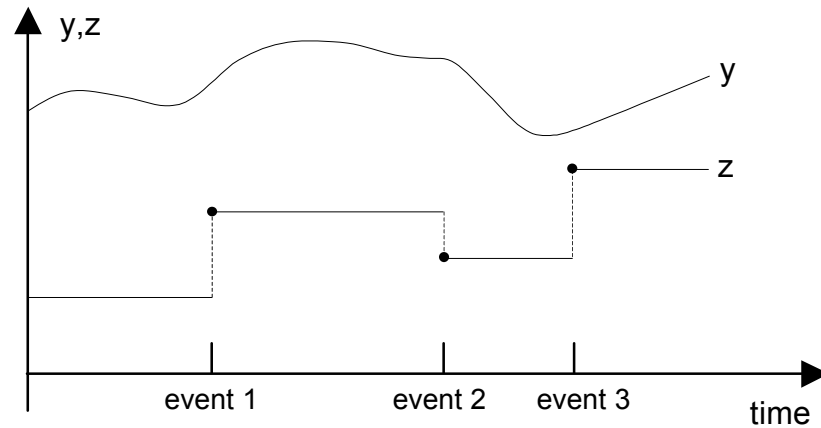


Figure 3-1. A discrete-time variable z changes value only at event instants, whereas continuous-time variables like y may change both between and at events.

3.2.3 Single-Assignment Equations

A single-assignment equation is quite close to an assignment, e.g.:

```
x = eval_expr(env, e);
```

but with the difference that the unbound variable (here x) which obtains a value by solving the equation, only gets its value once, whereas a variable in an assignment may obtain its value several times, e.g.:

```
x := eval_expr(env, e);
x := eval_expr2(env, x);
```

3.2.4 Pattern Equations in Match Expressions

In this section we present our addition to the Modelica language which allows definitions of semantic specifications. The new language features are pattern equations, match expressions and union datatypes.

Pattern equations are a more general case than single-assignment equations, e.g.:

```
Env.BOOLVAL(x, y) = eval_something(env, e);
```

Unbound variables get their values by using pattern-matching (i.e., unification) to solve for the unbound variables in the pattern equation. For example, x and e might be unbound and solved for in the equations, whereas y and env could be bound and just supply values.

The following extension to Modelica is essential for specifying semantics of language constructs represented as abstract syntax trees:

- Match expressions with pattern-matching case rules
- Local declarations
- Local equations.

It has the following general structure:

```
match expression optional-local-declarations

  case pattern-expression opt-local-declarations
    optional-local-equations then value-expression;
  ...
  else optional-local-declarations
    optional-local-equations then value-expression;

end match;
```

The `then` keyword precedes the value to be returned in each branch. The local declarations started by the `local` keyword, as well as the equations started by the `equation` keyword are optional. There should be at least one `case...then` branch, but the `else`-branch is optional.

A `match` expression is closely related to pattern matching in functional languages, but is also related to switch statements in C or Java. It has two important advantages over traditional switch statements:

- 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 in simple cases, but is unification in the general case.

Local equations in match expressions have the following properties:

- Only algebraic equations are allowed as local equations, no differential equations.
- Only locally declared variables (local unknowns) declared by local declarations within the case expression are solved for, or may appear as pattern variables.
- Equations are solved in the order they are declared (this restriction may be removed in the future, allowing more general local algebraic systems of equations).
- If an equation or an expression in a case-branch of a `match`-expression fails, all local variables become unbound, and matching continues with the next branch.

3.3 High-level Data Structures

To support simple meta-modeling features the MetaModelica extends the Modelica language with new constructs which we present in the following.

3.3.1 Union-types

To facilitate meta-modeling of abstract syntax trees we also need to introduce the possibility to declare recursive tree data structures in Modelica, e.g.:

```
uniontype Exp
  record RCONST Real x1; end RCONST;
  record PLUS   Exp x1; Exp x2; end PLUS;
  record SUB    Exp x1; Exp x2; end SUB;
  record MUL    Exp x1; Exp x2; end MUL;
  record DIV    Exp x1; Exp x2; end DIV;
  record NEG    Exp x1;          end NEG;
end Exp;
```

A small expression tree, of the expression $12+5*13$, is depicted in Figure 3-2. Using the record constructors PLUS, MUL, RCONST, this tree can be constructed by the expression `PLUS (RCONST(12), MUL (RCONST(5), RCONST(13)))`

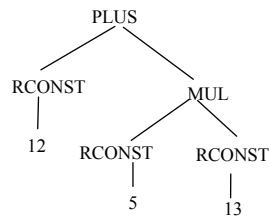


Figure 3-2. Abstract syntax tree of the expression $12+5*13$

The `uniontype` construct has the following properties:

- Union types can be recursive, i.e., reference themselves. This is the case in the above Exp example, where Exp is referenced inside its member record types.
- Record declarations declared within a union type are automatically inherited into the enclosing scope of the union type declaration.
- Union types can be polymorphic
- A record type may currently only belong to one union type. This restriction may be removed in the future, by introducing polymorphic variants.

This is a preliminary union type design, which however is very close (just different syntax) to similar datatype constructs in declarative languages such as Haskell,

Standard ML, OCaml, and RML. The union types can model any abstract syntax tree while the match expressions are used to model the semantics, composition or transformation of the specified language.

3.3.2 Lists, Tuples and Option Types

Besides union-types, the MetaModelica language extends Modelica with new high-level types that improve the meta-modeling capability of the language. All these constructs can be type-parameterized.

3.3.2.1 Lists

Lists are very useful data structures that are highly used in imperative or functional programming. The syntax of a list is a comma-separated list of values or variables of the same type, e.g. `{..., ...}`. The following is a list of integers, using the list data constructor:

```
{1, 2, 3, 4}
```

The declaration of list variables uses a Java like syntax. For example a variable with its value described by the list above has the following declaration:

```
list<Integer> varName;
```

In MetaModelica the list constructor `{..., ...}` is overloaded because the Modelica language already contains the same syntax for array construction. The MetaModelica compiler prototype deduces from the context and the variable declarations if the constructor refers to an array or a list. In the pattern matching context, the list constructor is used for the list decomposition.

List can also be constructed/deconstructed using the cons operator `::` that constructs/deconstructs a list from its head and its tail (the rest of the list):

```
1:: {2, 3, 4} = {1, 2, 3, 4};
```

The `nil` keyword can be used to specify an empty list and is equivalent to `{}`.

3.3.2.2 Tuples

Tuples are like records, but without field names. They can be used directly, without previous declaration of a corresponding tuple type. The syntax of a tuple is a comma-separated list of values or variables, e.g. `(..., ..., ...)`. The following is a tuple of a real value and a string value, using the tuple data constructor:

```
(3.14, "this is a string")
```

The declaration of tuple variables uses a Java like syntax. For example a variable with its value described by the tuple above has the following declaration:

```
tuple<Real, String> varName;
```

Tuples already existed in a limited way in previous versions of Modelica since functions with multiple results are called using a tuple for receiving results, e.g.:

```
(a,b,c) := foo(x, 2, 3, 5);
```

In the pattern matching context the syntax that constructs the tuple, `reverses` its semantics and it used to access the values of its elements.

3.3.2.3 Options

Option types are used to model optional constructs. The option types are similar to C/C++ or Java null values. The values of a variable of this type can be `NONE` or `SOME(value)`:

```
SOME(3.14);
NONE();
```

The declaration of option variables uses a Java like syntax. For example a variable with its value described by the option above has the following declaration:

```
Option<Real> varName;
```

The `SOME(...)` and `NONE()` constructors are also used for decomposing option values in the pattern matching context. An option type can also be viewed as a union type consisting of two records `SOME` (with one field) and `NONE` (with no fields).

3.4 Solution of Equations

The process of solving systems of equations is central for the execution of equation-based languages. For example:

- Differential equations are solved by numeric differential equation solvers.
- Differential-algebraic equations are solved by numeric DAE solvers.
- Algebraic equations are solved by symbolic manipulation and/or numeric solution
- Single-assignment equations are solved by performing an assignment.
- Pattern equations are solved by the process of unification which assigns values to unbound variables in the patterns.

The first three solution procedures are used in current Modelica. By the addition of local equations in match expressions to be solved at run-time, we generalize the allowable kinds of equations in Modelica.

3.5 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. This section is partially based on (Stavåker et al. 2008 [149]).

Pattern matching is a well-known, powerful language feature found in functional programming languages. In this section we present the design 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.

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 $p1, \dots, pN$ 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 (Hudak 2000 [68]), OCaml (Leroy et al. 2007 [84]), and Standard ML (Milner et al. 1997 [97]). However, pattern matching is currently missing from state-of-the-art object-oriented equation-based (EOO) languages. Pattern matching features are also rare in imperative object-oriented languages even though some research has been carried out (Liu and Myers 2003 [87], Moreau et al. 2003 [102], Odersky and Wadler 1997 [110], Zenger and Odersky 2001 [176]). In (Liu and Myers 2003 [87]), for instance, the JMatch language which extends Java with pattern matching is described.

The language described in (Emir et al. 2007 [36]) promotes the use of pattern matching constructs in object-oriented languages as a means of exploring class hierarchies. One could for instance apply the visitor pattern to solve the same problem but as (Odersky 2006 [109]) notes this requires a lot of code scaffolding and nested patterns are not supported.

The pattern matching construct for Modelica was first presented in a paper on Modelica meta-programming extensions (Pop and Fritzson 2006 [130]).

3.5.1 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. When the `matchcontinue` keyword is used a failure within the case statement execution will continue with the execution of the next case that matches the same pattern. When the `match` keyword is used, a failure in any of the cases will trigger the failure of the entire function. 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).

3.5.2 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 sub-patterns will be matched (recursively) against parts of `v`.

3.5.3 Discussion on Type Systems

Modelica features a structural type system (Modelica.Association 2007 [100]). Another class of type systems is nominal 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 (Pierce 2002 [124]).

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 a 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.

3.6 Exception Handling

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 section we present the design of exception handling for Modelica. The next chapter presents the implementation of exception handling. To our knowledge this is the first approach of integrating equation-based object-oriented languages (EOO) with exception handling.

According to the terminology defined in IEEE Standard 100 (IEEE 2000 [69]), 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 (Bunus 2004 [19]) and (Broman 2007 [18]). However, here 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 (Gosling et al. 2005 [62]), C++ (Stroustrup 2000 [150]), and functional languages such as Haskell (Hudak 2000 [68]), OCaml (Leroy et al. 2007 [84]), and Standard ML (Milner et al. 1997 [97]). However, exception handling is currently missing from object-oriented equation-based (EOO) languages.

A short sketch of the syntax of exception handling for Modelica was presented in a paper on Modelica meta-programming extensions (Fritzson et al. 2005 [51]), 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 (Pop et al. 2008 [134]). 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.

3.6.1 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 if;
end log;

```

The function `log` 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 the `Modelica.Utilities` package) that deal with errors due to the operating system. A standard hierarchy of exceptions in common for all tools could be defined in the Modelica Standard Library for all the exception categories needed. Depending on the simulation runtime implementation (i.e., language of choice) of the Modelica tool implementation, 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 the `Test` model examples of 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:

- Compile time for expressions or functions that are evaluated at compile time, e.g. as part of parameter expressions.
- Simulation time, due to exceptions thrown within the solver, functions, expressions, or equations.

All the exceptions thrown 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.

3.6.2 Exception Handling Syntax and Semantics

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* (Modelica.Association 2007 [100]) 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_expression }
        [ 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 formal parameter can only appear inside the catch clause and will throw the currently caught exception. This grammatical constraint is not specified in the above grammar to keep it simple, since it can instead be checked by the semantics phase.

The try-catch clauses shown here belong to the various contexts rules in the Modelica grammar: expressions, algorithm sections, and equation sections.

3.6.2.1 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.

3.6.2.2 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.

3.6.2.3 Exception Handling for Equation Sections

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 different semantics for try-catch in equation sections. Some of them are discussed in Section 3.6.5.

3.6.2.4 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 at compile time automatically find out 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` (see the Modelica grammar for the element list specification) of the function or model:

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

The possible exceptions to be thrown are not really needed to specify using a special language construct, we could use annotations instead:

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

Names used above are defined according to the exception `name` grammar rule specified at the beginning of this section.

In the literature this feature of the compiler (or the language) is called *Checked exceptions* (Roy and Haridi 2004 [138]).

3.6.3 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.

3.6.3.1 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.

3.6.3.2 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`.

3.6.3.3 Exceptions as 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 exception values and their types 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.

3.6.4 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 the 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 an 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.

3.6.5 Further Discussion

During the design and implementation of exception handling we have encountered various issues which we present in this section. The exception handling in expressions and algorithm sections is 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 is 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 an example where alternative equations for simulation might be needed we can think of the same system at a different level of detail. Where the detailed model can fail due to

complexity and numerical problem the simulation can be continued with the less complex model.

3.6.5.1 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 section 3.6.1)
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 believe that the most useful design for exception handling in equation sections is the one that has both features 1 and 2 active.

3.7 Related Work

We are not aware of any other existing EOO language that contains general purpose meta-modeling and meta-programming facilities.

With regards to the meta-modeling facilities present in the MetaModelica language we can consider as related work the Unified Modeling Language (UML). Modeling in the UML sense has more emphasis on graphical notation for modeling rather than precise mathematical model definitions as in the modeling languages mentioned in the previous sections. Initially, execution support was lacking, but during recent years code generators from executable subsets of UML2 have appeared. Also, during recent years, there has been an increased interest in model-driven developments and the OMG has launched the model-driven architecture, primarily based on UML models.

The idea of meta-modeling has attracted increased interest: a meta-model describes the structure of models at the next lower abstraction level. Meta-modeling and meta-programming allows transformations and composition of models and programs, which is becoming increasingly relevant in order to specify and manage complex industrial software and system applications.

However, UML has developed into a rather heterogeneous collection of modeling notations. Also, precise mathematically defined semantics is not always available for these graphical notations. By contrast, MetaModelica is defined

exclusively based on equations, functions, and meta-functions. Similar meta-programming facilities are present in functional languages like SML, Haskell and OCaml but the execution strategy is different in these languages as they do not support backtracking to select cases.

Further related work is presented in the next chapter where we give performance evaluation of the MetaModelica compiler prototype implementation.

3.8 Conclusions and Future Work

We have presented the integration of two executable specification languages: RML for Natural Semantics specifications of programming languages, and Modelica for equation-based semantics and mathematical modeling of complex systems. The language resulted from the integration is called MetaModelica – a unified mathematical and semantical modeling language generalizing the concept of equation and introducing local equations, match expressions and exception handling in the Modelica language. This gives interesting perspectives for the future regarding safe meta-modeling, model transformations, and compositions during simulation, etc.

The OpenModelica compiler has been ported to the new unified Modelica modeling language, resulting in ~140000 lines of code expressed in the unified language. A compiler for MetaModelica has been completed and its implementation and evaluation are the focus of the next chapter. We have also developed an integrated development environment (see Chapter 8) based on Eclipse which facilitates the development and debugging of MetaModelica models (PELAB 2006-2008 [119]). The MetaModelica language can be used to write semantic specifications for a broad spectrum of languages ranging from functional to imperative languages. We have also translated all our RML specification examples to MetaModelica in order to provide teaching material for the new language. The current specifications include imperative, functional, equation-based, and object-oriented languages.

The unified MetaModelica language gives new perspectives for a broad range of items, from programming and modeling languages to physical systems, but also including model transformations and composition. Apart from language specification to generate interpreters and compilers, symbolic differentiation rules for differentiating expressions and equations have been specified in MetaModelica and is in use.

We have also presented the design 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 exceptional conditions and errors. There are several issues that have to be considered when designing and implementing these constructs which we have discussed in this chapter.

The design of pattern matching for Modelica was also addressed. By adding this language feature to Modelica we provide 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 is the most useful for handling MetaModelica constructs such as lists, tuple, options and union types. The possibility to pattern match over record-hierarchies is also considered. 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 that would be error-prone to be written.

One medium term goal for the MetaModelica language is its implementation in the OpenModelica compiler, thus being bootstrapped in itself. A long-term vision is the visual development of compilers for any language by using drag and drop on semantic components from libraries which are then connected together in a similar way the physical systems are modeled today in Modelica.

Chapter 4

Efficient Implementation of Meta-Programming EOO Languages

4.1 Introduction

In this chapter we present the implementation details of the systems supporting the MetaModelica language. To quickly prototype a compiler for the MetaModelica language we extended the RML compiler to support the new syntax and some of the new semantics.

For full MetaModelica language support we are currently working on extending the OpenModelica compiler (that supports the Modelica language) with the missing meta-modeling, meta-programming and exception handling features. Our goal is to bootstrap the OpenModelica compiler, thereby making the MetaModelica compiler prototype obsolete.

4.2 MetaModelica Compiler Prototype

The first prototype compiler for the MetaModelica language is based on the RML compiler. The RML compiler was extended with a new parser for MetaModelica and an internal translation phase from MetaModelica to RML. Also, debugging facilities (see Chapter 5 and Chapter 7) were added and the garbage collection of the RML system was extended with mutable references. The prototype is heavily used in the development of the OpenModelica compiler for several years. Since we switched to the MetaModelica syntax, implemented the debugger and the interactive environment (MDT) various people from the Modelica community started to provide contributions to the OpenModelica compiler.

However, our final goal is to be able to bootstrap the OpenModelica compiler to be able to use the full Modelica language features together with the meta-programming extensions.

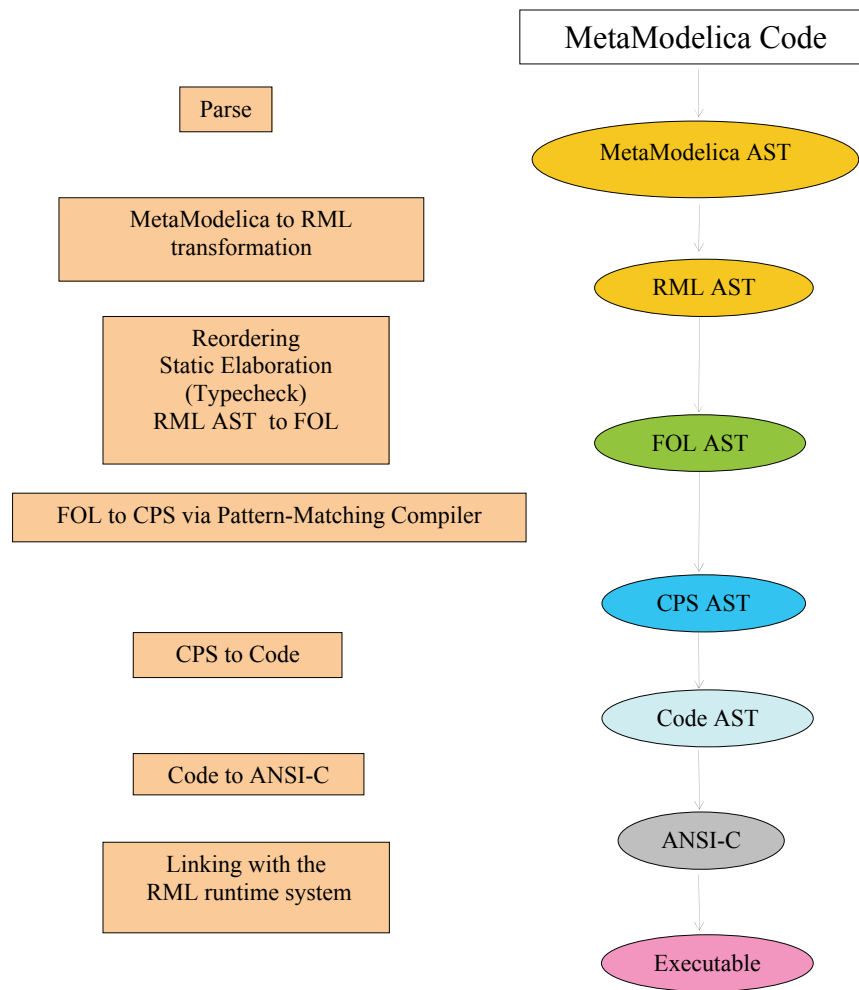


Figure 4-1. MetaModelica Compiler Prototype – compilation phases.

4.2.1 Performance Evaluation of the MetaModelica Compiler Prototype

We are not aware of any language that is very similar to the MetaModelica language. However, the meta-modeling and meta-programming parts of the MetaModelica language are close to logic/functional languages. Backtracking is

used within the match construct (matchcontinue) to select the correct case and the specifications can contain logical variables. The union types are similar to the SML datatype definitions, however MetaModelica functions have multiple inputs and outputs not just one argument like in SML. Also, because a reordering phase is applied to the MetaModelica code there is no need to explicitly declare mutually recursive types and functions.

All the information, the test code and the files needed to reproduce our results are available online at: <http://www.ida.liu.se/~adrpo/phd/tests>. Also you can contact the author for any information regarding the performance evaluation tests.

We have compared the execution speed of our generated code with SWI-Prolog 5.6.9 (SWI-Prolog [151]), SICStus Prolog 3.11.2 (SICS [147]), Maude MSOS Tool (MMT) on top of Maude 2.1.1 (Maude.Team [92]). The Maude MSOS Tool (MMT) is an execution environment for Modular Structural Operational Semantics (MSOS) (Mosses 2004 [103]) specifications that brings the power of analysis available in the Maude system to MSOS specifications. The Maude MSOS Mini-Freja translation was implemented by Fabricio Chalub and Christiano Braga and is available as a case study together with sources from <http://maude-msos-tool.sourceforge.net/>. SWI-Prolog is a widely known open source implementation of Prolog. SICStus Prolog is a commercial Prolog implementation.

The closest match to the meta-modeling and meta-programming facilities of the MetaModelica compiler prototype is the Maude MSOS Tool.

The test case is based on an executable specification of the Mini-Freja language (Pettersson 1999 [122]) running a test program based on the sieve of Eratosthenes. Mini-Freja is a call-by-name pure functional language. The test program calculates prime numbers. The Prolog translation (mf.pl) was implemented by Mikael Pettersson and this author corrected a minor mistake.

The comparison was performed on a Fedora Core4 Linux machine with two AMD Athlon(TM) XP 1800+ processors at 1500 MHz and 1.5GB of memory.

Table 4-1. Execution time in seconds. The – sign represents out of memory.

#	MetaModelica	SICStus	SWI	Maude MSOS Tool
8	0.00	0.05	0.00	2.92
10	0.00	0.10	0.03	5.60
30	0.02	1.42	1.79	226.77
40	0.06	3.48	3.879	–
50	0.13	–	11.339	–
100	1.25	–	–	–
200	16.32	–	–	–

The memory consumption was at peak 9Mb for MetaModelica and the others consumed the entire 1.5Gb of memory and aborted at around 40 prime numbers. With this test we stressed only the meta-programming and meta-modeling part of the compiler.

4.3 OpenModelica Bootstrapping

The MetaModelica compiler prototype cannot handle the entire Modelica language. To construct a compiler for the complete MetaModelica language we decided to extend the OpenModelica compiler (OMC) with the missing features: pattern matching, exception handling, union types, lists, etc. This way the OpenModelica compiler could be bootstrapped and the MetaModelica compiler would not be needed anymore.

This is also according to our long-term vision of the meta-programming and meta-modeling facilities in MetaModelica: to enable more modular and extensible tooling, as earlier discussed.

4.3.1 OpenModelica Compiler Overview

The OpenModelica compiler phases are presented in the following (see also Figure 4-2). The MetaModelica code is first parsed and then translated into a so-called “*flat model*”. This phase includes type checking, performing all object-oriented operations such as inheritance, modifications, compilation of pattern matching, translation of meta-functions to C code. The flat model includes a set of equations, declarations, functions, and meta-functions, with all object-oriented structure removed, apart from the dot notation within the names. This process is called the “*partial flattening*” of the model.

The next step is to solve the system of equations. First the equations need to be transformed into a suitable form for the numerical solvers. This is done by the symbolic and the numerical module of the compiler. The simulation code generator takes as input the flattened form of the equations. The equations are mapped into an internal data structure that permits simple symbolic manipulations such as: common subexpression elimination, algebraic simplifications, constant folding, etc. These symbolic operations substantially decrease the complexity of the system of equations. After this stage the Block Lower Triangular form of the system of equations is computed.

Finally, in the last phase, the procedural code (in our implementation C code), is generated based on the previously computed BLT blocks when each block is linked to a numerical solver and the runtime for the meta functions. Within the C code the meta functions are called like normal functions.

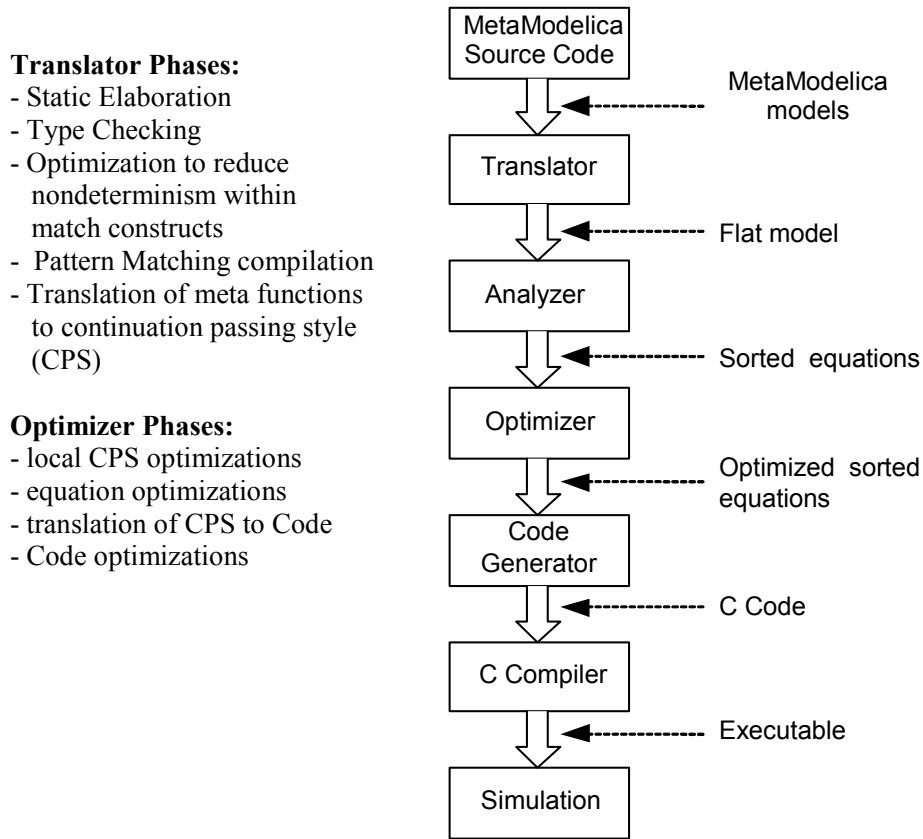


Figure 4-2. The stages of translation and execution of a MetaModelica model.

The detailed architecture of the OpenModelica compiler can be seen in Figure 4-3. One can see that there are three main kinds of packages:

- Function packages that perform a specified function, e.g. *Lookup*, code instantiation, etc.
- Data type packages that contain declarations of certain data types, e.g. *Absyn* that declares the abstract syntax.
- Utility packages that contain certain utility functions that can be called from any package, e.g. the *Util* package with general list processing functions.

The functionality classification is not clear cut, since certain packages perform several functions. For example, the *SCode* package primarily defines the lower-level *SCode* tree structure, but also transforms *Absyn* into *SCode*. The *DAE* package defines the *DAE* equation representation, but also has a few routines to emit equations via the *Dump* package.

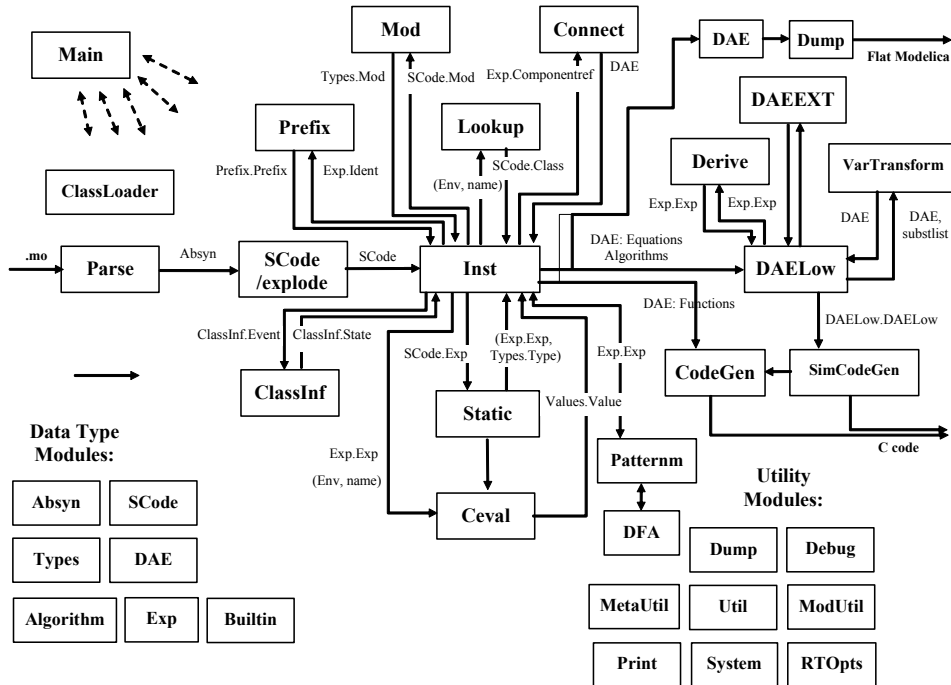


Figure 4-3. OpenModelica compiler packages and their connection.

A short description of the most important packages is provided below:

- The *Main* package calls a number of functions in other packages, including the parser package *Parse*, etc.
- The parser generates abstract syntax (provided by the *Absyn* package) which is converted to the simplified intermediate form (specified in the *SCode* package).
- The code instantiation package *Inst* is the most complex module, and calls many other packages. It calls *Lookup* to find a name in an environment, *Prefix* for analyzing prefixes in qualified variable designators (components), *Mod* for modifier analysis and *Connect* for connect equation analysis. It also generates the DAE equation representation which is simplified by *DAELow* and fed to the *SimCodeGen* code generator for generating equation-based simulation code, or directly to *CodeGen* for compiling Modelica/MetaModelica functions into C functions
- The *Ceval* package performs compile-time or interactive expression evaluation and returns values. The *Static* package performs static semantics and type checking.
- The *DAELow* package performs BLT sorting and index reduction. The *DAE* package internally uses *Exp.Exp*, *Types.Type* and *Algorithm.Algorithm*; data structures.

- The *Vartransform* package called from *DAELow* performs variable substitution during the symbolic transformation phase (BLT and index reduction).
- The *Patternm* package performs compilation of pattern match expressions, calling the *DFA* and *MetaUtil* packages.

4.4 High-level Data Structures Implementation

The implementation of the MetaModelica language extensions in the OpenModelica compiler involves the addition of several high-level data structures: union types, lists, tuple types and option types. We describe the general course of action for adding these novel high-level data structures. We refer to Figure 4-3 for an overview of the most important packages of the OpenModelica compiler and their interactions.

Generally, a new data structure type must be added to the compiler type system. Adding a new simple type to the compiler (such as an integer type) is a relatively straightforward process: the new type is added to the type system package (*Types*) and rules for matching expressions of this new type are added as well. In the back-end the new type should be matched against a corresponding type in the target language. Minor changes in a few other packages are needed as well.

However in this case we are dealing with high-level and, in some cases, parameterized data types, which need to be handled in a different way. The array parameterized data type, for instance, is treated in a separate manner in the OpenModelica compiler.

The new data structures may come with new syntax (other than the type keyword). For instance the list data structure uses new syntax for declaring the list type as well as list constructor syntax for building lists.

Shortly, the implementation for the extensions with the four high-level data structures mentioned above (union types, lists, tuple types and option types) follows these steps:

- Addition to the parser and the abstract syntax package. Note that lists, tuples and option type variables are parsed as variables of new complex types and union type variables are parsed as variables of a new restricted class.
- New type matching rules to the type system, etc.
- New expressions associated with the new data structures need to be handled. For instance the cons-constructor expression (`:`) in connection with the list type or the union type record constructor call - `MyRecord(1, 2, 3, 4)`.
- A union type restricted class declaration is treated in a special manner. Lists, tuples, and option types do not involve class declarations (tuples and option types can be said to involve class declarations explicitly).
- The new types should be handled as input and output to functions and in match expressions.

- A declaration of a variable of the new types has to be treated separately in the instantiation phase (the *Inst* package).
- The new types and the corresponding expressions and constructs are mapped to suitable target code constructs.

A description of the main packages that have been modified follows:

Absyn: New abstract syntax for the constructs has been added to this package.

Parser: ANTLR is used as an external OpenModelica parser. From a formal grammar, ANTLR generates a program that determines whether sentences conform to the language. Typically only a lexer and a parser are used but in the ANTLR case there is also a walker. The walker maps the abstract syntax from the parser into the abstract syntax of the OpenModelica Compiler, given by the constructs in *Absyn.h* (generated from the *Absyn* package). The new syntax constructs have been added to the lexer, parser and walker.

ClassInf: New states for the new types have been added to this package.

Codegen: In this package the new variable types are mapped to void-pointers. Expressions, such as the `Exp.CONST` expression representing the list cons constructor, are mapped into boxes consisting of two fields: a header and the data (in this case a first and a second field).

The same strategy is applied for union types, option types and tuples – they are all represented as boxes and void-pointers to these boxes. An option type variable will thus result in a void-pointer that either points to a nil-symbol/empty box or a normal box, in the generated code.

Exp: This package contains expressions after the instantiation phase, that is, expression with type information and which have been, perhaps, constant evaluated.

Inst: This is one of the most complex package of the compiler. In the function `instElement` a new case-branch has been added that takes care of variable declaration of the new MetaModelica types.

One must consider the fact that a type may be derived. When handling a variable/component the function `instElement` will lookup the type of the variable in the environment.

The type information is stored in a `SCode.Class` structure. This is true for both builtin types and derived types. In the new case-branch for the explicit MetaModelica variable declaration a `SCode.Class` structure with derived type is created (and not looked up in the environment).

Both the new case-branch as well as the case-branch for normal variables/components will call `instVar` which will call functions for instantiating the type class information: `instClass`, `instClassIn` and `instClassDef`. In `instClassDef` new case-branches have been added to handle `SCode.Classes` of derived MetaModelica types. These case-branches may call `instClass` recursively since we may have recursive type declarations.

Prefix: New rules for prefixing the new expressions have been added.

SCode: The union type restricted class declarations are transformed into more suitable form in this package.

Static: In the function `elabExp` (the function that elaborates expressions) new rules have been added that elaborates for instance an `Exp.CONS` expression.

Types: New type records have been added to this package. In the function `subtype`, rules for matching these new types have been added.

For more information regarding the implementation of high-level data structures in the OpenModelica compiler the reader is referred to (Björklén 2008 [13])

4.5 Pattern Matching Implementation

To achieve the meta-programming facilities in the OpenModelica compiler we have designed and implemented a pattern matching compiler. 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).

The pattern matching construct has been implemented in OMC using the algorithm described in (Pettersson 1999 [122]). 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 further code replication. The end result will have 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 SML 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.

4.5.1 Implementation Details

The specific OpenModelica translation path for Modelica code with `matchcontinue` constructs is presented in Figure 4-4. The `matchcontinue` expression has been added to the abstract syntax, `Absyn`. The pattern matching algorithm is invoked on the `matchcontinue` expression in the `Inst` package. The main function of the pattern matching algorithm is `PatternM.matchMain` which is given in a light version below.

The first function to be called in `matchMain` is `ASTToMatrixForm` 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 does not need to pass along a lot of extra code. The code in the right-hand sides is saved in another list and is later added.

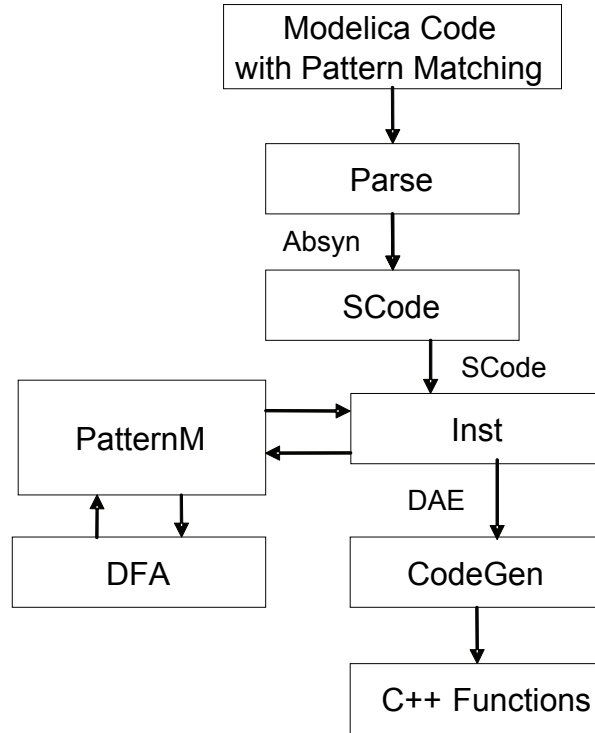


Figure 4-4. Pattern Matching Translation Strategy.

The `matchMain` function is presented below.

```

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.fromDFAtToIfNodes(dfaRec,...,  
                           localCache,localEnv,...);  
    then (localCache,expr);  
end matchcontinue;  
end matchMain;
```

After this step, 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. The function (roughly) distinguishes between three cases:

- All the patterns in the uppermost matrix row are wildcards.
- All 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 explained in (Pettersson 1999 [122]). 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 `fromDFAtToIfNodes`. 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 4.5.1.3). C++ code is then generated for the value block expression in the Codegen package.

4.5.1.1 Example of Code Generation

We first give an example of the compilation of a `matchcontinue` expression over simple 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)  
            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 4-4. 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 4-5 for the generated DFA.

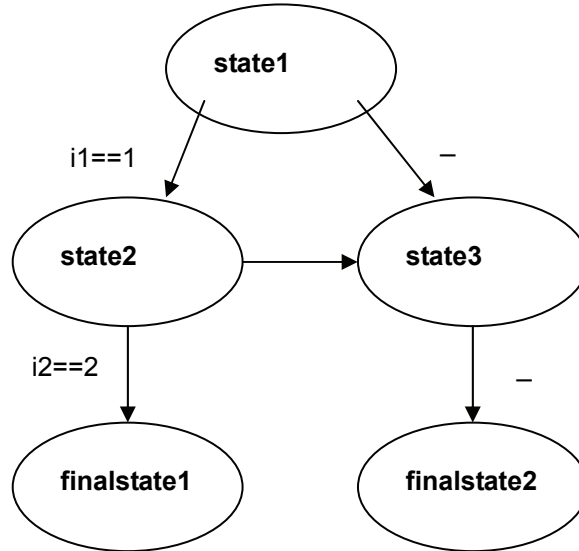


Figure 4-5. Code Example Generated DFA.

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.

4.5.1.2 Pattern Matching over Union, Lists, Tuples and Option Types

The remaining MetaModelica constructs (that are not present in Modelica) are currently being added to OMC: lists, union types, option types and tuples.

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 number zero (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;

```

Will result in:

```

if (true)
then
  Integer      $x1 = getVal(x, 1);
  list<Integer> $x2 = getVal(x, 2);
  if ($x1 == 1) then
    ...
  elseif (...)
    ...
  end if;
end if;

```

The symbol `::` is the cons constructor. 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;
```

Will result in:

```
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.

4.5.1.3 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.

4.6 Exception Handling Implementation

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

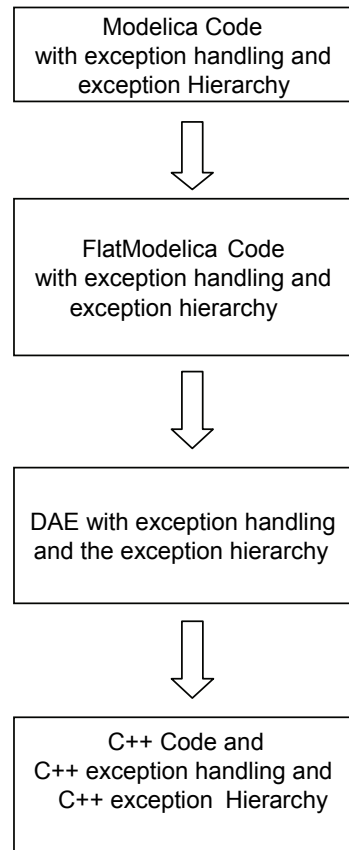


Figure 4-6. Exception handling translation strategy.

The general translation of Modelica with exception handling follows the path described in Figure 4-6. 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 4-7.

Implementing exception handling support in the OpenModelica compiler 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.

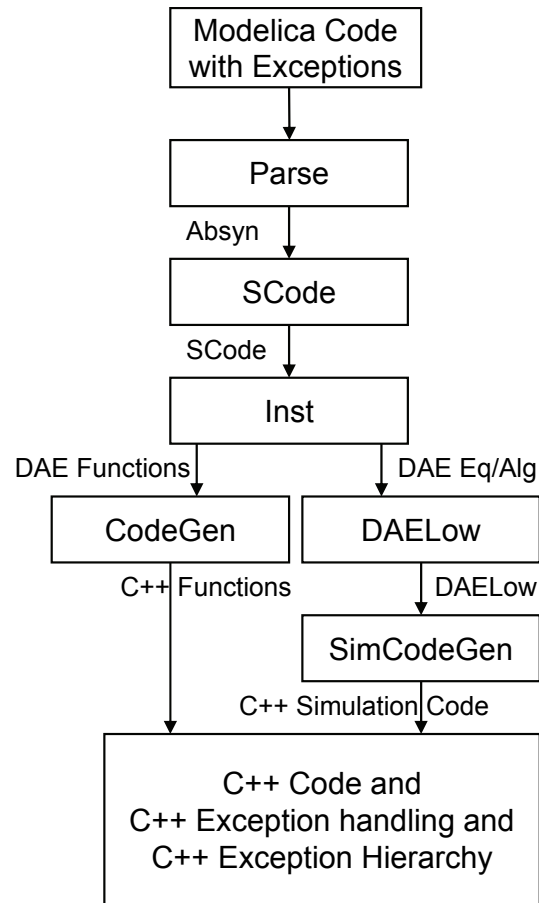


Figure 4-7. OpenModelica implementation.

4.6.1 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);`.

4.6.2 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++
<pre>x := try exp1 catch(E e) exp2 end try;</pre>	<pre>modelica_type temp; try { temp = exp1; } catch(E *e) { temp = exp2; } x = temp;</pre>
Modelica Statements	C++
<pre>try <stmts> catch(E e) <stmts> end try;</pre>	<pre>try { // Modelica corresponding // C++ statements } catch(E *e) { // Modelica corresponding // C++ statements }</pre>
Modelica Equations	C++
<pre>try <eqnsA> catch(Ex1 e1) <eqnsB> end try;</pre>	<pre>event1=false; event2=false; while time < stopTime {</pre>

<pre> try <eqnsC> catch (Ex2 e2) <eqnsD> end try; </pre>	<pre> try{ <i>call SOLVER for problem:</i> if event1 then eqnsB; else eqnsA end if; if event2 then eqnsD; else eqnsC; end if; } catch (Ex1 *e1) { <i>discard possible calculated current</i> <i>step values;</i> <i>reinit the solver with previous step</i> <i>values;</i> event1 = true; } catch (Ex2 *e2) { <i>discard possible calculated current</i> <i>step values;</i> <i>reinit the solver with previous step</i> <i>values;</i> event2 = true; } </pre>
---	--

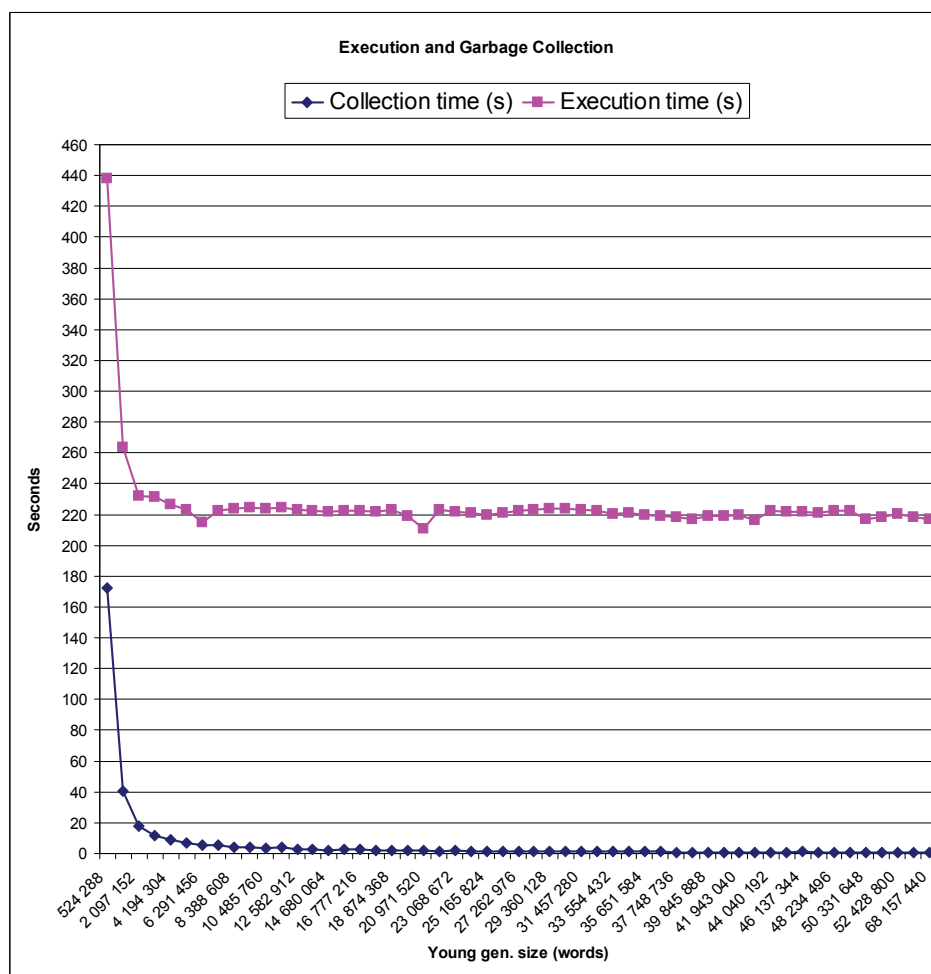
4.7 Garbage Collection

Garbage collection features relieves the programmer from the task of allocating and freeing memory. A very good survey of garbage collection is given in (Wilson 1994 [174]).

The OpenModelica compiler runtime features a generational garbage collector with two regions: young and current. The collector was ported and adapted from the MetaModelica compiler prototype. During execution, the data is allocated into the young region. When the young region fills a minor collection takes place and the live data is copied into the current region. When the current region is 80% filled a major collection takes place and the live data from the current region is copied to the reserve region and the regions switch places. If after a major collection the current region is still 80% filled then the current region is expanded so that is only 20% filled.

4.7.1 Layout of Data in Memory

All variable values (except 31 bit integers) are boxed to be distinguished by the garbage collector. Every boxed value has a small integer as its header. Composite values are boxed structures. The structure header contains a small integer tag which is used for pattern matching. Logical variables are represented as boxed references. A different header is used to represent unbounded or bounded logical variables.



4.7.2 Performance Measurements

We have measured the performance of the OpenModelica runtime system garbage collector. The OpenModelica compiler was instructed to run a script that:

- Loads a large model `RRLargeModel2.mo` of 1659 equations/variables. More info about the test files is given in section 5.4.2 and information about the test machine in section 5.4.1.
- Executes a check of the loaded model.

The OpenModelica compiler was executed multiple times with different young generation sizes and the execution time of the garbage collection time was generated together with the total execution time. The results are presented in Figure 4-8. At a young region of ~16MB (4MWords) the GC time is below 10 seconds out of 230 seconds total execution time which is ~4%. The GC time varies between 40% for a really small young region to 0.25% for a large young region. Increasing the young region over 80MB (20 MWords) does not improve the execution time.

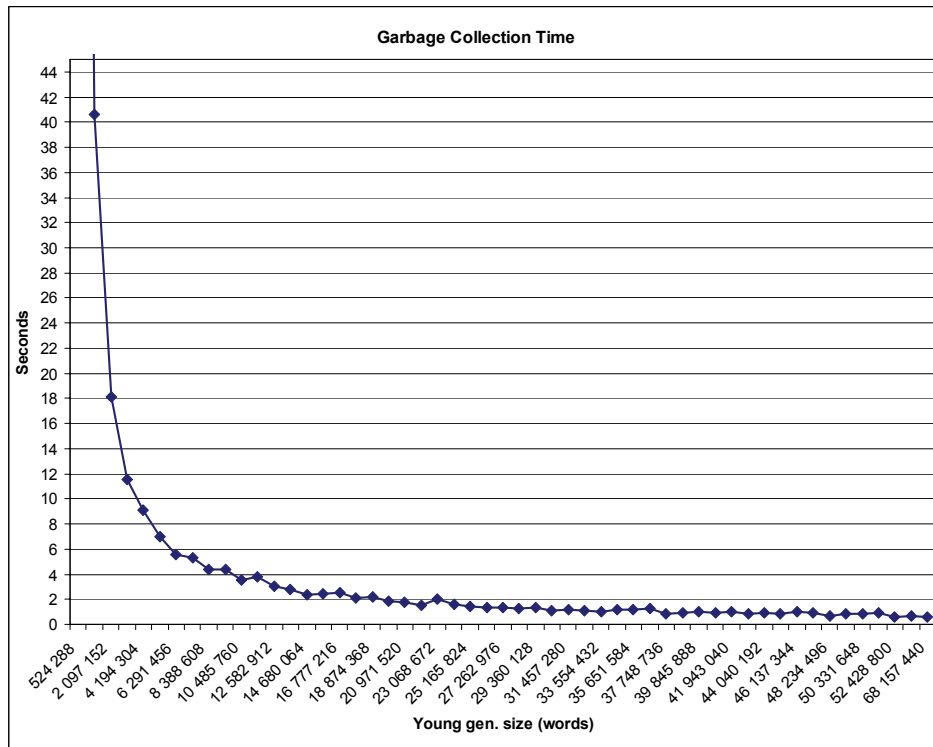


Figure 4-9. Garbage Collection time (s).

The table below presents the entire dataset for the garbage collector performance results.

Table 4-2. Garbage Collection Performance.

Young generation (words)	Current generation (words)	Collection time (s)	Execution time (s)	Minor collections	Major collections
524288	3787930	172.33	437.88	23681	4986
1048576	4194304	40.65	263.31	11840	824
2097152	8388608	18.08	231.81	5920	180
3145728	12582912	11.54	231.14	3946	77
4194304	16777216	9.09	226.75	2960	42
5242880	20971520	6.96	222.91	2368	26
6291456	25165824	5.56	215.01	1973	18
7340032	29360128	5.27	222.58	1691	13
8388608	33554432	4.36	224.02	1480	10
9437184	37748736	4.39	224.36	1315	8
10485760	41943040	3.51	223.81	1184	6
11534336	46137344	3.83	224.22	1076	6
12582912	50331648	3.03	223.02	986	4
13631488	54525952	2.78	222.14	910	3
14680064	58720256	2.39	222.06	845	3
15728640	62914560	2.42	222.69	789	2
16777216	67108864	2.51	222.36	740	2
17825792	71303168	2.07	221.51	696	2
18874368	75497472	2.21	223.39	657	2
19922944	79691776	1.83	218.75	623	1
20971520	83886080	1.75	210.81	592	1
22020096	88080384	1.51	223.09	563	1
23068672	92274688	2.01	221.92	538	1
24117248	96468992	1.58	220.77	514	1
25165824	100663296	1.46	219.67	493	1
26214400	104857600	1.33	221.06	473	1
27262976	109051904	1.38	222.74	455	0
28311552	113246208	1.29	223.16	438	0

29360128	117440512	1.39	223.56	422	0
30408704	121634816	1.06	223.84	408	0
31457280	125829120	1.16	223.01	394	0
32505856	130023424	1.11	222.75	381	0
33554432	134217728	1.05	220.17	370	0
34603008	138412032	1.21	220.78	358	0
35651584	142606336	1.16	219.81	348	0
36700160	146800640	1.23	219.13	338	0
37748736	150994944	0.87	218.66	328	0
38797312	155189248	0.94	217.14	320	0
39845888	159383552	0.99	219.24	311	0
40894464	163577856	0.91	218.97	303	0
41943040	167772160	1.01	220.02	296	0
42991616	171966464	0.81	216.11	288	0
44040192	176160768	0.91	222.53	281	0
45088768	180355072	0.85	221.86	275	0
46137344	184549376	1.05	221.63	269	0
47185920	188743680	0.91	221.25	263	0
48234496	192937984	0.71	222.41	257	0
49283072	197132288	0.85	222.58	251	0
50331648	201326592	0.81	216.81	246	0
51380224	205520896	0.93	218.42	241	0
52428800	209715200	0.58	220.19	236	0
62914560	251658240	0.66	218.45	197	0
68157440	272629760	0.56	217.25	182	0

4.8 Conclusions

This chapter presented the existing MetaModelica compiler prototype and our current work targeting the OpenModelica compiler bootstrapping.

Also, the MetaModelica compiler prototype implementation is presented and its performance compared to related systems is evaluated. The performance results show that the prototype is robust and generates very efficient code.

The chapter further presents implementation of high-level data structures, pattern matching and exception handling in the OpenModelica compiler.

The garbage collector of the OpenModelica compiler is presented and evaluated. The performance results show that the collector is efficient enough and the collection time takes a rather small part of the total execution time. In the future, further development (increasing the number of generations, allocation of similar structures in different regions without any header, etc) of the garbage collector could be investigated.

Part III

Debugging of Equation-based Object Oriented Languages

Chapter 5

Portable Debugging of EOO Meta-Programs

5.1 Introduction

The OpenModelica compiler is built from a large specification of the Modelica language written in MetaModelica. Further development of such a large specification is difficult without debugging tools. This chapter presents the design, implementation and evaluation of several debugging frameworks for MetaModelica. During his PhD work the author has designed and implemented four debugging frameworks (two for Natural Semantics specifications and two for MetaModelica specifications) supported by different integrated environments: Emacs, Eclipse-based Modelica Development Tooling (MDT), and Eclipse-based Structural Operational Semantics Development Tooling (SOSDT).

5.2 Debugging Method – Code Instrumentation

Our debugging implementation approach is based on instrumentation of the intermediate code representation (IR). During compilation the IR is instrumented with debugging nodes which are just calls to a debugging API.

The first debugging framework adds the debugging instrumentation very early in the compilation process, at the abstract syntax tree representation. We call this method *early instrumentation*.

The second debugging framework adds the debugging instrumentation very late in the compilation process, at the code representation. We call this method *late instrumentation*.

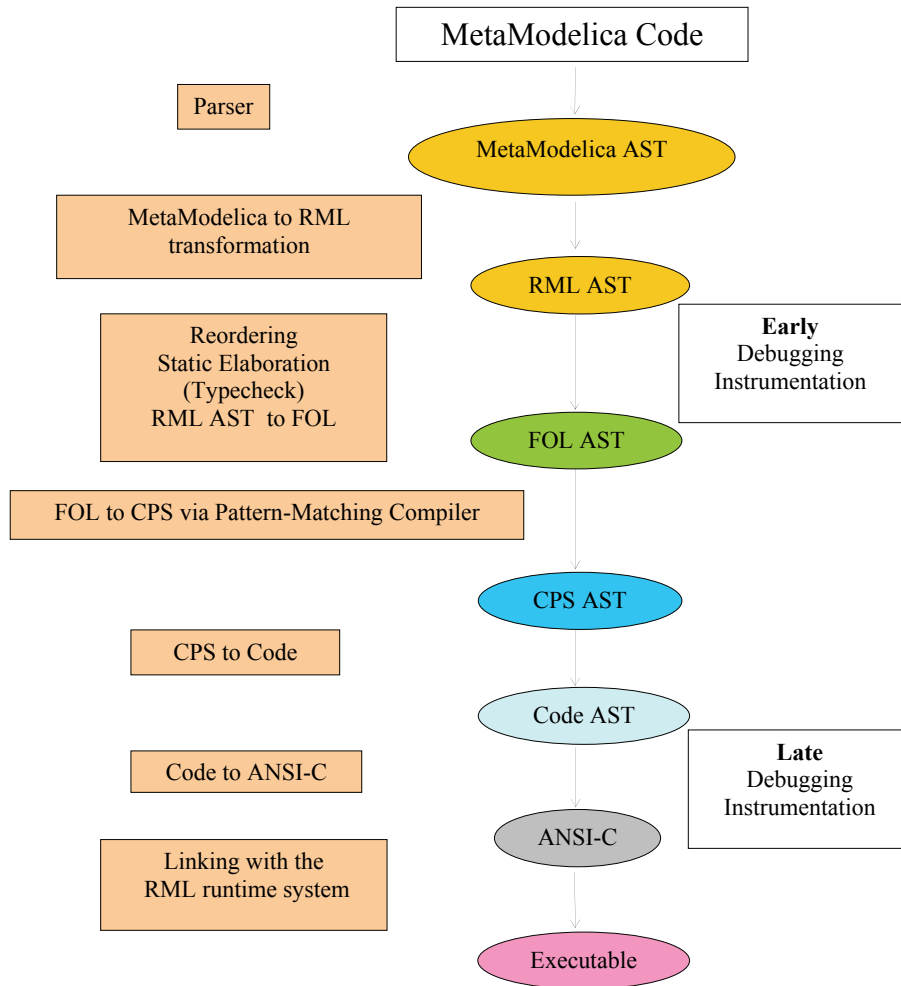


Figure 5-1. Early vs. Late Debugging Instrumentation in MetaModelica compiler.

5.2.1 Early Instrumentation

The design, implementation and evaluation of the debugging framework based on early instrumentation is presented in Chapter 7 and in (Pop and Fritzson 2005 [127], Pop and Fritzson 2005 [128]).

5.2.2 Late Instrumentation

The debugging framework based on early instrumentation was a positive start which encouraged us to experiment more with this idea and try to improve the compilation and run-times.

The debugging framework based on late instrumentation is an improvement of the early instrumentation debugging framework. We disabled the early instrumentation phase in the compiler and added a new phase closer to code generation. As a consequence we had to pass the debugging information (position of identifiers, function calls, type information, etc) through all the compiler phases.

5.3 Type Reconstruction

During debugging, both values and the types of the variables need to be available to the user. To provide type information for the user, the runtime system of the MetaModelica compiler prototype and the compiler itself had to be extended. In the following we present the type reconstruction procedure implemented in all the debugging frameworks we developed.

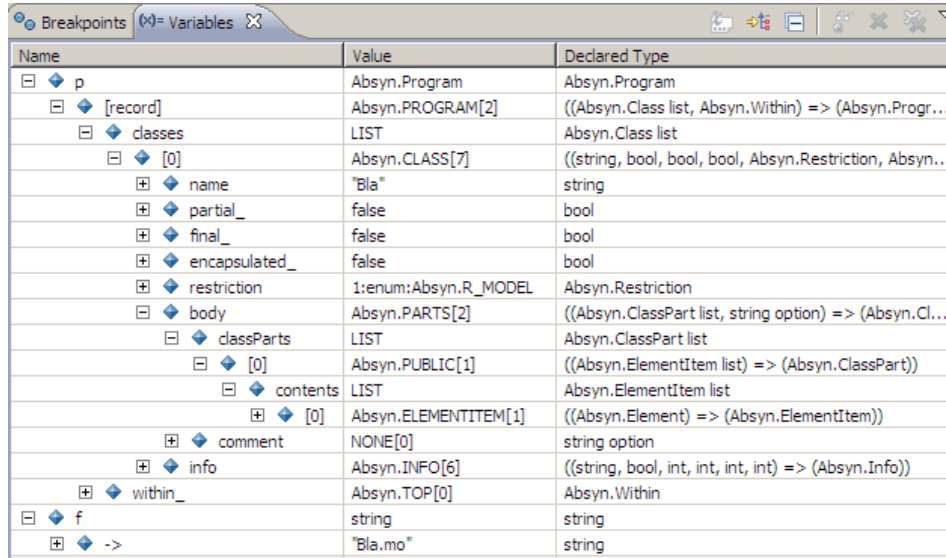
During the compilation phase the types of all the variables and the variable scope in the program is recorded in a program database for each package. During code generation the program database for a package is stored as static information (using C structures and variables) in the generated C code for that particular package. Our first debugging framework generated separate files with the program database for each package; this proved out to be very problematic as these additional files had to be stored in the same directory with the executable and the executable had to read and parse these additional files at startup (see more .in Chapter 7).

Before and after each function call the available (live) variables and the pointer to their boxed value are registered with the debugging runtime. During execution, when the debugger stops at a breakpoint, the available (live) variables are queried for their position in the source code (package, function and line number) and for their value pointer (the value pointer points to the boxed representation of the variable value). The position information for a variable is used to query the program database to fetch its type declaration. The debugger now has two structures:

- The type of the variable
- The pointer to the boxed variable value

These two structures are processed top-down simultaneously to output a variable value and its specified type. If the variable value represents a complex data structure (for example an AST representation) then the components of the variable value are matched with the components of the type declaration and the procedure continues recursively until the entire value is presented. Our first debugging framework printed the values on the standard output. The latest debugging

framework sends the value information (including type information) to the Eclipse (Eclipse.Foundation 2001-2008 [29]) environment for display.



Name	Value	Declared Type
p	Absyn.Program	Absyn.Program
[record]	Absyn.PROGRAM[2]	((Absyn.Class list, Absyn.Within) => (Absyn.Progr...
classes	LIST	Absyn.Class list
[0]	Absyn.CLASS[7]	((string, bool, bool, bool, Absyn.Restriction, Absyn...
name	"Bla"	string
partial_	false	bool
final_	false	bool
encapsulated_	false	bool
restriction	1:enum:Absyn.R_MODEL	Absyn.Restriction
body	Absyn.PARTS[2]	((Absyn.ClassPart list, string option) => (Absyn.Cl...
classParts	LIST	Absyn.ClassPart list
[0]	Absyn.PUBLIC[1]	((Absyn.ElementItem list) => (Absyn.ClassPart))
contents	LIST	Absyn.ElementItem list
[0]	Absyn.ELEMENTITEM[1]	((Absyn.Element) => (Absyn.ElementItem))
comment	NONE[0]	string option
info	Absyn.INFO[6]	((string, bool, int, int, int, int) => (Absyn.Info))
within_	Absyn.TOP[0]	Absyn.Within
f	string	string
->	"Bla.mo"	string

Figure 5-2. Variable value display during debugging using type reconstruction.

5.4 Performance Evaluation

This section presents an evaluation of the debugging frameworks based on early and late instrumentation. We tested the compile times and run-times of the compiled programs.

5.4.1 The Test Machine

The tests were run on a HP NC6400 laptop with 2GB of memory and a Core 2 Duo processor at 2GHz with Windows XP.

5.4.2 The Test Files

The MetaModelica compiler is a compiler-compiler, it takes as an input a compiler specification and generates as output an executable compiler for that specification. To test our debugging frameworks we compiled the OpenModelica compiler specification and measured the compilation times and execution times of the resulting compiler.

The OpenModelica compiler specification is very large:

- 4,65 MB of MetaModelica sources, ~140 000 lines of code
- 52 Packages
- 5422 Functions

To test the speed of the generated code with debugging we ran the OpenModelica compiler on:

- A large model; `RRLargeModel2.mo` provided by MathCore engineering. The model has 1659 equations/variables and ~27108 lines of code. The model can be provided on request.
- A small model: `BouncingBall.mo` presented below. This model has 5 equations/variables and is part of the OpenModelica release.

The `BouncingBall.mo` model:

```

model BouncingBall
  parameter Real e=0.7 "coefficient of restitution";
  parameter Real g=9.81 "gravity acceleration";
  Real h(start=1) "height of ball";
  Real v "velocity of ball";
  Boolean flying(start=true) "true, if ball is flying";
  Boolean impact;
  Real v_new;
equation
  impact = h <= 0.0;
  der(v) = if flying then -g else 0;
  der(h) = v;
  when {h <= 0.0 and v <= 0.0, impact} then
    v_new = if edge(impact) then -e*pre(v) else 0;
    flying = v_new > 0;
    reinit(v, v_new);
  end when;
end BouncingBall;

```

The OpenModelica compiler was instructed using scripts to load the models and run a check on them. For example the script `RRLargeModel2.mos` has the following contents:

```

loadFile("RRLargeModel2.mo");
checkModel(RRLargeModel2);

```

The script for loading and checking the `BouncingBall` model is similar to the one above:

```

loadFile("BouncingBall.mo");
checkModel(BouncingBall);

```

The `checkModel` function instantiates (flattens) the model, generates the hybrid DAE equation system and verifies if the system is balanced (number of equations is equals with the number of variables) hence solvable.

5.4.3 Compilation Performance

The performance of the MetaModelica system while compiling the OpenModelica specification is presented below.

The translation time was calculated by running the MetaModelica system on the .mo files until C code is generated. The total compilation time includes also the compilation of each generated C file using gcc with the highest optimization level (-O3) and the linking time.

The number of generated C functions is higher for the late instrumentation debugging as some of the low level optimizations are disabled to achieve one to one mapping of debugging information to the C code.

The size of the generated C code source is larger for early instrumentation because the high-level optimizations cannot be applied in the presence of early debugging instrumentation.

The compilation time with late instrumentation debugging is roughly 3 times slower (and with early instrumentation about 4 times slower) due to increased code size. These results are comparable to the debugger (Tolmach 1992 [155]) for Standard ML designed and implemented by Andrew P. Tolmach in the Standard ML of New Jersey (SML/NJ) system. He reports a compilation slowdown by a factor of 5.

Table 5-1. Compilation performance (no debugging vs. early vs. late instrumentation)

	Gen. C sources (MB)	No. gen. C functions	Translation time (s)	Total Compilation time (s)
No debugging	37	25 027	131.78	269.86
Early instrumentation	130	52 241	155.16	850.35
Late instrumentation	103	95 560	179.38	610.61

5.4.4 Run-time Performance

The run time performance of the generated OpenModelica compiler on the scripts `RRLargeModel2.mos` and `BouncingBall.mos` is presented below.

The execution time with late instrumentation debugging is about 4 times slower than with no debugging and about 6 times faster than the execution time with early instrumentation debugging. These results are comparable to the Standard ML of New Jersey (SML/NJ) debugger (Tolmach 1992 [155]) where they report a execution slowdown by a factor of 3 due to code instrumentation.

The stack usage is about the same for the large model. For the smaller model the late instrumentation uses more stack as the optimization that moves code and inline functions is disabled because the type reconstruction procedure would not work otherwise.

Table 5-2. Running performance of script RRLargeModel2.mos.

	Running time (s)	Minor Collections	No. Function Calls	Stack (words)
No debugging	223.01	394	2 059 658 665	119 760
Early instrumentation	5395.47	565	3 654 044 108	119 912
Late instrumentation	864.36	421	2 077 495 068	119 780

Table 5-3. Running performance of script BouncingBall.mos.

	Running time (s)	Minor Collections	No. Function Calls	Stack (words)
No debugging	0.01	0	284 706	365
Early instrumentation	1.84	0	3 012 932	415
Late instrumentation	0.04	0	474 064	2221

5.5 Tracing and Profiling

Tracing and profiling are also supported for the MetaModelica compiler prototype. The tracing functionality is very useful at pinpointing the location (function name) if the compiler crashes due to programming errors and the profiling functionality can pinpoint function that need re-design to improve their execution speed.

5.5.1 Tracing

The tracing functionality is enabled in the MetaModelica compiler prototype by a compilation flag: `-ftrace`. The flag instructs the compiler to instrument all generated C functions with additional code that outputs the function name on the standard error. The generated executable only outputs the trace on the standard error if is given the `-trace` flag. The tracing functionality is very efficient at pinpointing where the executable crashes as the last function in the trace is where the error happened. The tracing functionality adds very little slowdown (~1.5%) to the generated executable as presented in Table 5-4.

Table 5-4. The impact of tracing on execution time.

	First run	Second run	Average
Without tracing (s)	213.09	212.78	212.935
With tracing (s)	216.33	215.88	216.105
Slowdown (%)	1.52	1,45	1,48

Because the executable compiled with tracing is very efficient, we decided to compile the OpenModelica compiler releases with tracing by default. This way if a compiler crash happens the user can re-run it with `-trace` and discover where the

error happened (function name). The user can then report the error and its location to the OpenModelica development team for investigation.

The table presents the performance evaluation of the OpenModelica compiler running times while executing `RRLargeModel2.mos` (presented in section 5.4.2) compiled with and without tracing enabled.

5.5.2 Profiling

Profiling of executables generated by MetaModelica compiler prototype is supported through GNU GCC profiling facilities and the GNU profiler `gprof`. Because the MetaModelica compiler prototype generates C code, the C code can be compiled with profiling instrumentation using GCC. The profiling functionality can be enabled in the MetaModelica compiler prototype by using the `-p` flag. The executables compiled with profiling will dump a `gmon.out` file when executed. To display the profile information of the executable one can run the GNU `gprof` tool:

```
adrpo@KAFFKA ~> gprof omc

Each sample counts as 0.01 seconds.
%   cumulative   self           calls   name
time   seconds seconds           name
34.53    37.25    37.25   384695481 System__hash
13.29    51.59    14.34   257363327 Env__treeAdd2
...
```

By analyzing the output produced by GNU `gprof` one can pinpoint what functions take the most of the execution time. Using this information the compiler developer can re-evaluate and re-design the functions that have the most impact on execution time.

5.6 The Eclipse-based Debugging Environment

We have developed an Eclipse-based debugging environment for the late instrumentation debugging framework. The Eclipse environment is implemented as a set of plugins which are available in the Modelica Development Tooling (MDT) environment (presented in Chapter 8). In this section we present the GUI facilities of the existing debugging functionality.

The debugger functionality is presented Figure 5-3. The figure presents a debugging session of the OpenModelica compiler specification stopped at a breakpoint set after the parser invocation. In the top-right part a complex variable value (the AST of the parsed model) is explored (browsed). In the top-left part the stack trace is presented. In the bottom-left part the execution point is shown. In the bottom-right part the contents of the Modelica file is presented (and the current function is outlined). The middle-left part presents the model that was given as input to the debugged compiler.

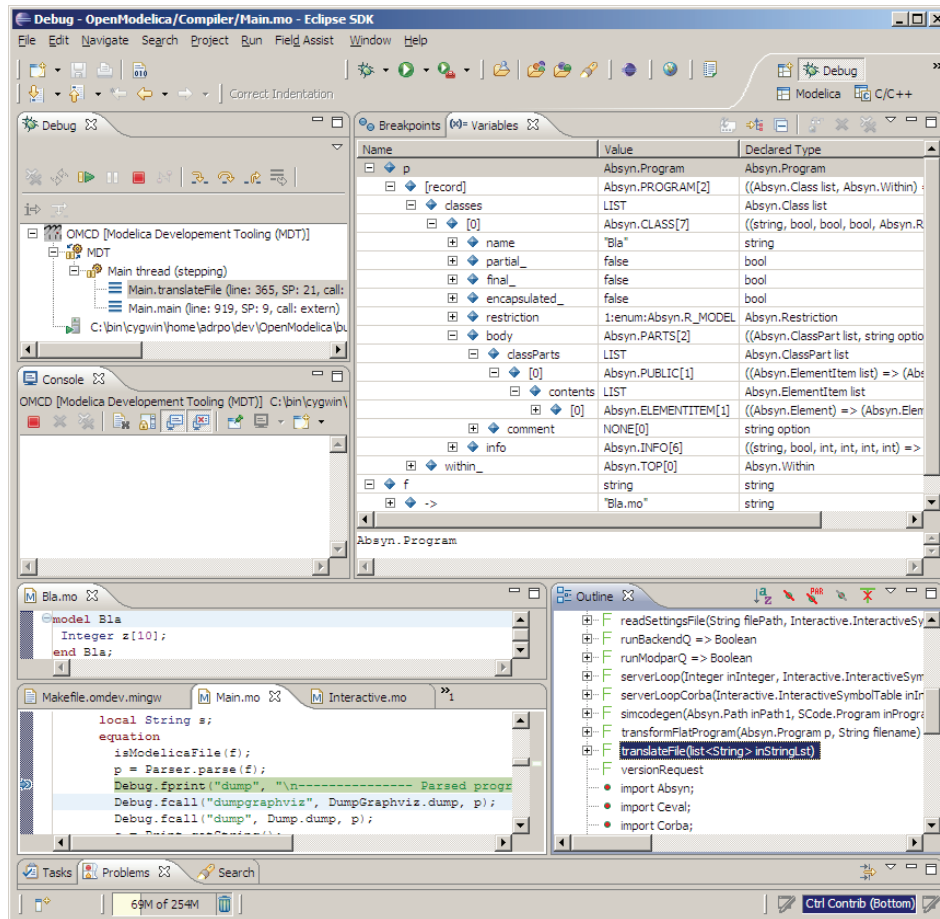


Figure 5-3. Advanced debugging functionality in MDT.

5.6.1 Starting the Modelica Debugging Perspective

The Eclipse platform provides several perspectives targeted to specific tasks (source code editing for a particular language, graphical modeling, debugging, etc). When a perspective is activated the environment configures itself to show and make available only the needed features for a particular task.

To be able to run in debug mode, the user has to go through the following steps:

- i) Creating and setting the debug configuration,
- ii) Setting breakpoints to stop the execution at interesting places,
- iii) Running the created debug configuration to start debugging.

All these steps are presented below using images.

5.6.2 Setting the Debug Configuration

While the Modelica perspective is activated, the user can select the bug icon on the toolbar and choose the Debug alternative in order to access the dialog for building debug configurations.

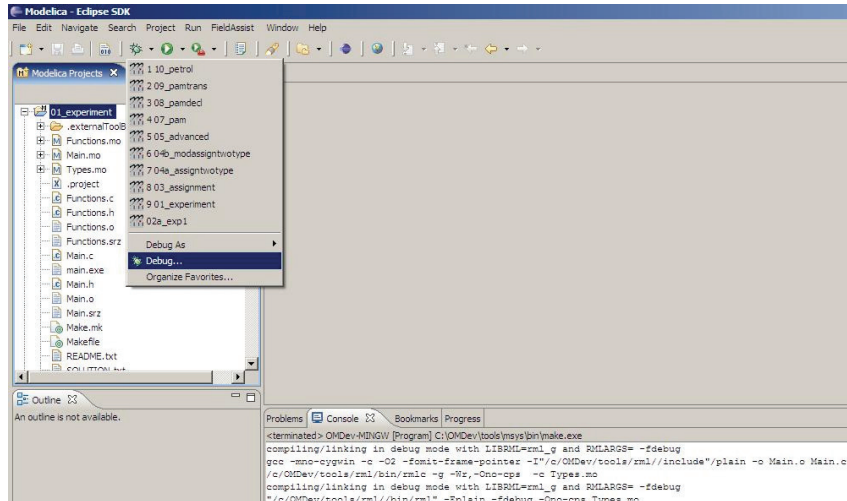


Figure 5-4. Accessing the debug configuration dialog.

To create the a debug configuration, the user can right click on the classification Modelica Development Tooling (MDT) and select New as in Figure 5-5. A name for the debugging configuration needs to be specified.

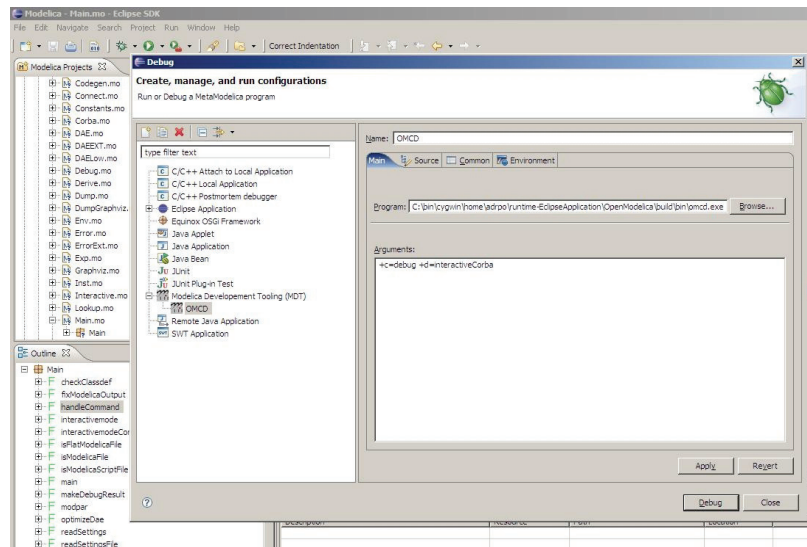


Figure 5-5. Creating the Debug Configuration.

The user also selects the executable to be debugged and provides command line parameters. The additional tabs available can be used for further debug configuration settings such as the environment in which the executable should be run.

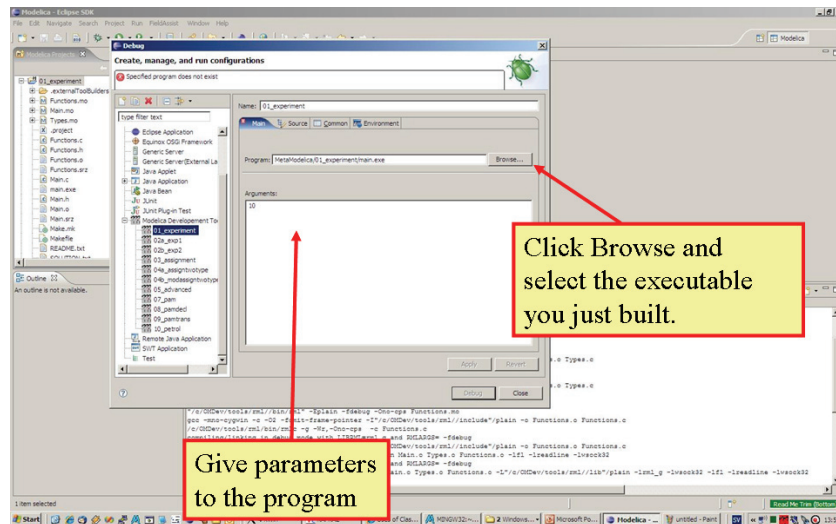


Figure 5-6. Specifying the executable to be run in debug mode.

5.6.3 Setting/Deleting Breakpoints

To enable breakpoints the user opens a file and double clicks on the editor ruler.

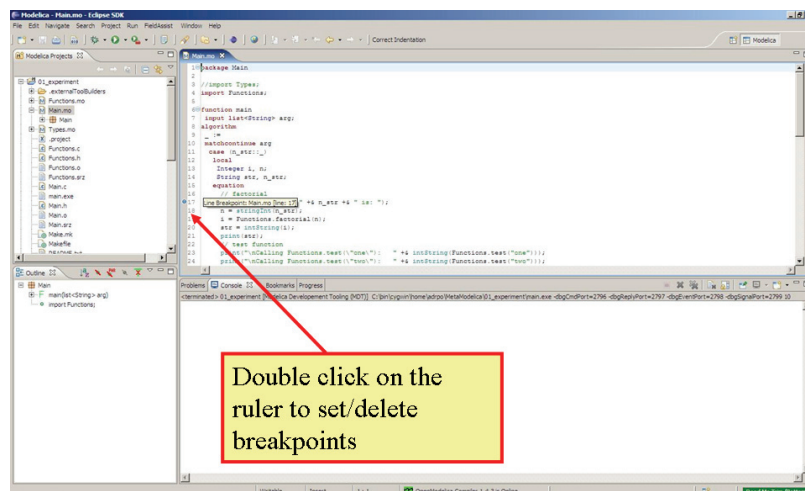


Figure 5-7. Setting/deleting breakpoints.

5.6.4 The Debugging Session and the Debug Perspective

The debugging session can be started from the menu by selecting the newly created debugging configuration.

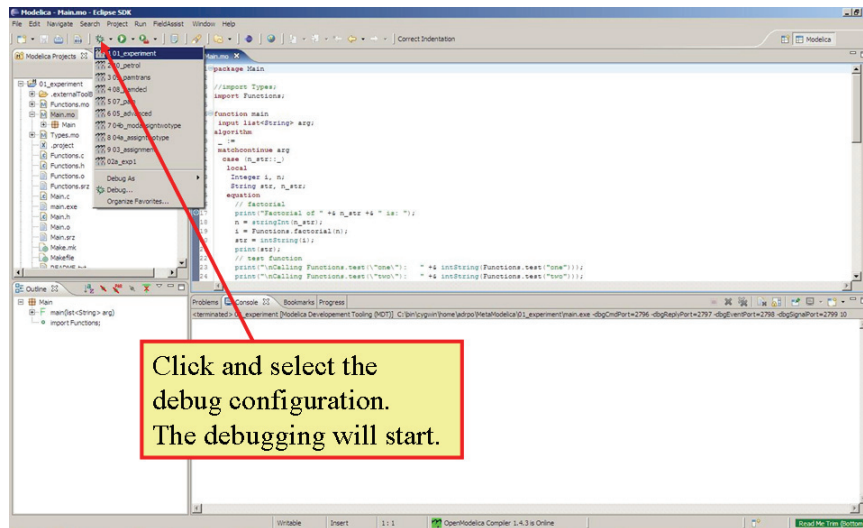


Figure 5-8. Starting the debugging session.

The Eclipse platform will automatically detect that a debugging session has started and will prompt the user to switch to the debugging perspective.

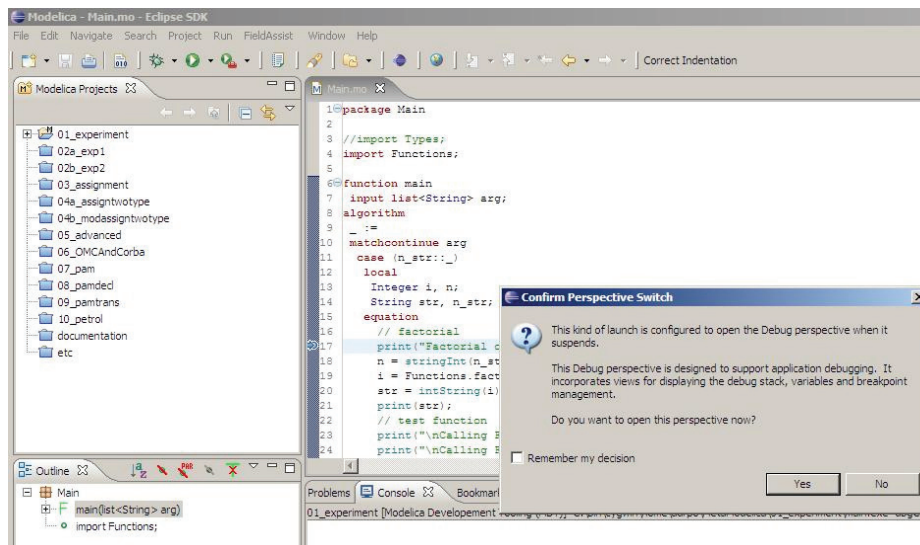


Figure 5-9. Eclipse will ask if the user wants to switch to the debugging perspective.

5.6.4.1 The Debugging Perspective

When the debugging perspective is selected by the user the environment activates and displays several views that are targeted to debugging: Variables, Breakpoints, Stack trace, Console and the Editor focused on the current execution point.

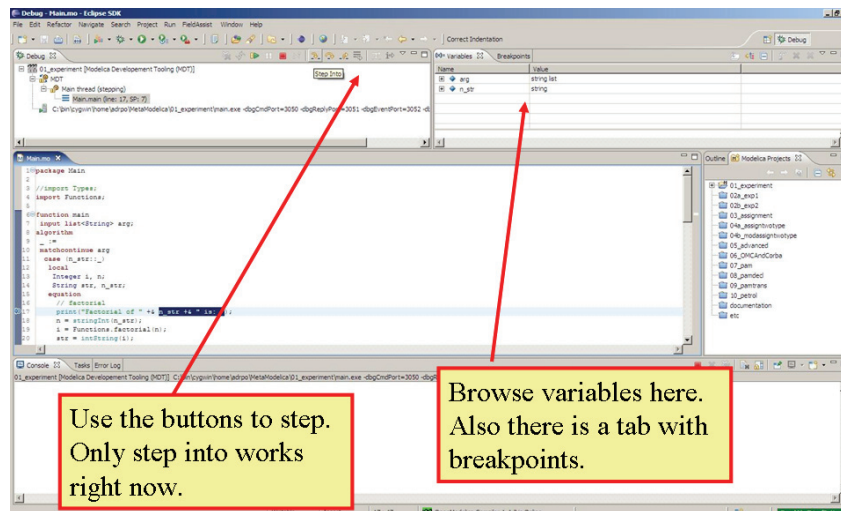


Figure 5-10. The debugging perspective.

At any time the user can switch between the available perspectives, activate additional views or change the placing of the views in the environment.

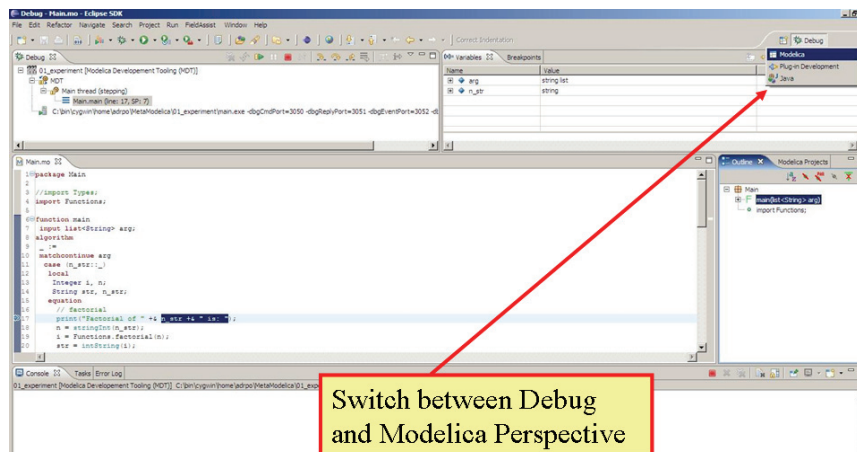


Figure 5-11. Switching between perspectives.

5.7 Conclusions

The increased ease of use, the high abstraction, and the expressivity of the MetaModelica language are very attractive properties. However, these properties come with the drawback that programming and modeling errors are often hard to find. To overcome these issues, several debugging methods and integrated frameworks for run-time debugging of the MetaModelica language have been designed, analyzed, implemented, and evaluated on non-trivial industrial applications.

We have presented in this chapter these portable debugging methods and their integration within the MDT development environment. The evaluation of the implemented debugging frameworks shows that the debugging methods are reliable and efficient. The chapter also considers tracing and profiling of MetaModelica code.

To conclude, this chapter presents a comprehensive Modelica debugger for an extended algorithmic subset of the Modelica language, including the meta-programming extensions. This replaces debugging of algorithmic code using primitive means such as print statements or asserts which is complex, time-consuming and error-prone. The debugger is portable since it is based on transparent source code instrumentation techniques that are independent of the implementation platform. The usual debugging functionality found in debuggers for procedural or traditional object-oriented languages is supported, such as setting and removing breakpoints, single-stepping, inspecting variables, back-trace of stack contents, tracing, etc. The debugger is integrated with the Modelica Development Tooling (MDT) environment within Eclipse. More information about MDT is given in Chapter 8.

Chapter 6

Run-time Debugging of EOO Languages

Random changes to a program fix bugs.

6.1 Introduction

The development of today's complex products requires advanced integrated environments and modeling languages for modeling and simulation. Equation-based object-oriented declarative (EOO) languages are emerging as the key approach to physical system modeling and simulation. The increased ease of use, the high abstraction and the expressivity of EOO languages are very attractive properties. However, these attractive properties come with the drawback that programming and modeling errors are often hard to find. In this chapter we propose an integrated framework for run-time debugging of equation-based modeling languages. The framework integrates classical debugging techniques with special techniques for debugging EOO languages and is based on graph visualization and interaction. The debugging framework targets the Modelica language.

6.2 Debugging Techniques for EOO Languages

In the context of debugging declarative equation-based object-oriented languages both the static and the dynamic (run-time) aspects have to be addressed.

The static aspect of debugging EOO languages deals with inconsistencies in the underlying system of equations:

- *Overconstrained system*: the number of variables is smaller than the number of equations, which means that some equations have to be removed when solving the system of equations.
- *Underconstrained system*: the number of variables is larger than the number of equations, which means that more equations have to be added in order to solve the system of equations.

The dynamic (run-time) aspect of debugging EOO languages addresses run-time errors that may appear due to faults in the simulated model:

- *Model configuration*: when parameters values for the model simulation are incorrect.
- *Model specification*: when the equations that specify the model behavior are incorrect.
- *Algorithmic code*: when the functions called from equations return incorrect results.

Methods for both static and dynamic (run-time) debugging of EOO languages have been proposed earlier (Bunus 2004 [19], Bunus and Fritzson 2003 [20]). With the new Modelica 3.0 language specification, static debugging of Modelica presents rather small benefits, since all model components are already required to be balanced. All models from checked libraries will already be balanced; only newly written models might be unbalanced.

In the context of the dynamic (run-time) aspect of debugging of EOO languages, (Bunus 2004 [19]) proposes an automated algorithmic debugging solution in which the user has to provide a correct diagnostic specification of the model which is used to generate assertions at runtime. Moreover, starting from an erroneous variable value the user explores the dependent equations (a slice of the program) and acts like an “oracle” to guide the debugger in finding the error.

In this chapter we present a different approach that does not require the user to write diagnostic specifications of the model. Our method is based the integration between graph visualization/interaction and execution-based debugging of algorithmic code.

6.3 Proposed Debugging Method

In this section we present our run-time debugging method. The proposed integration within a general debugging framework for EOO languages is presented in the next section.

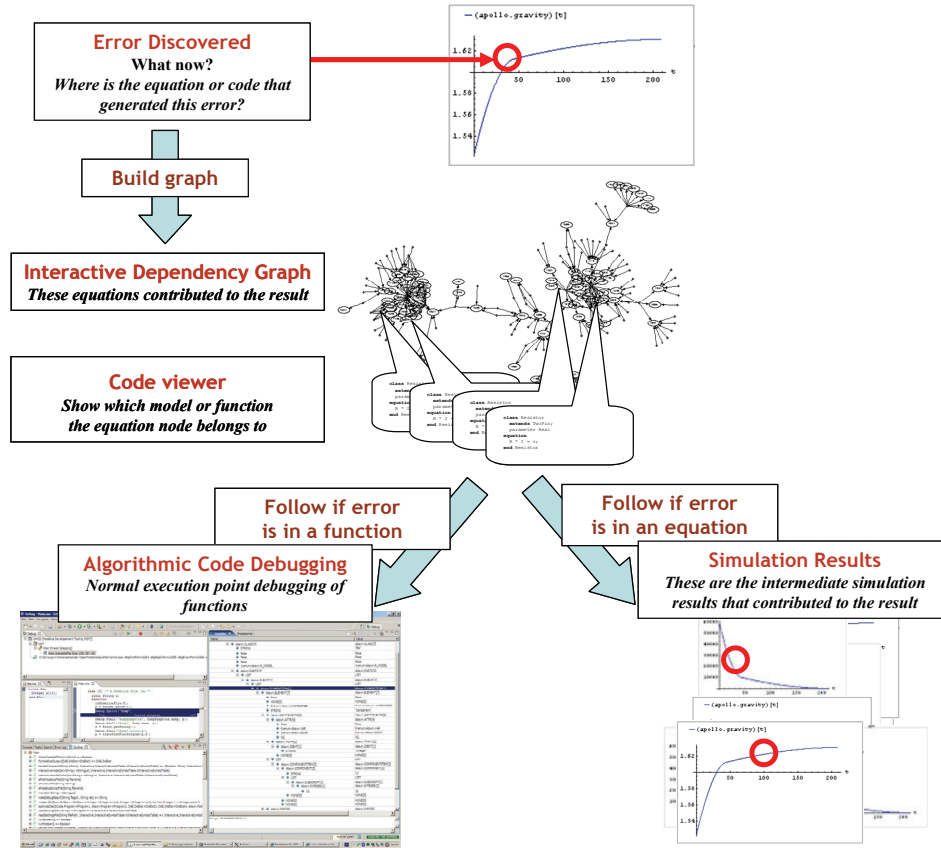


Figure 6-1. Debugging approach overview.

6.3.1 Run-time Debugging Method

Our method partly follows the approach proposed in (Bunus and Fritzson 2003 [20]). However, our approach does not require the user to write diagnostic specifications of models. Also, the approach we present here can also handle the debugging of algorithmic code using classic debugging techniques (Pop et al. 2006 [131]).

An overview of our debugging strategy is presented in Figure 6-1. In short, our run-time debugging method is based on the integration of the following:

- Graph visualization and interaction.
- Presentation of simulation results and modeling code.
- Mapping of errors to model code positions.
- Execution-based debugging of algorithmic code.

In the following we present a possible debugging session.

During the simulation phase, the user discovers an error in the plotted results. The user marks either the entire plot of the variable that presents the error or parts of it and starts the debugging framework. The debugger presents an (IDG) interactive dependency graph (the dynamic program slice with respect to the variable with the wrong value) where nodes consist of all the equations, functions, parameter value definitions, and inputs that were used to calculate the wrong variable value. The variable with the erroneous value is displayed in a special node which is the root of the graph. The interactive dependency graph contains two types of edges:

1. *Calculation dependency edges*: the directed edges labeled by variables or parameters which are inputs (used for calculations in this equation) or outputs (calculated from this equation) from/to the equation displayed in the node.
2. *Origin edges*: the undirected edges that tie the equation node to the actual model which this equation belongs to.

The user interacts with the dependency graph in several ways:

- *Displaying simulation results* through selection of the variables (or parameters) names (edge labels). The plot of a variable is shown in a popup window. In this way the user can quickly see if the plotted variable has erroneous values.
- *Classifying a variable* as having wrong values: addition of the variable to the set of variables with wrong values.
- *Classifying an equation as correct* eliminates the equation node from the graph and builds a new graph based on the inputs of the correct equation node.
- *Building a new dependency graph* based on the new set of variables with wrong values (classified variables) or by modifying the equations or parameter values nodes.
- *Displaying model code* by following origin edges.
- *Invoking the algorithmic code debugging subsystem* when the user suspects that the result of a variable calculated in an equation which contains a function call is wrong, but the equation seems to be correct.

Using these interactive dependency graph facilities the user can follow the error from manifestation to origin.

Our debugging method can also start from multiple variables with wrong values with the premise that the error might be at the confluence of several dependency graphs.

6.4 The Run-time Debugging Framework

In this section we present the first prototype of the debugging framework based on the proposed method from the previous section. The debugging framework is limited to error tracking of a single variable with wrong results.

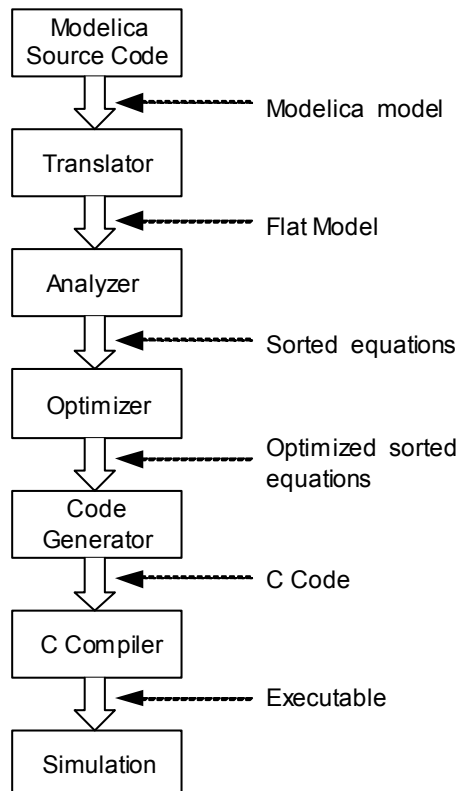


Figure 6-2. Translation stages from Modelica code to executing simulation.

6.4.1 Translation in the Debugging Framework

The debugging framework is closely related to the translation process. The translation process from the modeling language down to simulation code is presented in the following. The Modelica translation process has several stages (Figure 6-2):

- *Parser* – breaks the model down into tokens and builds the abstract syntax tree. (not in Figure 6-2)

- *Translator (Flattening and elaboration)* – reports the errors and flatten the model hierarchy and applying modification.
- *Analyzer* – analyses the system of equations and sorts the equations in the order they need to be solved
- *Optimizer* – optimizes the sorted system of equations
- *Code Generator* – generates C code linked with the simulation runtime and solvers.
- *C Compiler* – compiles the generated C code to an executable
- *Simulation* – the executable is executed to generate the simulation results.

As one can see, the translation process is complex and most of the transformations performed on the models are destructive. For debugging purposes all the transformations performed in each stage needs to be recorded to be able to point the errors to the user using the high level Modelica code.

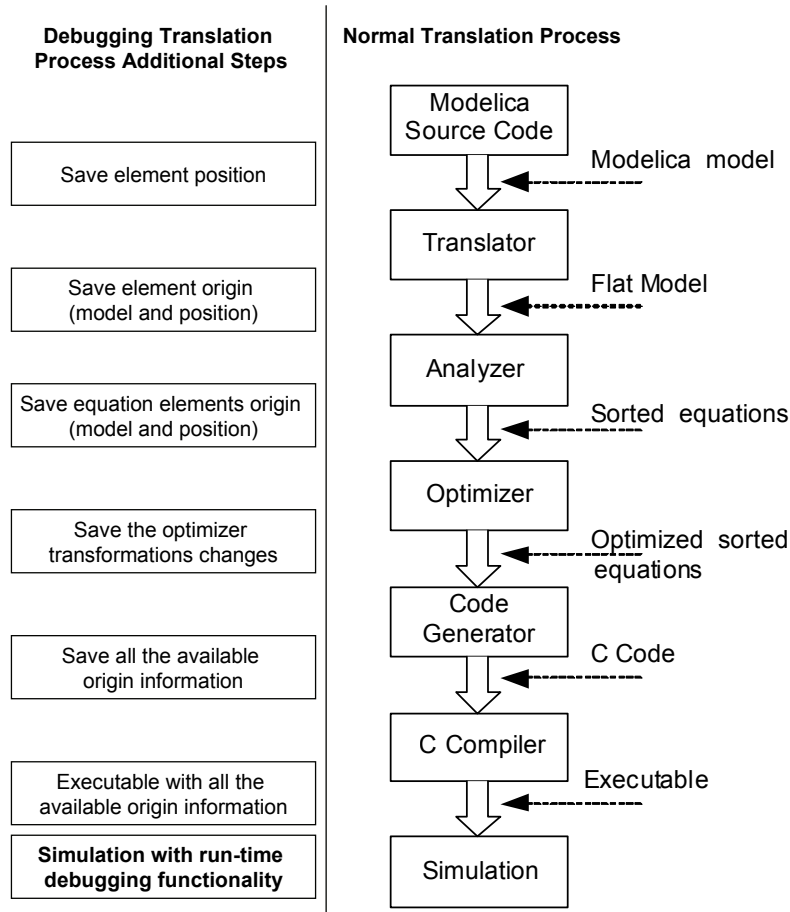


Figure 6-3. Translation stages from Modelica code to executing simulation with additional debugging steps.

The debugging framework alters the Modelica translation stages by introducing means to map (and save such mapping) each transformed model element back to its origin as presented in Figure 6-3.

The additional origin information needed by the debugging framework is saved by the debugging translation process within a file: `debug-info.xml`. The debug file is read by the simulation run-time only when needed.

If an error appears in the simulation results, the user can mark the variable with the wrong value and the error time interval(s) on the simulation plot. The simulation with run-time debugging functionality is then invoked with the error information.

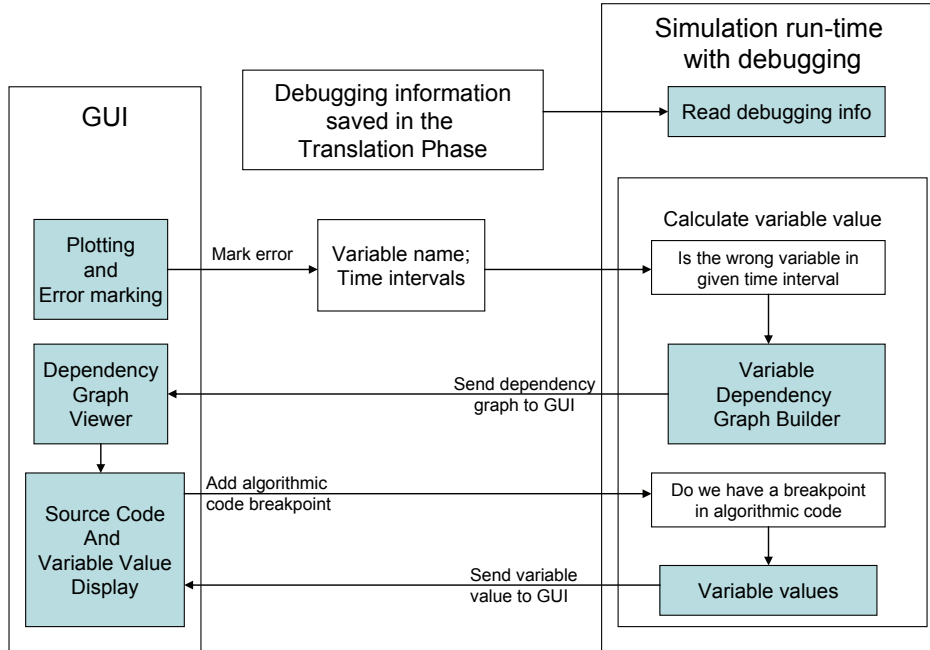


Figure 6-4. Run-time debugging framework overview.

6.4.2 Debugging Framework Overview

The run-time debugging framework overview is presented in Figure 6-4. The figure presents the interaction between the components of the graphical user interface (GUI) and the components of the simulation run-time with debugging. Typically, the user debugging starts at the end of a simulation when the user observes the erroneous behavior of a plotted variable value. The user marks the variable name and the time interval and invokes the debugging functionality. The simulation runtime with debugging is then invoked with the user selection as input.

In the next section we detail the debugging framework components.

6.4.3 Debugging Framework Components

The debugging framework has several components which deal with the user interaction (GUI part) and the handling of the debugging information (simulation runtime part). The information saved during the translation process also plays an important role in the debugging framework.

6.4.3.1 Plotting and Error Marking

This GUI component shows the values of a variable during simulation time. The component has special functionality which helps the user to mark an error on the plot using the mouse. The user markings are encoded as a variable name and time intervals. After marking the error, the user invokes the debugging functionality with this marking.

6.4.3.2 Dependency Graph Viewer

The dependency graph viewer is a GUI component that displays an interactive graph. The graph is given by the dependency graph builder component. The graph shows the calculated variable name and value, the equation in which this value was calculated and all the additional data (parameters, equation blocks, etc) which was used to calculate this value.

In this implementation the user has limited graph interaction possibilities. When the user double clicks on a graph node or edge, the origin of the selected element (variable, equation or parameter) is computed from the debugging information and the Source Code and Variable Value Display component is shown presenting the original source code element.

6.4.3.3 Source Code and Variable Value Display

The source code display is handled by this component. Also, the user can set breakpoints on the algorithmic code within this view. If the runtime reaches a breakpoint, the execution breaks and the variable values from this model can be examined.

6.4.3.4 Dependency Graph Builder

The most complex component of the debugging framework is the dependency graph builder. This component starts from a variable name and builds the dependency graph for that variable based on the debugging information saved in the translation phase.

The constructed graph is based on the Block Lower Triangular Dependency Graph (BLTDG) which is computed from the Block Lower Triangular form by considering the data dependencies. The calculation of the BLTDG is presented in detail in (Bunus 2004 [19]). The constructed graph contains also additional information regarding the origin of each involved element.

6.4.4 Implementation Status

Currently we are working on the integration of the debugging framework components. The debugging framework is developed in Eclipse as a set of plugins that integrate our into our MDT development environment (for code browsing and algorithmic code debugging presented in Chapter 5, Chapter 8 and also in (Pop et

al. 2006 [131])) with graph visualization and interaction libraries. The OpenModelica compiler has been adapted to produce the additional debugging information, the dependency graph and the simulation results.

6.5 Conclusions and Future Work

In this chapter we presented an integrated run-time debugging framework for EOO languages based on graph visualization and interaction. Our method partly follows the approach proposed in (Bunus and Fritzson 2003 [20]). However, our approach does not require the user to write diagnostic specifications of models. The approach we present here can also handle the debugging of algorithmic code using classic debugging techniques (Pop et al. 2006 [131]).

We argue that such debugging framework will ease both the run-time debugging and the understanding of EOO language models.

We are aware that the scalability of our method might be an issue and we plan to research different filtering techniques for pruning the dependency graph.

Our short term goal is to finalize the prototype implementation of the proposed debugging framework, evaluate it and report experience on debugging a set of selected models, and release it as part of the OpenModelica Development Environment.

Chapter 7

Debugging Natural Semantics Specifications

This chapter presents the design, implementation, and usage of a debugging framework for the Relational Meta-Language (RML) which is a language for writing executable Natural Semantics specifications. The language is successfully used at our department for writing large specifications for a range of languages like Java, Modelica, Pascal, MiniML etc. The RML system previously had no debugging facilities, which made it hard for specification writers to debug their specifications. With this work we address these issues by providing a debugging framework for debugging high level Natural Semantics specifications in RML.

The MetaModelica compiler prototype presented in Chapter 5 shares the same compiler and runtime system with the RML system. Thus all the contributions and results presented in this chapter also apply to the MetaModelica compiler prototype. Also, the contributions presented in Chapter 5 also apply to the RML system.

7.1 Introduction

No programming language environment can be considered mature if it is not supported by a strong set of tools which include debugging and profiling. At our department we have developed a language called Relational Meta-Language (RML) (PELAB 1994-2008 [117], Pettersson 1995 [120], Pettersson 1999 [122]) for writing Natural Semantics specifications.

The RML language is extensively used for teaching and writing large specifications for different languages like Java, Modelica, MiniML (Clément et al. 1986 [23]), Pascal, Modelica, etc. Even if the RML language has a short/medium learning curve, the absence of debugging facilities previously created problems of understanding, debugging and verification of large specifications.

To overcome these issues a debugging framework for RML was designed and implemented. The debugger is based on abstract syntax tree instrumentation (program transformation) in the RML compiler and some runtime support. Type reconstruction is performed at runtime in order to present values of the user defined types. For inspecting complex variable values, an external data browser was implemented. Post mortem analysis is possible by recording parts of or the entire specification trace in an XML file, which can be queried using available XML tools (XML (W3C [158]), XQuery (W3C [166]), XPath and XSLT (W3C [159]), etc).

7.2 Related Work

As pointed out in (Liebermann 1997 [85]), the computer science community is constantly ignoring the debugging problem even though the debugging phase of software development takes more than the overall development time. With our work we contribute to improving this state of affairs.

In lazy functional languages like Haskell the execution order is hard to understand. Partly for these reasons the Evaluation Dependence Tree (EDT) tree (Nilsson 1998 [106]) concept was proposed to help the understanding and debugging of the language. On the other hand, RML is a strict functional language where arguments are evaluated before the call and it is, in this respect, closer to Standard ML (Milner et al. 1997 [97]). Our work is related to the work done on the Standard ML debugger (Tolmach and Appel 1995 [154], Tolmach 1992 [155]). We have not yet implemented time traveling, but this is one of our future work directions. General design ideas were inspired from (Pettersson 1998 [121]).

Using assertions and print statements for debugging was and unfortunately still is many programmers choice for debugging programs. Source code instrumentation (or program transformation) that changes the program code in order to facilitate debugging is an approach present in the literature (Fritzson et al. 1994 [48], Pope and Naish 2003 [135]).

Explanation of program execution in deductive systems like Deductive Databases (Mallet and Ducassé 1999 [89]) or Description Logic reasoners (McGuinness 1996 [93], McGuinness and Borgida 1995 [94], McGuinness and Silva 2003 [95]) has similarities with our RML debugger because they generate and analyze proof-trees (or derivation trees). RML is based on the style and visual layout of Natural Semantics and has a top-down left-right determinate search with local backtracking as proof procedure.

7.3 The `rml2c` Compiler and the Runtime System

The `rml2c` compiler is written in Standard ML '97 (Milner et al. 1997 [97]) using the Standard ML of New Jersey (SML/NJ) (SML/NJ-Fellowship 2004-2008 [148]) compiler. The `rml2c` compiler (Figure 7-1) uses several intermediate

representations on which it makes extensive optimizations. The front-end generates ANSI-C code which is linked with the runtime system.

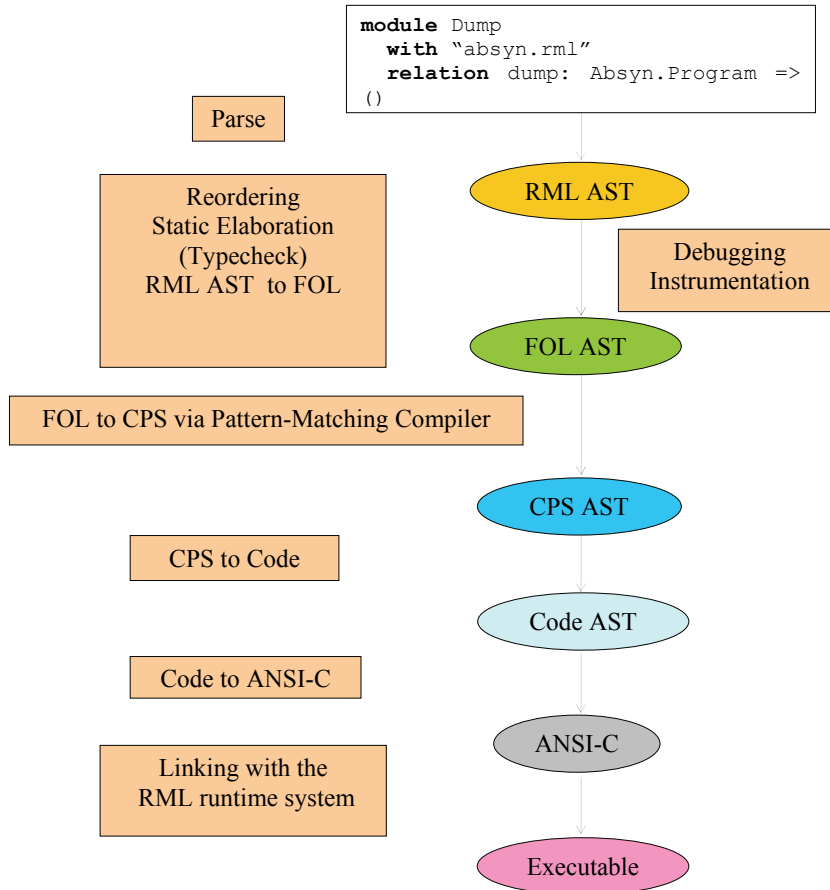


Figure 7-1. The `rml2c` compiler phases.

Immediately after parsing, the specification structure is saved in the RML Abstract Syntax Tree (AST). A reordering phase is performed in order to arrange the declarations in the correct order of dependencies. The static elaboration phase is performing type inference and it checks the program correctness. After the static elaboration phase the current RML AST representation is translated to FOL (a language similar to First Order Logic) representation. On this representation optimizations that improve determinism are applied and the result is translated to CPS (Continuation Passing Style) via a Pattern-Matching Compiler. Optimizations like constant and copy propagation and also inlining are applied to the CPS representation. The CPS representation is translated to a low level imperative representation (Code) that has explicit memory management, data construction and

control flow. In the last phase the Code is translated to ANSI-C. All these phases are depicted in Figure 7-1.

The RML system has two runtime systems: one for fast execution and one for profiling and some logging of the runtime internals.

7.4 Debugger Design and Implementation

The design of the debugger had the following requirements as starting points:

- Conventional debugger functionality (breakpoints, variable value inspections, call chain, stack trace, etc.)
- Inspection/printing of large values.
- Type querying facilities for variables, relations, datatypes.
- Special features for failure discovery. In RML, when a relation fails, the entire specification can also fail. Because of this, is very important to have special functionality for discovering where and under what conditions such failure took place.
- Modular design for easy integration with other tools and graphical user interfaces.
- Reuse of the existing `rm12c` compiler and runtime system.

These requirement specifications were driven by existing tool implementation (the `rm12c` compiler and the runtime system) and easy future extensions and integration. Also, extensive user knowledge and experience about writing RML specifications was used to derive the debugger requirements.

According to the requirements, the only changes of the `rm12c` compiler and runtime system to support debugging were:

- Addition of a new phase that instruments the RML AST with debugging nodes. This phase is triggered from a command line parameter.
- Small changes to the static elaboration phase to output a program database with names and types for all the language identifiers. This program database is used from external tools such as the RML Project Browser and the RML debugging runtime system to query for types of identifiers.
- Addition of a new runtime which has debugging functionality.

The new tools that were developed to aid the debugging task were the RML Data Browser, the Emacs Mode for RML debugging and the Post Mortem Analysis tool.

7.5 Overview of the RML Integrated Environment

The RML integrated environment with debugging and the various interactions between the components are presented in Figure 7-2.

In the following we only describe the use of the toolbox with regards to debugging. The RML Project Browser is a navigator for RML specifications that ease the browsing of relations and datatypes.

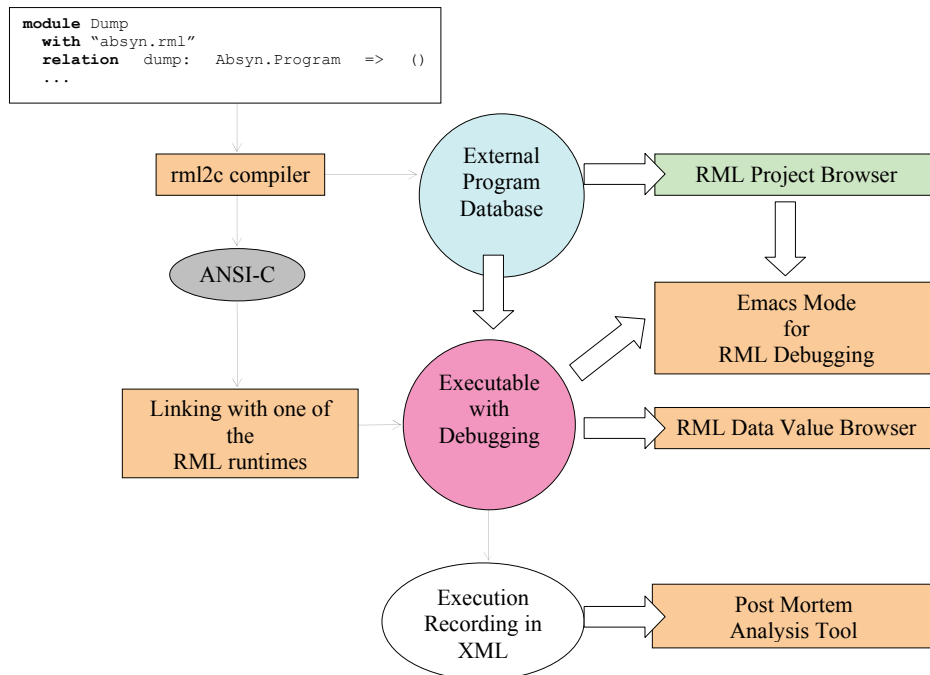


Figure 7-2. Tool coupling within the RML integrated environment with debugging.

The `rml2c` compiler takes as input an RML specification. The specification is instrumented with debug nodes. Then, the normal compilation phases are applied until C code is generated. The generated C code is compiled and linked with the debugging runtime system. Also, the compiler dumps the program database at the end of static elaboration phase, after performing type inference.

When started, the executable reads in the program database and waits for user commands. This is a good time to set breakpoints using commands or helpers from Emacs Mode for RML Debugging. Then the execution can be resumed. At breakpoints one can print variable values directly in the standard output or they can be sent to the RML Data Value Browser for thorough inspection.

User commands are available in the debugger for recording of the execution as an XML trace. The XML trace can be analyzed post-mortem using XML tools. In this way, when a certain relation fails and generates the failure of the entire

specification, one can understand when and why that happens by a post-mortem analysis of the execution trace.

7.6 Design Decisions

This section discusses the design decisions that were taken in the design process of our debugging tools.

7.6.1 Debugging Instrumentation

The RML compiler has several intermediate representations on which aggressive optimizations are applied. Because of this, debugging approaches that keep a mapping between intermediate representations and store reverse transformations of optimizations were out of the question. The best available approach was to apply debugging instrumentation at the RML AST level.

7.6.2 External Program Database

In order to present variable values using user-defined data structure one has to do type reconstruction at runtime. There were two possibilities of keeping a program database with the defined relations, variables, types and datatypes:

- Storing the needed information obtained after type inference in SML data structures and generating C code with this information in the Code to C phase of the compilation.
- Exporting the needed information to external files which can be read later by the runtime system.

We choose the second alternative because this kind of information is also useful in powerful RML IDE (which includes the RML Project Browser) that provides code assistance (IntelliSense), displaying of types when hovering over variables and relations, pattern writing wizards, project browser, etc. We have already developed such an IDE for RML (Pop and Fritzson 2006 [129]).

7.6.3 External Data Value Browser

After implementing the printing of variable values to standard output it soon became apparent that for large values such displaying is unreadable. As an alternative we have implemented a very simple but practical value browser prototype. One nice feature: the browser provides immediate information about where in the specification code each part of the data structure was defined. Future

work on this prototype could provide new functionality i.e., for searching, and analyses of the variables.

7.6.4 Why not an Interpreter?

Interpreters are good when one wants hands-on development with fast feedback. However, they are quite slow, because optimizations cannot be applied if one wants to give a clear feedback to the user. Also, we already had the compiler. Fast feedback to the developer can also be achieved by incremental compilation techniques, which is an approach we are currently working on.

7.7 Instrumentation Function

In this section we define the transformations that are performed by the instrumentation function over the RML AST. The instrumentation function is simple but very effective. In order to define this function we need to explain in more detail the parts of the RML language. The specification of RML is presented in (PELAB 1994-2008 [117], Pettersson 1995 [120], Pettersson 1999 [122]).

RML modules have two parts: the interface specification (which defines the signatures that are to be exported from the current module) and the actual declaration of relations, private module types, datatypes, relations, and global values. Clauses (rules and axioms) can be grouped together in relations. Rules have three parts: the matching pattern, premises, and results. Axioms are just rules without premises.

Premises (also called goals) can be of the following types:

Bindings	<code>let pat = exp</code>
Unification	<code>var = exp</code>
Relation calls	<code>longIdentifier(expseq) => patseq</code>
Negation	<code>not premise</code>
Sequence	<code>premise & premise</code>

Table 7-1. RML premise types. These constructs are swept for variables to be registered with the debugging runtime system.

Clauses (rules and axioms) have the following form:

```
rule <premise>
-----
var(pat) => result
```

```
axiom var(pat) => result
```

Premises can be optional in rules or a sequence of premises. Axioms are just rules without premises.

The debugging instrumentation `Instr` function transforms only premises in the following way:

```
Instr(premise) =  
  RML.register_in(parameters) &  
  RML.debug(...) & premise &  
  RML.register_out(results)
```

For a sequence of premises the result variables from the last executed premise, together with the parameter of the next premise, are registered with the debugging framework. Then the debug function `RML.debug(...)` checks for breakpoints, user commands or single-stepping. The debug function has as parameters the source filename, the line/column number of the premise, and the premise textual representation.

As one can see, for each premise a sequence of three premises are generated. We could have got the live variables for a premise from the runtime system, but we use instead call premises that register these in/out variables. We used this approach because in the runtime system some variables are not present due to optimizations and also a mapping should have been kept that map existing source code variable names to positional parameters of relations. The parameters of variable registration functions are built by sweeping the premises for variables that appear in expressions or patterns.

7.8 Type Reconstruction in the Runtime System

The debugging runtime system loads the program database files at startup and stores them in some internal structures. When the program is executed in the `RML.debug(...)` function the filename and the line/column position of the current execution point are known. With this knowledge and the name of the variable to be printed the program database information is searched for a rule that frames this point and contains the variable. The variable type is then retrieved.

The variable values are stored in the RML runtime heap as tagged pointers or immediate values. Immediate values are only integers. All other values are boxed and tagged. The tags contain information about the structure and elements of the values.

Starting from the variable type and the variable pointer which was registered using the `register_in/register_out` functions the variable value is traversed. At the same time the variable type is unfolded and the new type components are mapped to the current variable components.

7.9 Debugger Implementation

The implementation of the debugger follows the proposed design closely.

7.9.1 The `rml2c` Compiler Addition

In the `rml2c` compiler we implemented the instrumentation phase as a separate Standard ML module that has as input the RML AST and as output the transformed RML AST with the debug nodes added. This additional phase is triggered by a command line parameter to the `rml2c` compiler. Also, the instrumentation can be applied selectively module or relation-wise in order to instrument only the problematic parts of the specification and achieve a faster debugging execution.

In the static elaboration phase, after type inference is performed we saved the type information (that was normally discarded) in an identifier dictionary based on balanced search trees. At the end of the phase we write this information to the program database file in a flat format composed of: the identifier type, the file where it appears, the identifier, the line/column number and its type. A small portion of the program database file for our `exp1.rml` example specification is presented in the appendix.

7.9.2 The Debugging Runtime System

All the low-level runtime debugger functionality is implemented in C. The user commands are read by a command parser and the program database is read using another parser. The parsers are implemented using Flex (Lex) (GNU 2005 [58]) and Bison (Yacc) (GNU 2005 [56]) and the readline library (GNU 2005 [60]) (for history, command input handling, etc).

The program database is read and stored internally in the runtime as a list. An ordering phase is then performed to have the information indexed over module name (filename) and line number.

The `RML.debug(...)` relation is implemented also in C and uses the RML foreign function interface. The relation checks if a breakpoint was reached and in that case stops the execution, prints the next premise to be executed and waits for user commands. The relations `RML.register_in("var_name", var, ...)` and `RML.register_out("var_name", var, ...)` save the live variable information in internal arrays as (variable name, pointer to variable value) pairs. Only registered variables can be printed or sent to the external variable value browser.

The printing or sending of the variable values is realized by recursive functions that traverse both the value structure and the value type at the same time. The type of a certain variable is retrieved from the program database information by matching the file, the name of the variable, and the positional frame of the rule.

These traversing and displaying functions take into consideration the printing depth, which is a debugger setting and can be changed using commands. Sockets are used when variable values are sent to the external browser.

7.9.3 The Data Value Browser

The browser is implemented in Java to have the same high portability as the RML system. The browser waits to read variable value information from sockets and displays them in a tree structure constructed by using the traversal depth.

Syntax highlighting of RML files is performed by the browser, using a similar Emacs RML Mode style to keep the users on familiar grounds.

7.9.4 The Post-Mortem Analysis Tool

In this tool, at the moment we have only implemented a Failure analyzer that helps users understand where and why their specification failed. The analyzer is implemented in Java and replays the specification execution by navigation in the saved XML trace. One can stop, go back and forward in time, display variable values, etc. In general users start from the end of the execution and go back to where their specification failed.

The trace files can be quite large, on the order of several hundred megabytes. To overcome this problem we gave the users the possibility to configure the tracer using a small specification file that contains:

- Module, relation and/or rule to be traced.
- Selection of variable names to include only their value in the trace.

This file is read by the tracer function and all the information is filtered accordingly.

We plan to implement more analyses and automated debugging in the future. Also, tuning of the specification data structures and its operational properties could be suggested by trace analysis.

7.10 Debugger Functionality

The Emacs RML debug mode is implemented as a specialization of the Grand Unified Debugger (GUD) interface (`gud-mode`) from Emacs (GNU 2005 [57]). Because the RML debug mode is based on the GUD interface, some of the commands have the same familiar key bindings. The actual commands sent to the debugger are also presented together with GUD commands preceded by the RML debugger prompt: `rmlldb@>`.

If the debugger commands have several alternatives these are presented using the notation: `alt1|alt2`. The optional command components are presented using notation: `[optional]`.

In the Emacs interface: `M-x` stands for holding down the `Meta` key (mapped to `Alt` in general) and pressing the key after the dash, here `x`, `C-x` stands for holding down the `Control` (`Ctrl`) key and pressing `x`, `<RET>` is equivalent with pressing the `Enter` key and `<SPC>` with pressing `Space` key.

The next subsections present a debugging session on the RML example specification for the `Exp1` language presented in section 2.5.1.

7.10.1 Starting the RML Debugging Subprocess

The command for starting the RML debugger under Emacs:

```
M-x rmlldb <RET> executable <RET>
```

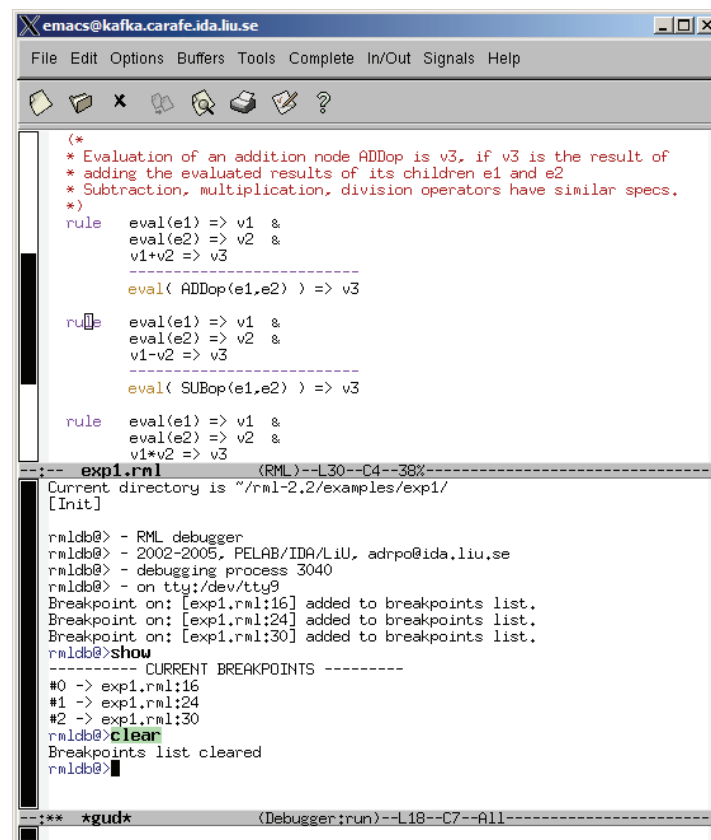


Figure 7-3. Using breakpoints.

7.10.2 Setting/Deleting Breakpoints

A part of a session using this type of commands is shown in Figure 7-3. The presentation of the commands follows.

To set a breakpoint on the line the cursor (point) is at:

```
C-x <SPC>
rmlldb@> break on file:lineno|string <RET>
```

To delete a breakpoint placed on the current source code line (gud-remove):

```
C-c C-d
C-x C-a C-d
rmlldb@> break off file:lineno|string <RET>
```

Instead of writing `break` one can use alternatives `br|break|breakpoint`. Alternatively one can delete/display all breakpoints using:

```
rmlldb@> clear <RET>
rmlldb@> show <RET>
```

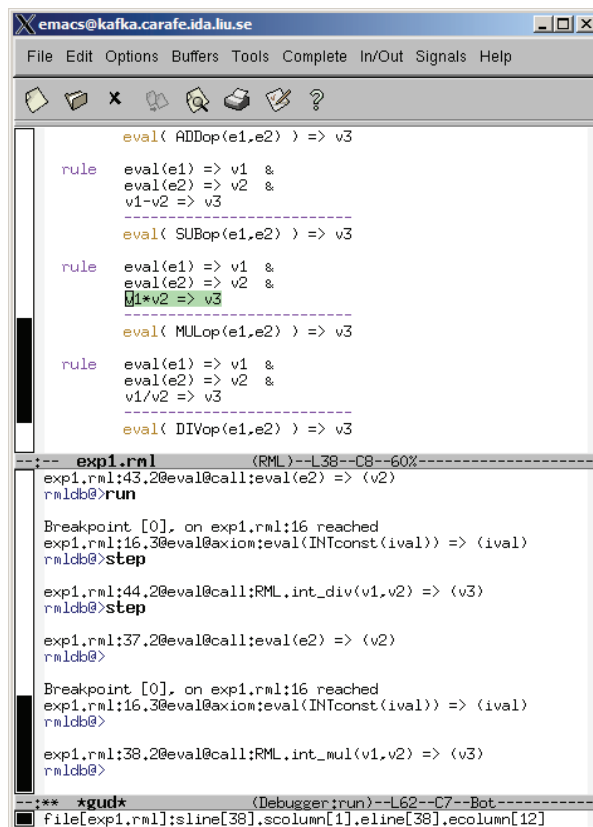


Figure 7-4. Stepping and running.

7.10.3 Stepping and Running

To perform one step (`gud-step`) in the RML code:

```
C-c C-s
C-x C-a C-s
rmlldb@> step <RET>
rmlldb@> <RET>
```

To continue after a step or a breakpoint (`gud-cont`):

```
C-c C-r
C-x C-a C-r
rmlldb@> run <RET>
```

Examples of using these commands are presented in Figure 7-4.

7.10.4 Examining Data

There are no GUD key bindings for these commands but they are inspired from the GNU Project debugger (GDB) (GNU 2005 [59]).

To print the contents/size of a variable one can write:

```
rmlldb@> print variable_name <RET>
rmlldb@> sizeof variable_name <RET>
```

at the debugger prompt. The size is displayed in bytes.

Variable values to be printed can be of a complex type and very large. One can restrict the depth of printing using:

```
rmlldb@> [set] depth integer <RET>
```

Moreover, we have implemented an external data value browser written in Java called `RMLDataViewer` to browse the contents of such a large variable. To send the contents of a variable to the external viewer for inspection one can use the command:

```
rmlldb@> browse|graph var_name <RET>
```

at the debugger prompt. The debugger will try to connect to the `RMLDataViewer` and send the contents of the variable. The external data browser has to be started a priori. If the debugger cannot connect to the external viewer within a specified timeout a warning message will be displayed. More about the external `RMLDataViewer` tool can be found in section 7.11.

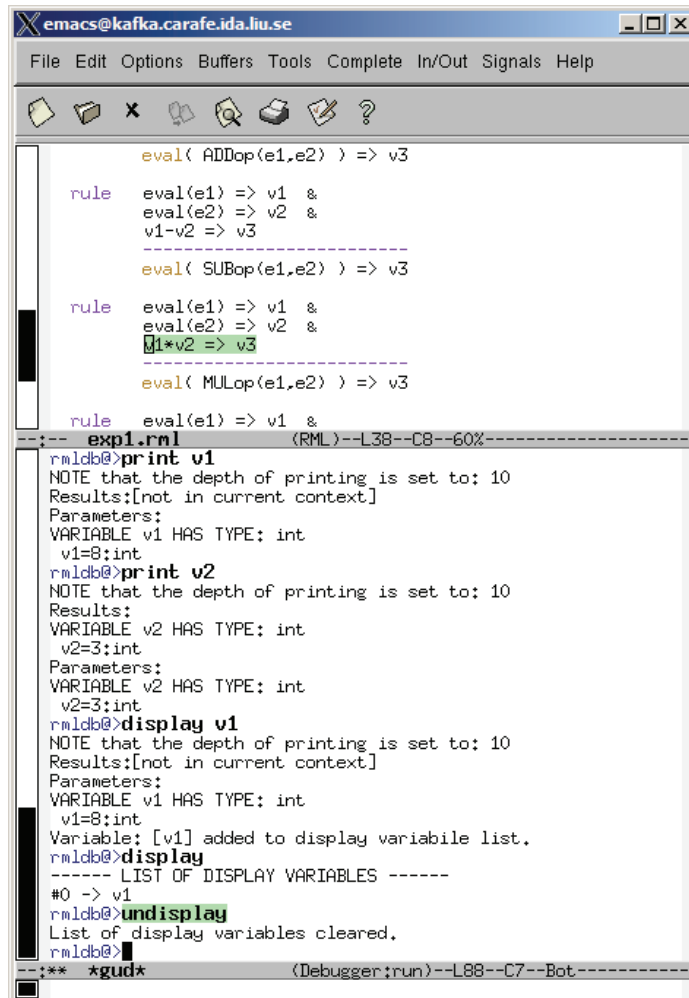


Figure 7-5. Examining data.

If the variable which one tries to print does not exist in the current scope, a notifying warning message will be displayed.

Automatic printing of variables at every step or breakpoint can be specified by adding a variable to a display list:

```
rmlldb> display variable_name <RET>
```

Removing a display variable from the display list:

```
rmlldb> undisplay variable_name <RET>
```

To print the entire display list or to remove all variables from it:

```
rmlldb> display <RET>
rmlldb> undisplay <RET>
```

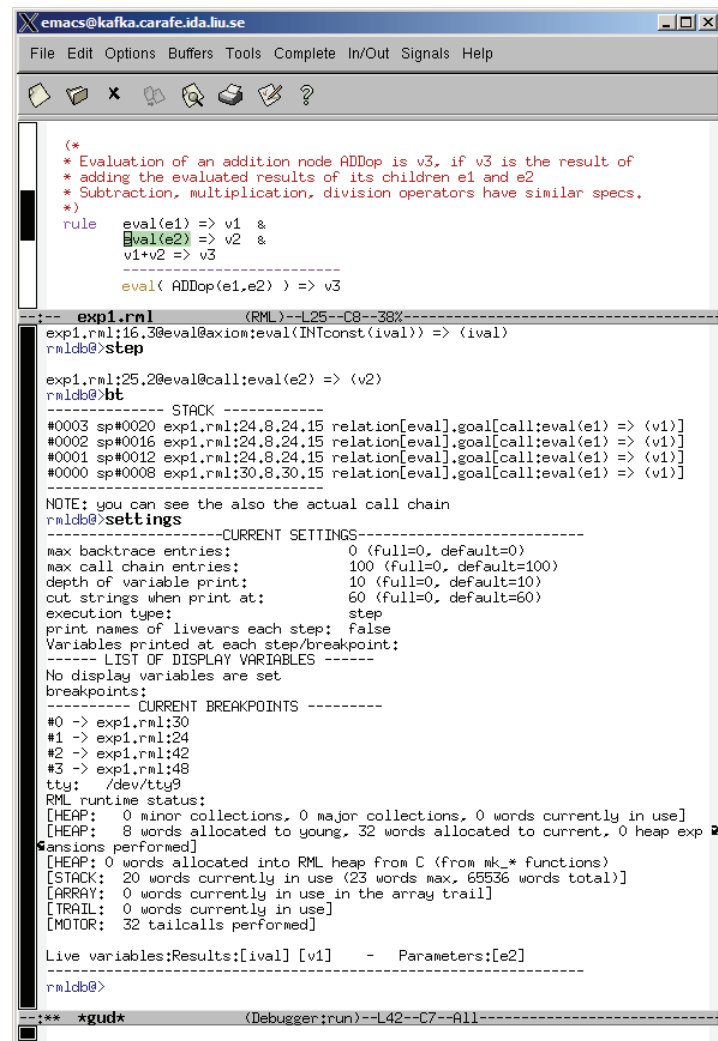

Printing the current live variables (variables available in the scope):

```
rmldb@> livevars <RET>
```

Instructing the debugger to print or to disable the print of the live variable names at each step/breakpoint:

```
rmldb@> [set] livevars on|off|<RET>
```

Figure 7-5 shows examples of some of these commands within a debugging session.



```

emacs@kafka.carafe.ida.liu.se
File Edit Options Buffers Tools Complete In/Out Signals Help

(*
 * Evaluation of an addition node ADDop is v3, if v3 is the result of
 * adding the evaluated results of its children e1 and e2
 * Subtraction, multiplication, division operators have similar specs.
 *)
rule
  eval(e1) => v1 &
  eval(e2) => v2 &
  v1+v2 => v3
  -----
  eval( ADDop(e1,e2) ) => v3

--:-- exp1.rml (RML)--L25--C8--38%-----
exp1.rml:16,3@eval@axiom:eval(INTconst(ival)) => (ival)
rmldb@>step
exp1.rml:25,2@eval@call:eval(e2) => (v2)
rmldb@>bt
----- STACK -----
#0003 sp#0020 exp1.rml:24,8,24,15 relation[eval].goal[call:eval(e1) => (v1)]
#0002 sp#0016 exp1.rml:24,8,24,15 relation[eval].goal[call:eval(e1) => (v1)]
#0001 sp#0012 exp1.rml:24,8,24,15 relation[eval].goal[call:eval(e1) => (v1)]
#0000 sp#0008 exp1.rml:30,8,30,15 relation[eval].goal[call:eval(e1) => (v1)]
NOTE: you can see the also the actual call chain
rmldb@>settings
-----CURRENT SETTINGS-----
max backtrace entries: 0 (Full=0, default=0)
max call chain entries: 100 (Full=0, default=100)
depth of variable print: 10 (Full=0, default=10)
cut strings when print at: 60 (Full=0, default=60)
execution type: step
print names of livevars each step: false
Variables printed at each step/breakpoint:
----- LIST OF DISPLAY VARIABLES -----
No display variables are set
breakpoints:
----- CURRENT BREAKPOINTS -----
#0 -> exp1.rml:30
#1 -> exp1.rml:24
#2 -> exp1.rml:42
#3 -> exp1.rml:48
tty: /dev/tty9
RML runtime status:
[HEAP: 0 minor collections, 0 major collections, 0 words currently in use]
[HEAP: 8 words allocated to young, 32 words allocated to current, 0 heap exp
ansions performed]
[HEAP: 0 words allocated into RML heap from C (from mk_* functions)
[STACK: 20 words currently in use (23 words max, 65536 words total)]
[ARRAY: 0 words currently in use in the array trail]
[TRAIL: 0 words currently in use]
[MOTOR: 32 tailcalls performed]

Live variables:Results:[ival] [v1] - Parameters:[e2]
rmldb@>

--:** *gud* (Debugger:run)--L42--C7--A11-----

```

Figure 7-6. Additional debugging commands.

7.10.5 Additional Commands

Additional commands provide functionality for displaying the call chain, the stack contents, the runtime status, etc. A session using some of these commands is presented in Figure 7-6.

The stack trace can be displayed using:

```
rmldb@> backtrace <RET>
```

Because the contents of the stack can be quite large, one can print a filtered view of it:

```
rmldb@> fbacktrace filter_string <RET>
```

Also, one can restrict the numbers of entries the debugger is storing using:

```
rmldb@> maxbacktrace integer <RET>
```

Also, the call chain is available in the debugger. Similar commands as for the backtrace are available for call chain trace.

For displaying the status of the RML runtime:

```
rmldb@> status <RET>
```

The status of the RML runtime comprises information regarding the garbage collector, allocated memory, stack usage, etc.

The current debugging settings can be displayed using:

```
rmldb@> settings <RET>
```

The settings printed are, i.e., the maximum remembered stack entries, the depth of variable printing, the current breakpoints, the live variables, the list of the display variables and the status of the runtime system.

One can invoke the debugging help or exit the debugger by issuing:

```
rmldb@> help <RET>  
rmldb@> quit <RET>
```

7.11 The Data Value Browser

The RMLDataViewer is a browser for variable values and a new addition to our debugging tools for RML. The need for such a tool became apparent when debugging specifications that use very large data structures (for example abstract syntax tree definitions for a certain language).

From the executable, at the debugging prompt one can invoke a browse command which sends the queried variable value for displaying in the external browser. The variable values can be limited in depth using set depth command. In this way only needed parts of the variable value are sent.

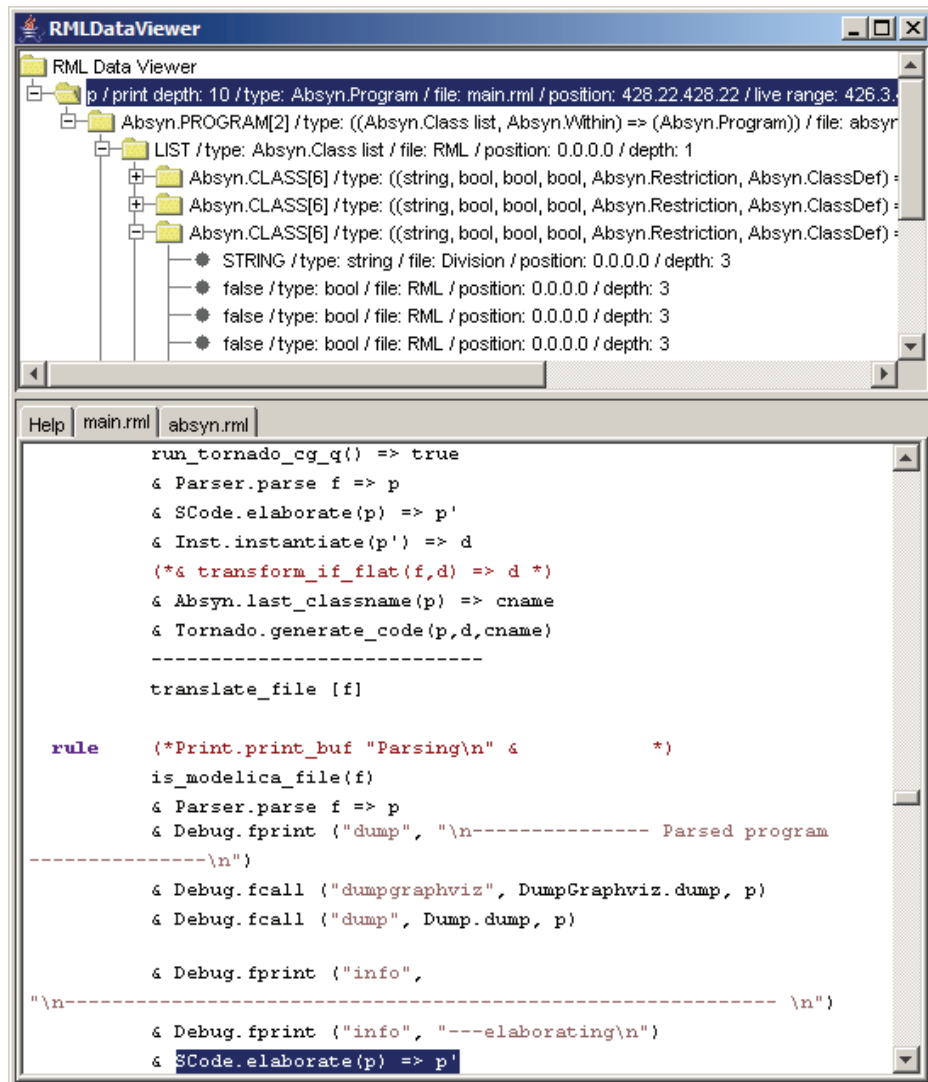


Figure 7-7. Browser for variable values showing the current execution point (bottom) and the variable value (top).

The variable values are displayed in the browser as trees. The trees are collapsed, but one can expand them further until the needed information is found. The children of the root are the browsed variable names. When users click on the variable names the bottom part of the browser shows (using tabs) the file where the execution point is/was when the variable was sent to the browser. This functionality is presented in Figure 7-7. To make it easy for users to understand their variables, the browser shows datatype definitions connections to pieces of variable values like in Figure 7-8.

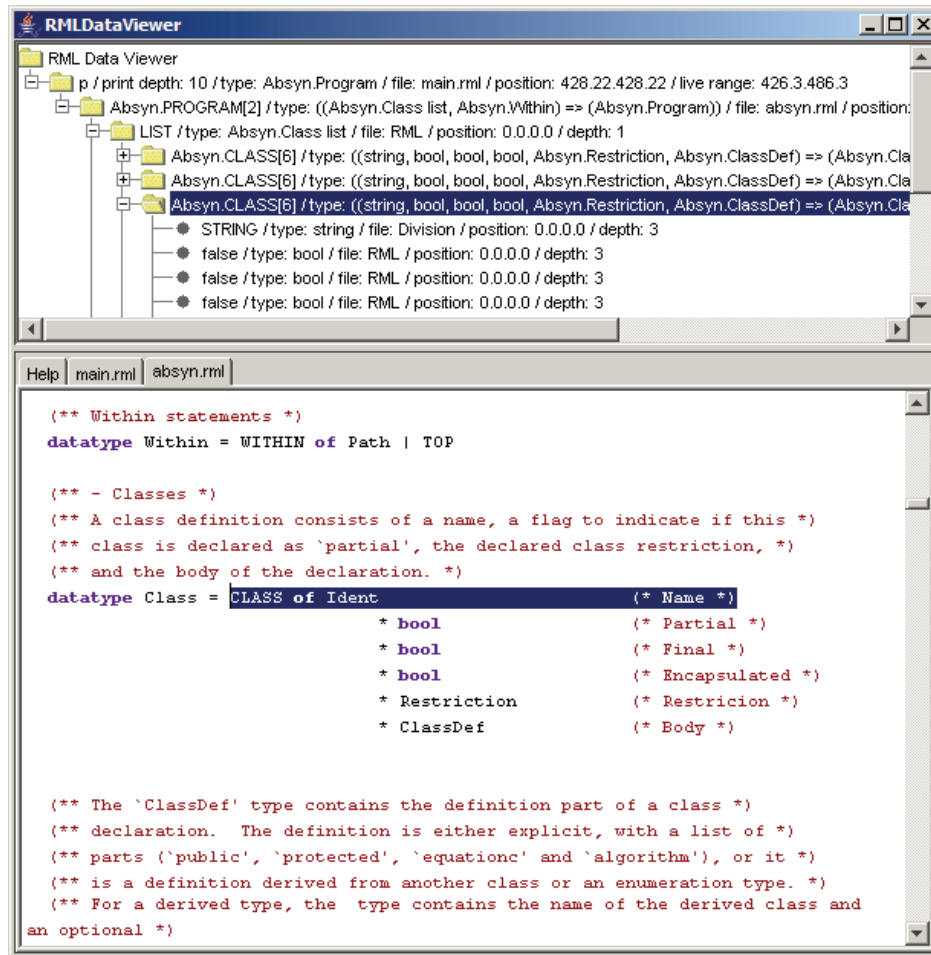


Figure 7-8. When datatype constructors are selected, the bottom part presents their source code definitions for easy understanding of the displayed values.

The screens were captured while debugging the OpenModelica compiler specification and the variable value consists of the abstract syntax tree of the Modelica language.

7.12 The Post-Mortem Analysis Tool

As pointed out in the debugger design and implementation, one can record parts of or the entire execution trace of the specification in an XML file. The trace can then be analyzed by tools that point out specific issues.

In our post-mortem analysis environment we have developed a tool called Failure analyzer. The Failure analyzer is a replay debugger which is able to walk back and forth in time, display variable values, execution points, etc. When their specification fails the users can run this analyzer over the recorded trace, start from the end of the execution and go back and investigate where the execution has failed and why. This tool was very important for our users, because, for large specifications, is not trivial to understand where and why your specification failed.

The Failure analyzer tool is similar to the data value browser, but has buttons for navigation in time, setting/deleting breakpoints and displaying values.

7.13 Performance Evaluation

In this section we make performance evaluation of our debugging strategy on three real-world semantic specifications that define compilers for extended Pascal (petrol), a small functional language (MiniML (Clément et al. 1986 [23])) and a large Modelica compiler (OpenModelica). The first two specifications are part of the examples bundled with the RML system (PELAB 1994-2008 [117], Pettersson 1995 [120], Pettersson 1999 [122]) and the Modelica compiler was implemented in the OpenModelica project and is also available for download at the project address. The semantic specifications were compiled to two versions of executables, one in release mode and one in debugging mode. The compilers were then used to compile programs and the compilation performance was measured.

We have tested the performance of our debugger on an Intel Pentium Mobile at 1.5Ghz with 480 MB of RAM memory. We compared code growth, execution time, stack consumption, and number of relation calls.

If we consider that a premise (one call) is executed in $O(1)$ then the complexity of the call combined with the instrumentation will be $O(\text{number of variables from the premise}) + O(\text{premise}) + O(\text{call to the step function})$ which is a complexity in the order of the numbers of variables present in the specification.

7.13.1 Code Growth

Table 7-2 below shows the additional number of lines of code added during code instrumentation. The code growth is between 1.3 and 1.7 which is quite limited. We can see that for very large specifications like the OpenModelica compiler the code grows less than for smaller specifications. The code growth was measured on the files obtained from the abstract syntax tree unparsing before and after the instrumentation. The comments were ignored.

test/mode (debug/normal)	normal	debug
petrol (1.63)	2513	4083

miniml (1.57)	1112	1747
OpenModelica (1.36)	57186	77961

Table 7-2. Size (#lines) without and with instrumentation.

7.13.2 The Execution Time

The execution time was also measured and the results are presented below.

test/mode (debug/normal)	normal (seconds)	debug (seconds)
petrol (24.63)	0.12	2.96
Miniml (11.19)	6.14	68.71
OpenModelica (20.55)	0.20	4.11

Table 7-3. Running time without and with debugging.

Table 7-3 presents a performance evaluation of our debugger. As one can notice, the programs compiled in debug mode are between 10 and 25 times slower than the programs compiled without debugging. We find this acceptable, as this is the first prototype. For the user, the delay times due to the added debugging code are practical. We can note also that very large specifications can be debugged without too much penalty.

7.13.3 Stack Consumption

We have investigated the stack consumption needed during debugging versus the normal memory consumption. The results are summarized in Table 7-4.

test/mode (debug/normal)	normal (words)	debug (words)
petrol (1.19)	249	297
miniml (1.01)	8966	9126
OpenModelica (1.06)	1447	1543

Table 7-4. Used stack without and with debugging.

It is normal that the debugging version of the runtime needs more stack because it has more calls. This can be seen in the next subsection in Table 7-5. However, one can see that the stack growth due to debugging is small, which means that the high level optimization (that improve determinism) in the `rml2c` compiler are very effective.

7.13.4 Number of Relation Calls

Presented in Table 7-5 is the total number of relations called during execution. Here one can see that the debugger is using a large number of calls to register variables and to check breakpoints or steps.

test/mode (debug/normal)	normal	debug
petrol (6.30)	350305	2209984
miniml (16.30)	2809705	45805284
OpenModelica (5.30)	510321	2706378

Table 7-5. Number of performed relation calls.

7.14 Conclusions and Future Work

In this chapter we have presented our practical debugging framework for Natural Semantics. The debugging design, implementation and usage (functionality) was detailed.

We can report that some of our RML users who have debugged their specifications using this debugging framework have given us positive feedback and also various suggestions for improvement.

While this is a good start, many improvements can be made to this framework. As future direction we plan to improve the debugger execution speed, implement time traveling without the need of execution tracing, define more post-mortem analyses. One of our goals is to integrate of all our tools in an integrated development environment (IDE) for RML based on the Eclipse platform (Eclipse.Foundation 2001-2008 [29]). We already designed and implemented such an RML IDE (Pop and Fritzson 2006 [129]).

Part IV

Advanced Integrated Environments

Chapter 8

Modelica Development Tooling (MDT)

The OpenModelica (MDT) Eclipse Plugin integrates the OpenModelica compiler and debugger with the Eclipse Integrated Development Environment Framework.. MDT, together with the OpenModelica compiler and debugger, provides an environment for Modelica development projects. This includes browsing, code completion through menus or popups, automatic indentation even of syntactically incorrect models, and model debugging. Simulation and plotting is also possible from a special command window. To our knowledge, this is the first Eclipse plugin for an equation-based language. Eclipse (Eclipse.Foundation 2001-2008 [29]) is an open source framework for creating extensible integrated development environments (IDEs) using plugins.

8.1 Introduction

The goal of our work with the Eclipse framework integration in the OpenModelica modeling and development environment is to achieve a more comprehensive and powerful environment. It can be useful to first take a general look at this area including some background.

8.1.1 Integrated Interactive Programming Environments

An integrated interactive modeling and simulation environment is a special case of programming environment aimed at applications in modeling and simulation. Thus, it should fulfill the requirements both from general integrated environments and from the application area of modeling and simulation mentioned in the thesis.

The main idea of an integrated programming environment in general is that a number of programming support functions should be available within the same tool in a well-integrated way. This means that the functions should operate on the same data and program representations, exchange information when necessary, resulting in an environment that is both powerful and easy to use. An environment is interactive and incremental if it gives quick feedback, e.g. without recomputing

everything from scratch, and maintains a dialogue with the user, including preserving the state of previous interactions with the user. Interactive environments are typically both more productive and more fun to use.

There are many things that one wants a programming environment to do for the programmer, particularly if it is interactive. What functionality should be included? Comprehensive software development environments are expected to provide support for the major development phases, such as:

- Requirements analysis.
- Design.
- Implementation.
- Maintenance.

A programming environment can be somewhat more restrictive and need not necessarily support early phases such as requirements analysis, but it is an advantage if such facilities are also included. The main point is to provide as much computer support as possible for different aspects of software development, to free the developer from mundane tasks so that more time and effort can be spent on the essential issues. The following is a partial list of integrated programming environment facilities, some of which were already mentioned in (Sandewall 1978), that should be provided for the programmer:

- Administration and configuration management of program modules and classes, and different versions of these.
- Administration and maintenance of test examples and their correct results.
- Administration and maintenance of formal or informal documentation of program parts, and automatic generation of documentation from programs.
- Support for a given programming methodology, e.g. top-down or bottom-up. For example, if a top-down approach should be encouraged, it is natural for the interactive environment to maintain successive composition steps and mutual references between those.
- Support for the interactive session. For example, previous interactions should be saved in an appropriate way so that the user can refer to previous commands or results, go back and edit those, and possibly re-execute.
- Enhanced editing support, performed by an editor that knows about the syntactic structure of the language. It is an advantage if the system allows editing of the program in different views. For example, editing of the overall system structure can be done in the graphical view, whereas editing of detailed properties can be done in the textual view.
- Cross-referencing and query facilities, to help the user understand interdependences between parts of large systems.
- Flexibility and extensibility, e.g. mechanisms to extend the syntax and semantics of the programming language representation and the functionality built into the environment.

- Accessible internal representation of programs. This is often a prerequisite to the extensibility requirement. An accessible internal representation means that there is a well-defined representation of programs that are represented in data structures of the programming language itself, so that user-written programs may inspect the structure and generate new programs. This property is also known as the principle of program-data equivalence.

Early work in interactive integrated programming environments supporting a specific language was done in the InterLisp system for the Lisp language: (Teitelman 1974), common principles and experience of early interactive Lisp environments are described in (Sandewall 1978), interactive and incremental Pascal with the DICE system: (Fritzson 1983), the integrated Mjölner environment, (Lindskov, Knudsen, Lehrmann-Madsen, and Magnusson 1993).

8.1.2 The Eclipse Framework

Eclipse (Eclipse.Foundation 2001-2008 [29]) is an open source framework for creating extensible integrated development environments (IDEs). One of the goals of the Eclipse platform is to avoid duplicating common code that is needed to implement a powerful integrated environment for development of software. By allowing third parties to easily extend the platform via the plugin concept, the amount of new code that needs to be written is decreased.

8.1.3 Eclipse Platform Architecture

By itself, Eclipse does not provide extensive end-user functionality. The important contribution of Eclipse is based on its plugins. The smallest architectural unit of the Eclipse platform is the plugin.

At the core of Eclipse is the Eclipse Platform Runtime. The Runtime in itself mostly provides the loading of external plugins. The Java Development Tooling (JDT) is for example a collection of plugins that are loaded into Eclipse when they are requested. The fact that Eclipse is in itself written in Java and comes with the Java Development Tooling as default often leads newcomers to believe that Eclipse is a Java IDE with plugin capabilities. It is in fact the other way around, with Eclipse being just a base for plugins, and the Java Development Tooling plugging into this base.

To extend Eclipse, a set of new plugins must be created. A plugin is created by extending a certain extension point in Eclipse. There are several predefined extension points in Eclipse, and plugins can provide their own extension points. This means that you can plug in plugins into other plugins.

An extension point can have several plugins attached, and what plugin will be used is determined by a property file. For example, the Modelica Editor is loaded at the same time as the Java Editor is loaded. When a user opens a Java file, the Java

Editor will be used, based on a property in the Java Editor extension. In this case, it is the file name extension that determines what editor that should be used.

As the number of plugins in Eclipse can be very large, a plugin is not actually loaded into memory before its contribution is directly requested by the user. This design makes the memory impact reasonably low while running Eclipse.

A user-friendly aspect of Eclipse is the Eclipse Update Manager which allows you to install new plugins just by pointing Eclipse to a certain website. This website is provided by the developers of the plugin that you may wish to install. An update site at the OpenModelica web site is for example provided for easy installation of the latest version of MDT.

8.1.4 OpenModelica MDT Eclipse Plugin

The MDT Eclipse plugin provides file and class hierarchy browsing and text editing capabilities. Some syntax highlighting facilities and a compilation manager are also included in MDT, as well as integration to the debugger.

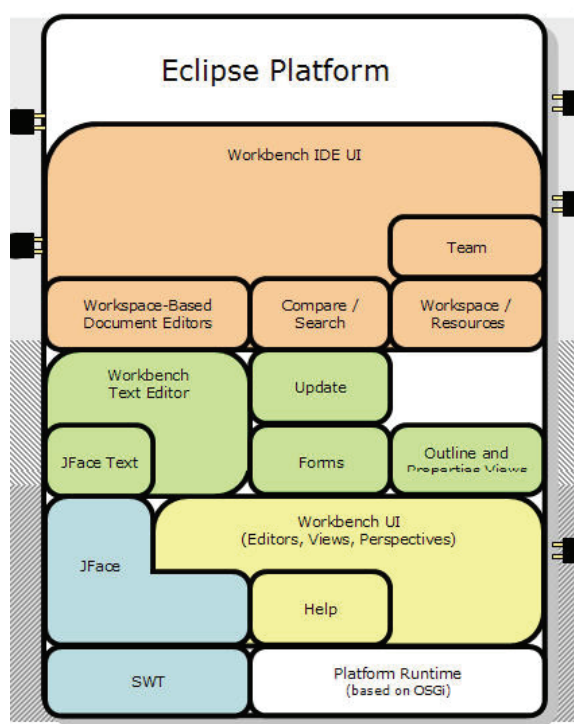


Figure 8-1. The architecture of Eclipse, with possible plugin positions marked.

The Eclipse framework (Figure 8-1) has the advantage of making it easy to add future extensions.

8.2 OpenModelica Environment Architecture

The MDT Eclipse plugin is integrated in the OpenModelica environment which consists of several interconnected subsystems, as depicted in Figure 8-2 below.

Arrows denote data and control flow. Several subsystems provide different forms of browsing and textual editing of Modelica code.

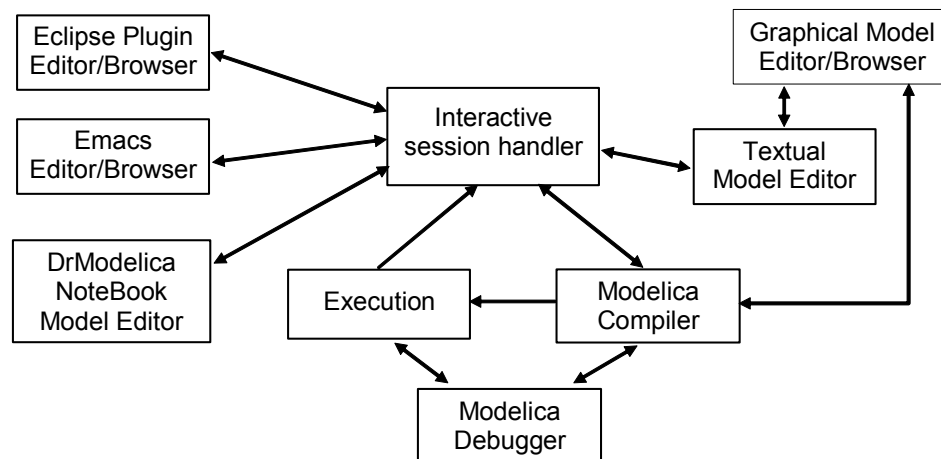


Figure 8-2. The architecture of the OpenModelica environment.

OpenModelica is structured as several communicating processes in a client-server architecture, primarily exchanging information through a Corba interface, see Figure 8-3. The OpenModelica compiler/interpreter (OMC) is the server, communicating with clients. The Eclipse MDT plugin is one of the clients.

Messages from the Corba interface are of two kinds. The first group consists of expressions or user commands which are evaluated by the `Ceval` package. The second group consists of declarations of classes, variables, etc., assignments, and client-server API calls that are handled via the `Interactive` package, which also stores information about interactively declared/assigned items at the top-level in an environment structure.

A more detailed description of the OpenModelica compiler (OMC) is given in section 4.3 of Chapter 4 where the important packages of the compiler are described.

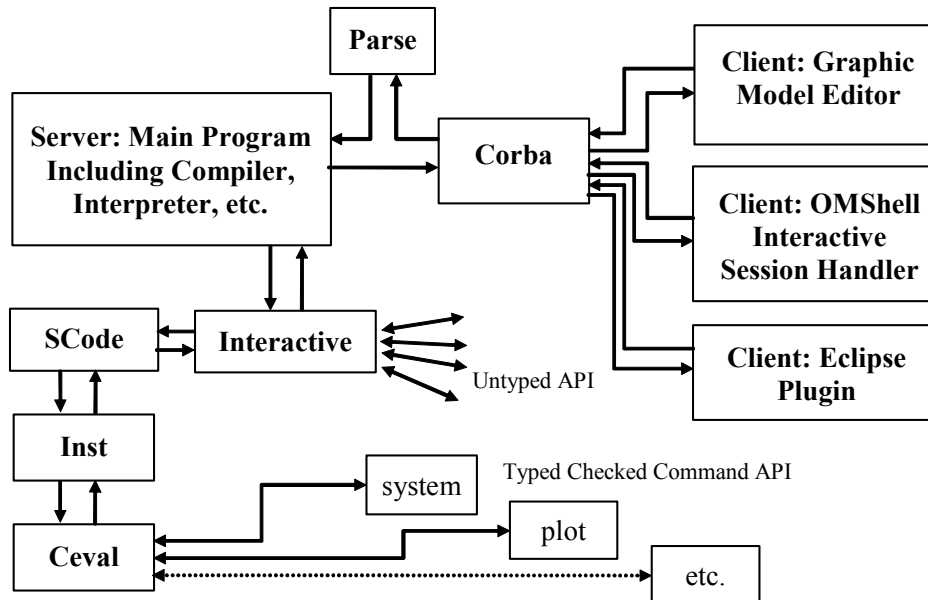


Figure 8-3. The client-server architecture of the OpenModelica environment.

8.3 Modelica Development Tooling (MDT) Eclipse Plugin

As mentioned, the Modelica Development Tooling (MDT) Eclipse Plugin provides an environment for working with Modelica development projects.

The following features are available:

- Browsing support for Modelica projects, packages, and classes.
- Wizards for creating Modelica projects, packages, and classes.
- Syntax color highlighting.
- Syntax checking.
- Code completion when writing code to reference a class.
- Code completion/signature information when writing function calls.
- Browsing of the Modelica Standard Library and other Modelica package hierarchies.
- Support for MetaModelica extensions to standard Modelica.

8.3.1 Using the Modelica Perspective

The most convenient way to work with Modelica projects is to use the Modelica perspective. To switch to the Modelica perspective, choose the `Window` menu item, and select `Open Perspective` followed by `Other...`. Select the `Modelica` option from the dialog presented and click `OK`.

8.3.2 Creating a Project

To start a new project, use the `New Modelica Project Wizard`. It is accessible through `File->New->Modelica Project` or by right-clicking in the `Modelica Projects` view and selecting `New->Modelica Project`.

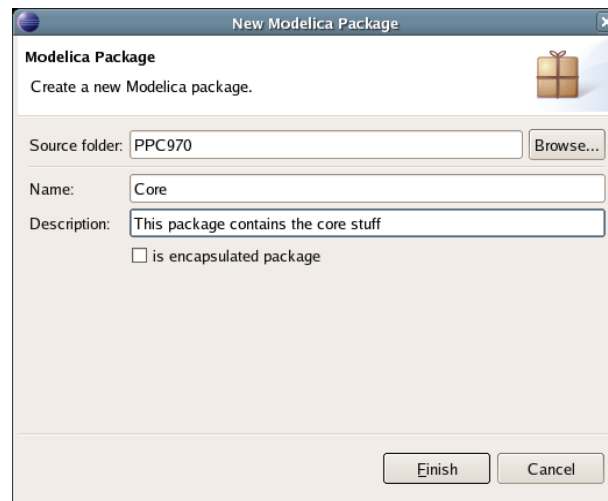


Figure 8-4. Creating a new package.

8.3.3 Creating a Package

To create a new package inside a Modelica project, select `File->New->Modelica Package`. Enter the desired name of the package and a description of what it contains.

8.3.4 Creating a Class

To create a new Modelica class, select where in the hierarchy that you want to add your new class and select `File->New->Modelica Class`. When creating a

Modelica class you can add different restrictions on what the class can contain. These can for example be `model`, `connector`, `block`, `record`, or `function`.

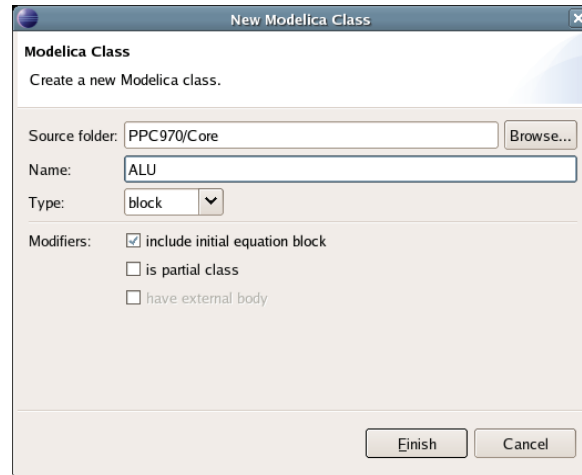


Figure 8-5. Creating a new class.

When you have selected your desired class type, you can select modifiers that add code blocks to the generated code. 'Include initial code block' will for example add the line 'initial equation' to the class.

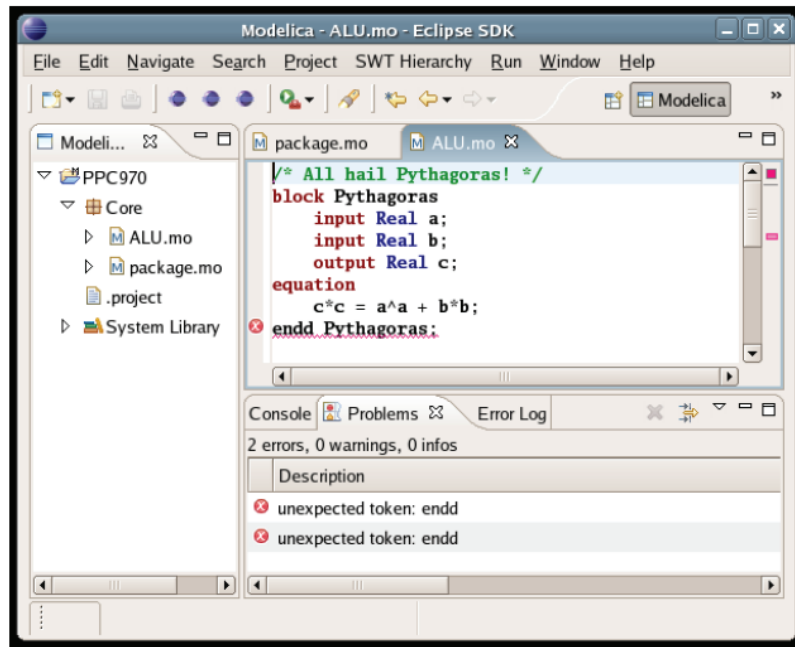


Figure 8-6. Syntax checking.

8.3.5 Syntax Checking

Whenever a Modelica (.mo) file is saved by the Modelica Editor, it is checked for syntactical errors. Any errors that are found are added to the Problems view and also marked in the source code editor.

Errors are marked in the editor as a red circle with a white cross, a squiggly red line under the problematic construct, and as a red marker in the right-hand side of the editor. To reach the problem, one can either click the item in the Problems view or select the red box in the right-hand side of the editor.

8.3.6 Code Completion

MDT supports Code Completion in two variants. The first variant, code completion when typing a dot after a class (package) name, shows alternatives in a menu:

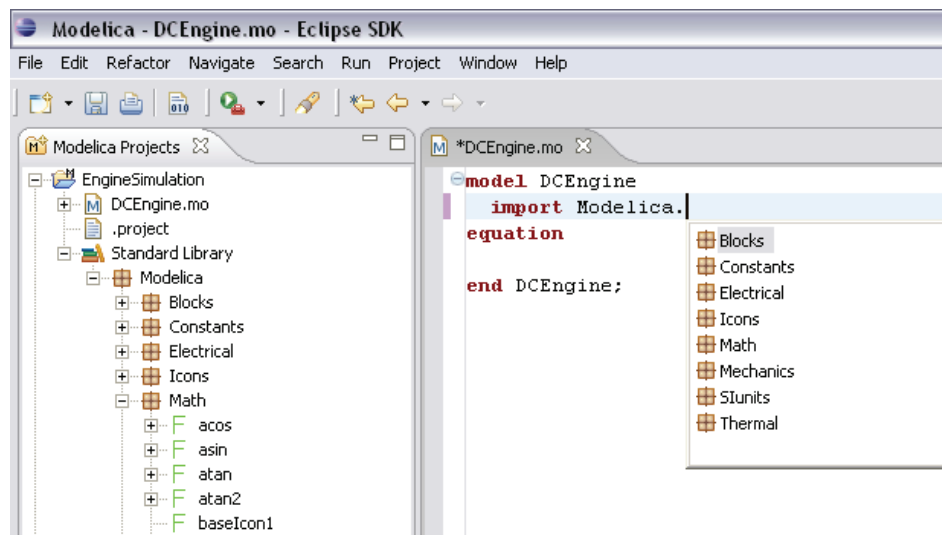


Figure 8-7. Code completion using a popup menu after a dot

The second variant is useful when typing a call to a function. It shows the function signature (formal parameter names and types) in a popup when typing the parenthesis after the function name, here the signature `Real sin(SI.Angle u)` of the `sin` function:

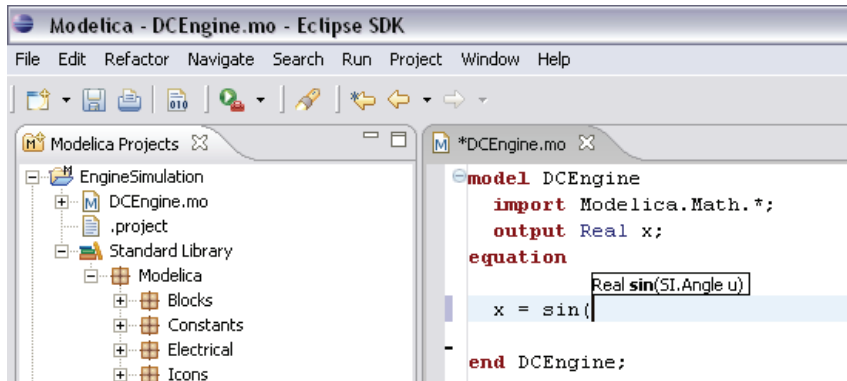


Figure 8-8. Code completion showing a popup function signature after typing a left parenthesis.

8.3.7 Automatic Indentation

MDT also has support for automatic indentation. When typing the Return (Enter) key, the next line is indented correctly. You can also correct indentation of the current line or a range selection using CTRL+I or “Correct Indentation” action on the toolbar or in the Edit menu.

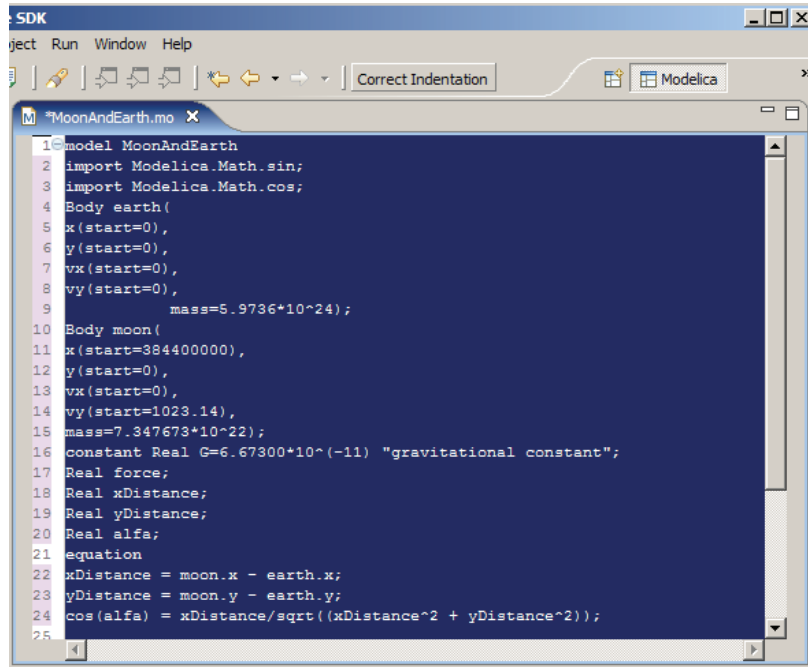


Figure 8-9. Example of code before indentation.

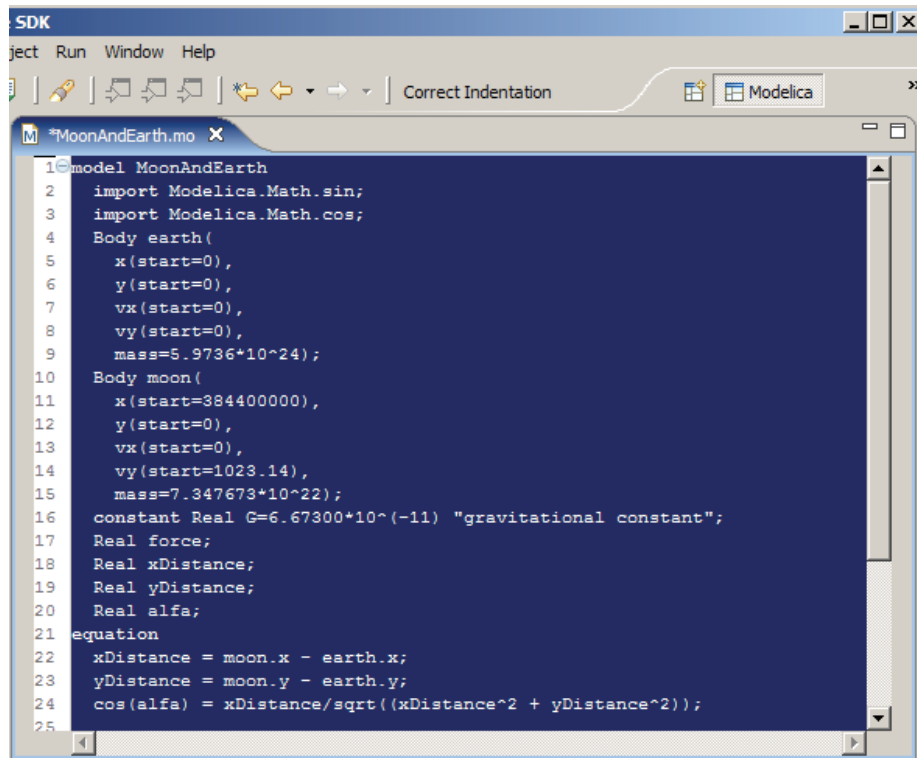


Figure 8-10. Example of code after automatic indentation.

Indentation can be applied to incomplete code as a heuristic Modelica scanner is used and the indentation is based only on the tokens generated by this scanner. The indenter indents one line at a time. For example, consider that line four (4) in Figure 8-10 should be indented. The indenter asks the heuristic scanner to give tokens from in backwards direction to the start of the file until a scope introducer is recognized, which for this particular file is `model MoonAndEarth`. The reference position of the start of the scope introducer is computed and line four (4) is indented from this reference position on indent unit. The indentation result is presented in Figure 8-10.

Indenting Modelica code is far from trivial when incomplete (possibly incorrect) code should be indented correctly. Most of the difficulty comes from Modelica scopes which are hard to recognize using just a scanner and some logic behind it. In languages like C/C++ and Java finding enclosing scopes is very easy as one character tokens are used for the scope opening and closing: "{" and "}". In Modelica you need at least two tokens and a lot of case analysis to find where a scope starts and ends. Complications also arise when mixing if statements with if expressions (which was further complicated by the introduction of the conditional declarations). In this particular case we implemented a parser emulator that recognizes these constructs based on scanner tokens delivered backwards.

The indenter works in almost all cases, but there are cases in which it is impossible to find the correct indentation. For example when the indentation of a line consisting of `end Name;` is requested and the scope introducer for `Name` is not found (that is identifier `Name` followed backwards by `class`, `model`, `package`, `block`, `record`, `connector` etc.) then the indenter fails and returns the indentation of the previous line.

8.4 The OpenModelica Debugger Integrated in Eclipse

We have integrated our algorithmic debugger (Chapter 5), also (Pop and Fritzson 2005 [128]) within the Eclipse debugging framework.

The communication protocol between MDT and the debugger (which is included in the compiled executable build for simulation) is based on a client-server architecture and is implemented via sockets. The debugger is the client and MDT is the server. When the debugged model is simulated, the debugger receives from MDT all the breakpoints set within the algorithmic code. Then the debugger resumes the program. When a break condition becomes true the debugger stops the program and listens on commands it may receive from MDT. The commands accepted by the MDT client are classic: variable value printing, stack trace printing, stepping, running, etc. MDT sends appropriate commands to the debugger, parses the information received and displays it within the MDT debugging views to be inspected by the programmer.

Because algorithmic code can be executed millions of times within a simulation, is very important to be able to specify breakpoints based on variable values and/or the number of times a function executes. These types of breakpoints were newly added to the debugging framework and are now available.

8.5 Simulation and Plotting from MDT

Simulation and plotting is possible from a special command window, where commands are sent to OMC. For example, to simulate:

```
>> simulate(Influenza,startTime=0.0, stopTime=3.0)

record
  resultFile = "Influenza_res.plt"
end record
```

The simulated population is plotted, which is shown in Figure 8-11.

```
>> plot({Infected_Popul.p})
true
```

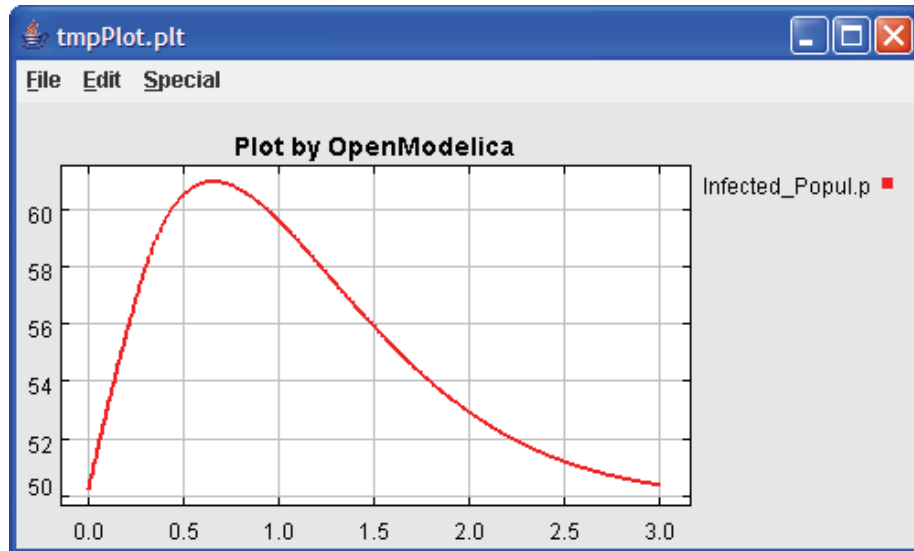


Figure 8-11. Plot of the Influenza model.

8.6 Conclusions

The OpenModelica integrated development environment for Modelica has been augmented with a plugin to the Eclipse framework. The plugin, called MDT (Modelica Development Tooling) (Pop et al. 2006 [131]), is primarily aimed at development of large models or specifications. It has support for browsing, editing, code completion, automatic indentation, building executables, and debugging. It also allows simulation and plotting from a special command window. Further extension and integration of MDT with UML-based modeling is presented in Chapter 10.

To summarize, MDT provides a rather complete integrated development environment, and it is also the first available Eclipse plugin for an equation-based language.

Chapter 9

Parsing-Unparsing and Refactoring

In this chapter based on (Fritzson et al. 2008 [52]) we present a strategy for comment- and indentation preserving refactoring and unparsing for Modelica. The approach is general, and is currently being implemented for Modelica in the OpenModelica environment. We believe this to be one of the first unparsing approaches for equation-based object-oriented languages that can preserve all user-defined indentation and comment information, as well as fulfilling the principle of minimal replacement at refactorings.

9.1 Introduction

Integrated programming environments, e.g. InterLisp (Warren 1974 [171]) and Eclipse (Eclipse.Foundation 2001-2008 [29]) provide various degrees of support for program transformations intended to improve the structure of programs – so-called *refactorings* (Fowler et al. 1999 [39]) (see also Section 9.7).

Such operations typically operate on abstract syntax tree (AST) representations of the program. Therefore the program needs to be converted to tree form by *parsing* before refactoring, and be converted back into text by the process of *unparsing*, also called *pretty printing*. This is supported by a number of environments (section 9.7).

However, a well-known problem is that of preserving comments and user-defined indentation while performing refactorings. Essentially all current environments either lose the comments (except for special comments that are part of the language syntax and AST representation), or move them to some other place. User-defined indentation is typically lost and replaced by machine-generated standard indentations. This is accepted by some developers, but judged as unacceptable by others. However, if the objective only is to improve indentation, then a semi-automatic indenter can be used instead (section 9.5.3.3).

Currently Modelica-based tools are handling only declaration comments that are part of the model and are discarding or moving all the other comments, i.e. the ones between `/* */` and after `//...`. Such behavior is highly undesirable from a user perspective and heavily affects the ease-of-use of code-versioning tools.

A goal for the work presented here is to support Modelica code refactoring with minimal disruption of user-defined comments and indentation. In this chapter we present such an approach for unparsing in conjunction with refactorings.

9.2 Comments and Indentation

Regard the following contrived Modelica example. It has one declaration comment which is part of the language syntax, and two “textual” comments `Itemcomm` and `MyComm` which would be eliminated by a conventional parser. It is also nicely hand formatted so that the start positions of each component name in the text are vertically aligned.

```
record MODIFICATION "Declaration comment"
  Boolean          finalItem; //Itemcomm
  Each /* MyComm */ eachRef;
  ComponentRef     componentReg;
end MODIFICATION;
```

Assume that this is parsed and unparsed by a conventional (comment-preserving) unparsing, putting two blanks between the type and the component name of each component. The manual indentation would be lost, and the “textual” comments would be moved to some standard positions (or be lost):

```
record MODIFICATION "Declaration comment"
  Boolean  finalItem; //Itemcomm
  Each  eachRef; /* MyComm */
  ComponentRef  componentReg;
end MODIFICATION;
```

9.3 Refactorings

Below we make some general observations and give examples of refactorings.

9.3.1 The Principle of Minimal Replacement

For a refactoring to have minimal disruption on the existing code, it is desired that it supports the principle of minimal replacement:

- *When replacing a subtree, the minimal subtree that contains the change should be replaced.*

This also has the consequence of minimal loss or change of comments. For example, if a name (an identifier) is changed, only the identifier node in the tree should be replaced, not the surrounding subtree.

9.3.2 Some Examples of Refactorings

Here we mention a few common refactorings. There are also numerous, more advanced and specialized refactorings.

- *Component name change.* Change name of a component name in a record. For example:

```
record MODIFICATION "Declaration comment"
  Boolean          finalItem; //Itemcomm
  Each /* MyComm */ eachRef;
  ComponentRef     componentReg;
end MODIFICATION;
```

The name of the component reference name is currently `componentReg`, which is an error. It should be `componentRef`. We would like to change the name both in the declaration and all its uses, thus avoiding updating all named references by hand, which would be quite tedious.

- *Function name change.* Change the name of a function, both the declaration and all call sites.
- *Add record component.* Add a new component declaration to a record. In MetaModelica, that would also mean putting an underscore '_' at the correct position in all patterns for that record type with positional matching.
- *Add function formal parameter.* Add an input or output formal parameter to a function. The question is, how much is possible to do automatically? Adding arguments to recursive calls to the function itself is no great problem, but calls from other functions can be more problematic since meaningful input data needs to be provided. This can be handled easily in those cases a default value can be passed to the function's new formal parameter.

9.3.3 Representing Comments and User-Defined Indentation

How should information about comments and user defined indentation be represented in the internal (AST) program representation? There are basically two possibilities for a chunk of code, e.g. a model:

- *Tree.* The AST representation is the main storage (the TRUTH). Comments and indentation as extra nodes/attributes in the AST.

- *Text*. The text representation, including indentation and comments, is the main storage (the TRUTH).

The tree approach may seem natural, since the refactorings and the compiler operate on the tree representation. However, it has some disadvantages:

- Since white space and comments can appear essentially anywhere, between nodes, associated with nodes, the AST will become cluttered and increase the required memory usage and complexity of the tree, perhaps by a factor 2-3.
- The large number of extra nodes in the AST may complicate code accessing and traversing the tree.

Regarding the text representation we make the following observations:

- The text representation exists from the start, since this is the storage form used in the file system. Environments like Eclipse use text buffers for direct interaction with the programmer.
- The text representation includes all indentation and comment information, and is compact.
- The structure of the program in the text representation is not apparent, and cannot be easily manipulated.

Why not combine the advantages of each representation, and try to avoid the disadvantages?

- Use the text representation as the basic storage format including indentation and comment information. The text might be conceptually divided into chunks, where for example each class definition gives rise to a text chunk.
- Use the tree representation for compilation and refactoring. Create it when needed and keep it during the current session. Create it piece-wise, e.g. for one class at a time.
- Create a mapping from the tree representation to the text representation; each node in the tree has a corresponding position and size in the text representation. Create this mapping when needed, for appropriate pieces (e.g. class definitions) of the total model.

9.4 Implementation

The strategy used for the implementation is described in the following sections.

9.4.1 Base Program representation

The text representation is the TRUTH, the source, and the AST representation is a secondary representation derived from the source, used during compilation and refactoring.

The class information attribute of a class definition in the AST should be extended, e.g. with the byte start position (directly addressing within a file), or by a text chunk corresponding to the text of a class declaration. A package which contains classes would instead refer to the definitions of those classes.

Text positions and text sizes of each AST node should be indirectly associated with each AST node.

9.4.2 The Parser

The following special considerations need to be addressed by the parser:

- In order not to clutter the produced AST tree, the parser produces two trees: a standard AST tree, and a positioning tree (produced in parallel) with the same number of nodes, containing text positions and sizes of each subtree.
- The parser should return the start text position and text size of each built AST tree. Moreover, if there are any comments within the AST tree text range, a list of the start positions and sizes of these comments should be associated with the parallel tree node.
- The pure AST tree should be clean and not cluttered with position and comment information.
- As mentioned, a text position tree with the same number of nodes and children as the AST is created in parallel to the AST. The positioning tree is only produced when needed for refactorings or text positioning, and thrown away when not needed.

For example, a child nr 3 of a node at level 2, will find its text positions in the parallel tree in the node at level 2 and child nr 3.

9.4.3 The Scanner

The text position and size of each token is returned together with the token itself.

9.4.4 The New Unparser

The new unparser will use a combined strategy as follows, combining existing text with new text generated by the tree unparser:

- If there exist already indented text associated with a node, use this text to produce the unparsing text.
- If there is no existing text, this must be a new tree node produced by the refactoring tool. Call the tree unparsing to convert this subtree into text that is inserted into the final unparsing result.

9.5 Refactoring Process

The following steps are to performed in this order during the actual refactoring:

- Traverse the AST and perform insertion/deletion/ replacement of subtrees.
- For each insertion/deletion/replacement operation, put each such an operation descriptor in a list, together with the text position and size of the text of the subtree to be replaced/deleted etc.
- After traversal, sort these operations according to text position, and perform the operations in the text in backwards order (take those at the highest text position first).

9.5.1 Example of Function Name Refactoring

The example below is used to illustrate the refactorings and the used combined tree and text chunk representation.

All loaded models (including the `Modelica` package) reside in an un-named top-level scope that we can call `Top`. A model may be a top-level model, but more typically a package which in turn may consist of subpackages:

```
01 within ParentPackage;  
02 package pack  
03   function addOne "function that adds 1"  
04     input Real x = 1.0; // line comment  
05     output Real y;      /* multiple  
06                          line  
07                          comment */  
08   algorithm  
09     y := x + 1.0;  
10   end addOne;  
11  
12   class myClass  
13     Real y;  
14   equation  
15     y = addOne(5); // Call to addOne  
16   end myClass;  
17 end pack;
```

Line numbers are given to help the reader follow the example. The position tree constructed by the parser is given in the appendix as it is quite large. A portion of the abstract syntax tree is also shown in order to understand the example.

A function name refactoring will be applied to the example which will change the name of the function "addOne" to "add1". The refactoring can be performed in the OpenModelica environment by loading the example and calling the interactive API function:

```
loadFileForRefactoring("Example.mo");
refactorFunctionName(pack.addOne, "add1");
```

The compiler will execute the first command by calling the new parser that also builds the position tree together with the AST:

```
(ast,posTree) = Parse.refactorParse(file);
```

The result of the load command is two trees. The second (`posTree`) is the position tree presented (partly) in the appendix. The first (`ast`) is the abstract syntax tree of the loaded file which is presented also in the appendix entirely. Here is just a overview picture of the AST:

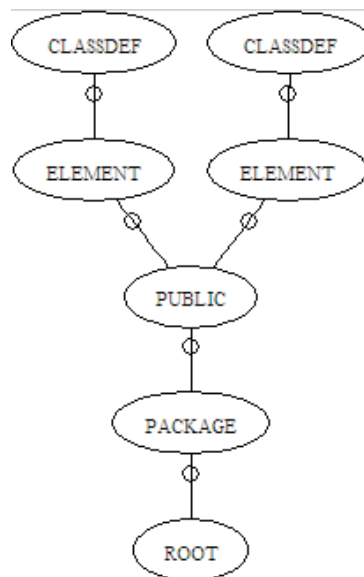


Figure 9-1. AST of the Example.mo file.

The figure shows that the program has one package with two public elements which are class definitions.

Actually only two refactoring operations are needed to implement any refactoring: add and delete or add and replace.

When `refactorFunctionName` is called the compiler will perform these operations:

9.5.1.1 Lookup pack.addOne

Lookup of a class definition is performed by walking the AST while keeping track of a numbered path in the tree. To reach the `addOne` identifier, the path: 1, 6, 1, 1, 1, 5, 2, 1, 1 is applied. The path goes via the following AST nodes in order to reach the desired class name: PROGRAM [1] / CLASS [6] / PARTS [1] / PUBLIC [1] / ELEMENTITEM [1] / ELEMENT [5] / CLASSDEF [2] / CLASS [1] / IDENT("addOne") [1].

9.5.1.2 Lookup Any Uses of pack.addOne

Lookup of the uses are performed by walking the AST, keeping track of the scope, while keeping track of a numbered path. To reach the function call of `addOne`, the path: 1, 6, 1, 1, 1, 5, 2, 1, 1 is applied. The path goes via the following AST nodes:

```
PROGRAM [1] / CLASS [6] / PARTS [1] / PUBLIC [2] / ELEMENTITEM
[1] / ELEMENT [5] / CLASSDEF [2] / CLASS [6] / PARTS[1] /
EQUATIONS [1] / EQUATIONITEM [1] / EQ_EQUALS [2] / CALL[1] /
CREF_IDENT [1] / IDENT("addOne") [1].
```

9.5.1.3 Apply the Refactoring to the Actual Text

Now that the paths needed for the minimal refactoring were discovered in the AST, apply these paths to the position tree and fetch the positions of the elements at the end of the paths:

- Function name: IDENT, Start:047, End:053
- Function use: IDENT, Start:313, End:319

The text operations are applied bottom-up because otherwise the character positions of the elements below an applied operation would change. Ordering of text operations is needed to have them applied in a bottom-up fashion:

- `ReplaceText(file, 319, 313, "add1");`
- `ReplaceText(file, 53, 47, "add1");`
- `Close(file);`
- `(ast, posTree) = // re-parse the file Parse.refactorParse(file);`

After the file is closed either a reparsing is performed to load the new AST (as exemplified here) or the refactoring operations are performed on the tree already in the memory. Of course the best alternative would be to perform the refactoring during lookup as we have implemented it in the OpenModelica compiler.

As one can notice the comments stay in place so there is minimal disruption to the text representation. This is very valuable from a user point of view but also for code-versioning tools.

9.5.2 Calculation of the Additional Overhead

There is not too much overhead for the refactoring both with respect to memory usage and time spent walking the tree. In the following table we discuss such overhead and give specific numbers for needed memory size and time complexity of the refactoring procedure.

Memory overhead	Time overhead
<p>Space is required for storing the position tree. The size of this space is two integers (of 4 bytes) for each AST node. Also the list of operations to be applied to the text needs memory for storing the paths and the operations themselves, but this memory is negligible compared to the AST and position tree and can also be freed.</p> <p>Example: there are about 50 nodes in the example, which means an additional memory of $\sim 50 \text{NrNodes} \times 2 \text{Positions} \times 4 \text{Bytes} = 400 \text{Bytes}$ are needed for the position tree. Of course, the position tree can be built on demand and then freed when memory is needed.</p>	<p>Walking two trees while performing the refactoring has a time impact of $\text{NumberOfNodesWalked} \times O(1)$ to walk a node: $O(\text{NrOfNodesWalked})$. Walking the position tree while and applying the text operations to the file is negligible compared to the refactoring operation.</p> <p>Example: it took about 0.2 seconds to perform the function name refactoring for the example file using the OpenModelica system. Refactoring old graphical annotations of the Modelica Standard Library version 1.6 to the new style graphical annotations took about 9.6 seconds, which is very good for such a demanding refactoring.</p>

9.5.3 Unparsers/Prettyprinters versus Indenters

As mentioned previously, an unparser converts an AST program representation into (nicely indented) text. A reformatting indentation tool uses another approach: it operates directly on the text representation to produce a more nicely indented text.

9.5.3.1 Pretty printers/Unparser Generators

An unparser generator produces an unparser from a specification, a grammar-like description of unparsing-related aspects of the language. A number of systems mentioned in Section 9.5.3 support unparsing or generation of unparsers from such specifications.

9.5.3.2 OpenModelica Tree Unparser

The current OpenModelica version 1.4 unparser is hand implemented in MetaModelica, recursively traversing the AST while generating the Modelica text representation. It can be invoked by the OpenModelica `list` command. Comments are currently lost (except for declaration comments).

9.5.3.3 Reformatting Indentation in the OpenModelica Eclipse Plugin

A *text reformatting indentation* tool operates directly on the text representation, and analyzes the text by a combination of scanning and piecemeal heuristic partial parsing to recognize certain combinations of tokens. It inserts or removes white space in order to produce a nice indentation, or improve an existing one. Such mechanisms are typically invoked by the user on a few lines at a time, and are not completely automatic; the user is often required to perform the final adjustments. An advantage with this approach is that comments are not lost.

This kind of indentation tool is for example available for a number of languages in their respective Emacs modes, or as part of Eclipse plugins, e.g. for C++, Java, and more recently for Modelica in the OpenModelica MDT Eclipse plugin.

MDT includes support for automatic indentation, as described in Chapter 8 and in (Pop et al. 2006 [131]). When typing the Return (Enter) key, the next line is indented correctly. The user can also correct indentation of the current line or a range selection using CTRL+I or “Correct Indentation” action on the toolbar or in the Edit menu.

Indentation can be applied to incomplete code as a heuristic Modelica scanner is used and the indentation is based only on the tokens generated by this scanner. The indenter indents one line at a time. For example, consider that line four (4) in Figure 8-10 should be indented. The indenter asks the heuristic scanner to give tokens from the starting token in backwards direction to the start of the file until a scope introducer is recognized, which for this particular file is `model MoonAndEarth`. The reference position of the start of the scope introducer is computed and line four (4) is indented from this reference position one indent unit. The indentation result is presented in Figure 8-10.

Indenting Modelica code is far from trivial when incomplete (possibly incorrect) code should be indented correctly. Most of the difficulty comes from Modelica scopes which are hard to recognize using just a scanner and some logic behind it. In languages like C/C++ and Java finding enclosing scopes is very easy as one character tokens are used for the scope opening and closing: “{” and “}”. In Modelica you need at least two tokens and much more case analysis to find where a scope starts and ends. Complications also arise when mixing if-statements with if-expressions (which was further complicated by the introduction of conditional declarations in the Modelica language). In this particular case we implemented a parser emulator that recognizes these constructs based on scanner tokens delivered backwards.

The indenter works well in almost all cases, but there are cases in which is impossible to find the correct indentation. For example when the indentation of a

line consisting of "end Name;" is requested and the scope introducer for Name is not found (that is identifier Name followed backwards by class, model, package, block, record, connector etc.) then the indenter fails and returns the indentation of the previous line.

9.6 Further Discussion

This section addresses additional questions raised during a presentation of the article that this chapter is based on at the Modelica 2008 conference:

Question: "A question I have always had is whether there are any "mistakes" in the grammar that should be corrected with respect to these issues. Similarly, how is this handled with the Java tools in Eclipse?"

Answer: The answer to this question highly depends on the syntactic mistake the user made. For example if an "end if;" is missing at the end of an equation section, but is followed by "end Model;", then such a mistake can be automatically corrected using a heuristic parser. However, if an opening scope is missing, i.e., `model Model` (or alternatively an ending scope) there is no way to know where it should be introduced. There are a lot of places that can be proposed:

- Just after the enclosing scope starts (after i.e., `package MyPack` introduction) if there exists such scope or the start of the file if no such scope exists.
- Just after the every existing ending scope of a model found by going backwards from the `end Model;`

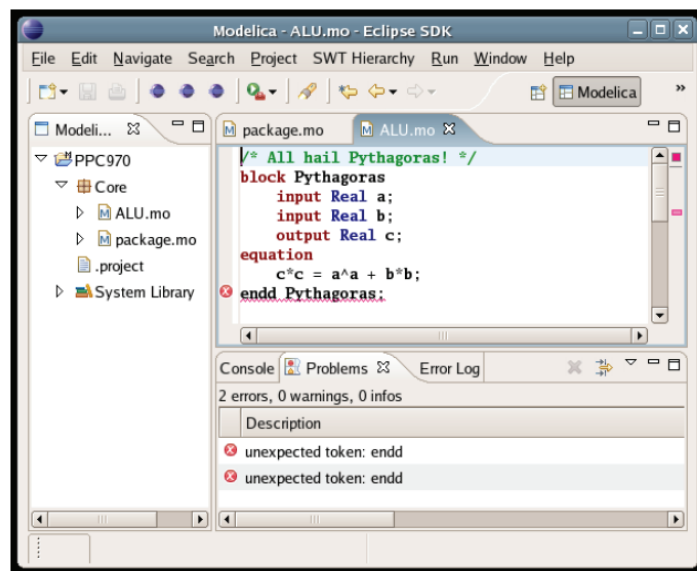


Figure 9-2. Syntax checking.

Right now the Eclipse environment will call the OpenModelica compiler to parse the file each time the file is saved. The parsing errors are reported in the Eclipse environment as a list of errors, but also underlined where the error occurs as shown in Figure 9-2. Of course if the user selects an entire file and calls the automatic indentation routine, the indentation will work correctly if there are no *large grammatical errors in the file*.

Question: “Dymola’s pretty printing algorithm does not appear to be deterministic (it sometimes changes files for no reason just because they have been re-saved). Please discuss this deterministic issue and also what implications the algorithms will have for version control tools (i.e. avoiding complex or unnecessary changes since this will complicate “merge” operations).”

Answer: As exemplified in Sections 9.3.1 and 9.5.1 the disruption to the actual text is minimal so the code-versioning tools would have no problem with merging operations. This was one of our goals when designing and implementing the refactoring tools presented in the chapter. The algorithms in this chapter also apply to Modelica models constructed programmatically because these can also be viewed as refactorings. In general the construction of models programmatically is performed by a visual component diagram editor. The editor will give commands: `addModel(...)`, `addComponent(...)`, `addConnection(...)`, etc., to the internal handler of the textual model (that works on the AST and the positionTree) which in the case of a file with code formatting will minimally disrupt the existing code and add all the new code correctly indented at the end or in other appropriate places.

9.7 Related Work

The term refactoring and its use in a general and systematic sense was introduced by Martin Fowler et al (Fowler et al. 1999 [39]), also based on earlier work, even though similar code transformation operations were previously available, e.g. in the InterLisp environment (Warren 1974 [171]).

Early work in interactive integrated programming environments including unparsing/pretty printing supporting a specific language was done in the InterLisp system for the Lisp language (Warren 1974 [171]), common principles and experience of early interactive Lisp environments are described in (Sandewall 1978 [143]), a generic editor/unparser/parser generator used for Pascal (and later Ada) in the DICE system (Fritzson 1984 [42]), (Fritzson 1983 [41]), the integrated Mjölner environment with multi-language editing and unparsing support (Lindskov et al. 1993 [86]). None of these approaches preserve comments when unparsing, except the InterLisp environment where the comments were already part of the AST which was just pretty printed with a more readable indentation. However, also in the InterLisp case, all hand indentation and white space added by the user is lost, and text style comments (not part of the AST) are also lost.

Many parser generation systems, e.g. ANTLR (Parr 2005 [116]), Eli (Kastens et al. 2007 [77]), CoCo (Mössenböck et al. 2000 [105]), also support unparsing from

[illegible]

```
(list<ElementItem>, (Start: 80, End: 221, {
  (ElementItem, (Start: 80, End: 100, {
    (Element, (Start: 80, End: 100, {
      (Boolean final, (Start: -1, End: -1)
      (Option<RedeclareKeywords>,
        (Start: -1, End: -1)
      (InnerOuter, (Start: -1, End: -1)
      (Ident, (Start: 91, End: 92)
      (ElementSpecEL3, (Start: 91, End: 100, {
        (ElementAttributes, (Start: 80, End: 85, {
          (Boolean flow, (Start: -1, End: -1)
          (Variability, (Start: -1, End: -1)
          (Direction, (Start: 80, End: 85)
          (ArrayDim, (Start: -1, End: -1))
        (TypeSpec, (Start: 86, End: 90, {
          (Path, (Start: 86, End: 90, {
            (Ident, (Start: 86, End: 90))
          (Option<ArrayDim>,
            (Start: -1, End: -1)
          ))
        ))
      ... // truncated text due to its large size
    }) (Option<String>, (Start: -1, End: -1)
  }) (Info, (Start: -1, End: -1)
}))
(Within, (Start: 1, End: 7,
  (Path, (Start: 8, End: 22, {(Ident, (Start: 8, End: 22))}))
```

Here is another version of the example with character positions for end and start of a Modelica construct:

```
[001]within[007] [008]ParentPackage;[022]
[023]package[030] [031]pack[035]
[036] [038]function[046] [047]addOne[053] [054]"function
that adds 1"[076]
[077] [080]input[085] [086]Real[090] [091]x[092]
[093]=[094] [095]1.0;[099]
[100]// line comment[115]
[116] [119]output[125] [126]Real[130] [131]y;[133]
[139]/* multiple
line
comment */[221]
[222] [224]algorithm[233]
[234] [237]y[238] [239]:=[241] [242]x[243] [244]+[245]
[246]1.0;[250]
[251] [253]end[256] [257]addOne;[264]
[265]
[266] [268]class[273] [274]myClass[281]
[282] [286]Real[290] [291]y;[293]
[294] [296]equation[304]
```

```

[305]      [309]y[310] [311]=[312] [313]addOne[319] (5);[323]
[324]// Call to addOne[341]
[342] [344]end[347] [348]myClass;[356]
[357]end[360] [361]pack;[366]

```

Parts of the abstract syntax tree (AST) of the Example.mo in the example section is presented below. The AST has exactly the same structure as the position tree.

```

adrpo@KAFKA /c/home/adrpo/doc/projects/modelica2008/
$ omc +d=dump Example.mo
Absyn.PROGRAM([
  Absyn.CLASS(Absyn.IDENT("pack"),
    false, false, false, Absyn.R_PACKAGE,
    Absyn.PARTS(
      [Absyn.PUBLIC(
        [Absyn.ELEMENTITEM(
          Absyn.ELEMENT(false, _, Absyn.UNSPECIFIED,
            "function",
            Absyn.CLASSDEF(false,
              Absyn.CLASS(Absyn.IDENT("addOne"),
                false, false, false, Absyn.R_FUNCTION,
                Absyn.PARTS(
                  [Absyn.PUBLIC(
                    [Absyn.ELEMENTITEM(
                      Absyn.ELEMENT(false, _, Absyn.UNSPECIFIED,
                        "comp",
                        Absyn.COMPONENTS(Absyn.ATTR(false,
                          Absyn.VAR, Absyn.INPUT, []),
                          Absyn.PATH(Absyn.IDENT("Real")),
                          [Absyn.COMPONENTITEM(
                            Absyn.COMPONENT(Absyn.IDENT("x"), [],
                              SOME(Absyn.CLASSMOD([],
                                SOME(Absyn.REAL(1.0))))), NONE)]),
                        Absyn.INFO("Example.mo",
                          false, 4, 4, 4, 22)), NONE)),
                    Absyn.ELEMENTITEM(
                      Absyn.ELEMENT(false, _,
                        Absyn.UNSPECIFIED, "component",
                        Absyn.COMPONENTS(Absyn.ATTR(false,
                          Absyn.VAR, Absyn.OUTPUT, []),
                          Absyn.PATH(Absyn.IDENT("Real")),
                          [Absyn.COMPONENTITEM(
                            Absyn.COMPONENT("y", [], NONE), NONE)]),
                        Absyn.INFO("Example.mo",
                          false, 5, 4, 5, 17)), NONE)))]),
                  Absyn.ALGORITHMS(
                    ALGORITHMITEM(
                      ALG_ASSIGN(
                        Absyn.CREF(Absyn.CREF_IDENT("y", [])),
                        Absyn.BINARY(
                          Absyn.CREF(Absyn.CREF_IDENT("x", [])),

```

```
        Absyn.ADD,  
        Absyn.REAL(1.0)))))],  
        SOME("function that adds 1")),  
        Absyn.INFO("Example.mo", false, 3, 3, 10, 13))  
    ... // truncated text due to its large size  
], // end of Absyn.CLASS list  
Absyn.WITHIN(Absyn.IDENT("ParentPackage"))  
) // end Absyn.PROGRAM
```


Chapter 10

UML and Modelica System Modeling with ModelicaML

10.1 Introduction

Complex products are increasingly consisting of both software and hardware components which are closely interacting. Thus, modeling tools and processes need to support co-design of software and hardware in an integrated way. Currently, UML is the dominant graphical modeling notation for software, whereas Modelica is the major object-oriented mathematical modeling language for component-oriented modeling of complex physical systems, e.g., systems containing mechanical, electrical, electronic, hydraulic, thermal, control, electric power or process-oriented subcomponents. Here we present the first comprehensive UML-Modelica-SysML integrated modeling environment as a ModelicaML profile integrated in Eclipse as a plugin. The profile reuses some artifacts from the System Modeling Language (SysML) profile, and combines the major UML diagrams with Modelica graphic connection diagrams. Requirement, equation, and simulation diagrams are also supported in an integrated way. Moreover, the availability of the UML-style internal class diagram view for Modelica classes may also ease the understanding of modeling with Modelica for software developers with a UML background.

One of the most important paradigm shifts occurring in engineering system design and product development may well be the adoption of common system models, as a foundation for product/system design. This allows for a much more effective product development process since a system can be analyzed and tested in all stages of design.

The development in system modeling has come to the point where complete modeling of systems is possible, e.g. the complete propulsion system, fuel system, hydraulic actuation system, etc., including embedded software can be modeled and simulated concurrently. This does not mean that all components are dealt with down

to the very smallest details of their behavior. It does, however, mean that all functionality is modeled, at least qualitatively.

Furthermore, in contrast to the usual problem oriented approach, the test applications to be simulated with the model typically are not explicitly known when the model is established. Perhaps more importantly, an aspect-oriented system model can carry all information about the system under development, and be the blueprint that all engineers work towards.

Model-based product development needs multi-disciplinary competence. Until recently rather few efforts have been started to bring these together despite the industrial importance of such an integration.

10.2 SysML vs. Modelica

The System Modeling Language (SysML) has recently been proposed and defined as an extension of UML targeting at systems engineers. The goal of SysML is to unify different approaches and languages used by system engineers into a single standard which supports specification, analysis, design and verification of complex systems. SysML models may span different domains, for example, electrical, mechanical and software. Even if SysML provides means to describe system behavior like Activity and State Chart Diagrams, the precise behavior can not be described and simulated without complex transformations and additional information provided for SysML models. In that respect, SysML is rather incomplete compared to Modelica.

Analogous to SysML, Modelica was created to unify and extend various object-oriented mathematical modeling languages. It has powerful means for describing precise component behavior and functionality in a declarative way. Modelica models can be graphically composed using Modelica connection diagrams which depict the structure of designed system. However, complex system design is more than just a component assembly. In order to build a complex system, system engineers have to gather requirements, specify system components, define system structure, define design alternatives, describe overall system behavior and perform its validation and verification.

The current work combines UML with Modelica. Particularly, a UML profile for Modelica, named ModelicaML, is proposed. The ModelicaML UML profile is based on the SysML UML profile and reuses its artifacts required for system specification. SysML diagrams are also extended to support all Modelica constructs. We argue that with ModelicaML system engineers are able to specify entire systems, starting from requirements, continuing with behavior and finally perform system simulations.

10.3 ModelicaML: a UML profile for Modelica

ModelicaML reuses several diagrams types from SysML without any extension, extends some of them, and also provides several new ones. The ModelicaML diagram overview is shown in Figure 10-1. Diagrams are grouped into four categories: Structure, Behavior, Simulation and Requirement. In the following we present the most important ModelicaML profile diagrams. For a full description of the profile, please refer to (Akhvlediani 2007 [1]).

The most important properties of the ModelicaML profile are outlined below:

- The ModelicaML profile supports modeling with all Modelica constructs and properties i.e. restricted classes, equations, generics, discrete variables, etc.
- Using ModelicaML diagrams it is possible to describe all aspects of a system being designed and thus support system development process phases such as requirements analysis, design, implementation, verification, validation and integration.
- ModelicaML is partly based on SysML, but reuses and extends its elements.
- The profile supports mathematical modeling with equations since equations specify behavior of a (Modelica) system. Algorithm sections are also supported.
- Simulation diagrams are introduced to model and document simulation parameters and results in a consistent and usable way.
- The ModelicaML meta-model is consistent with SysML in order to provide SysML-to-ModelicaML conversion.

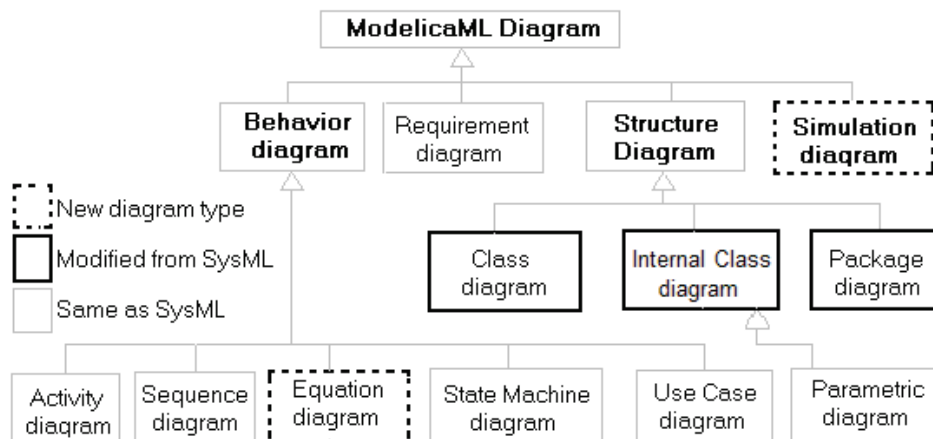


Figure 10-1. ModelicaML diagrams overview.

Three SysML diagram types have been partly reused and changed for the ModelicaML profile:

- The SysML Block Definition Diagram has been updated and renamed to Modelica Class Diagram.
- The SysML Internal Block Diagram has been updated and renamed to Modelica Internal Class Diagram (some of the SysML constructs are disabled).
- The Package Diagram has been changed in order to fully support the Modelica language (i.e. Modelica package constants).

Thus, the following diagram types are available in the ModelicaML profile:

- The *Modelica Class Diagram* usually describes class definitions and their relationships such as inheritance and containment.
- The *Modelica Internal Class Diagram* describes the internal class structure and interconnections between parts.
- The *Package Diagram* groups logically connected user defined elements into packages. In ModelicaML the primarily purpose of this diagram is to support the specifics of the Modelica packages.
- Activity, Sequence, State Machine, Use Case, Parametric and Requirements diagrams have been reused without modification from SysML.
- Two new diagrams, *Simulation Diagram* and *Equation Diagram*, not present in SysML, have been included in the ModelicaML profile.

10.3.1 Modelica Class Diagrams

Modelica uses restricted classes such as `class`, `model`, `block`, `connector`, `function` and `record` to describe a system. Modelica classes have essentially the same semantics as SysML blocks and provide a general-purpose capability to model systems as hierarchies of modular components. ModelicaML extends SysML blocks by defining features which are relevant or unique to Modelica.

The purpose of the Modelica Class Diagram is to show features of Modelica classes and relationships between classes. Additional kind of dependencies and associations between model elements may also be shown in a Modelica Class Diagram. For example, behavior description constructs – equations, may be associated with particular Modelica Classes. The detailed description of structural features of ModelicaML is provided below. ModelicaML structural extensions are defined based on the SysML block definitions

10.3.1.1 ModelicaML Class Definition

The graphical notation of ModelicaML class definitions is shown in Figure 10-2. Each class definition is adorned with a stereotype name that indicates the class type it represents.

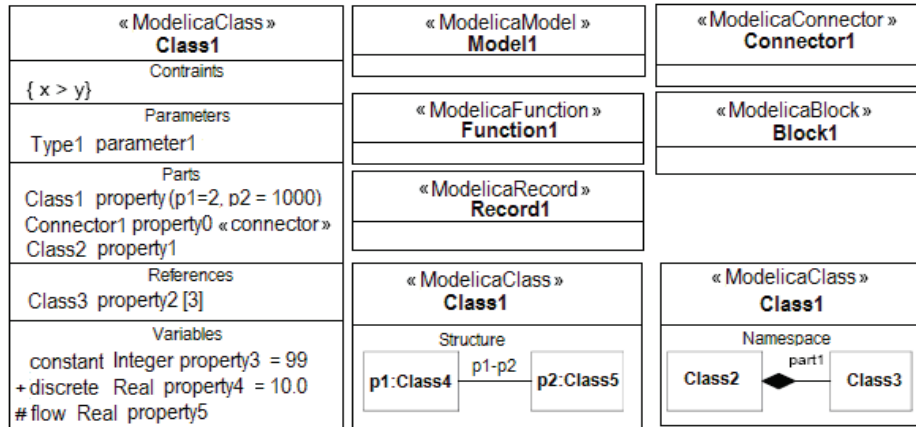


Figure 10-2. ModelicaML class definitions.

The ModelicaML Class Definition has several compartments to group its features: parameters, parts, variables. Some compartments are visible by default; some are optional and may be shown on ModelicaML Class Diagram with the help of a tool. Property signatures follow the Modelica textual syntax and not the SysML original syntax, reused from UML. A ModelicaML/SysML tool may allow users to choose between UML or Modelica style textual signature presentation. Using Modelica syntax on a diagram has the advantage of being more compatible with Modelica and being more straightforward for Modelica users. The Modelica syntax is quite simple to learn even for users not acquainted with Modelica.

ModelicaML provides extensions to SysML in order to support the full set of Modelica constructs and features. For example, ModelicaML defines unique class definition types `ModelicaClass`, `ModelicaModel`, `ModelicaBlock`, `ModelicaConnector`, `ModelicaFunction` and `ModelicaRecord` that correspond to `class`, `model`, `block`, `connector`, `function` and `record` restricted Modelica classes.

10.3.1.2 Modelica Internal Class Diagram

The Modelica Internal Class Diagram is based on the SysML Internal Block Diagram. The Modelica Class Diagram defines Modelica classes and relationships between classes, like generalizations, association and dependencies, whereas a Modelica Internal Class Diagram shows the internal structure of a class in terms of parts and connections. The Modelica Internal Class Diagram is similar to Modelica connection diagram, which presents parts in a graphical (icon) form.

An example Modelica model presented as a Modelica Internal Class diagram is shown in Figure 10-3.

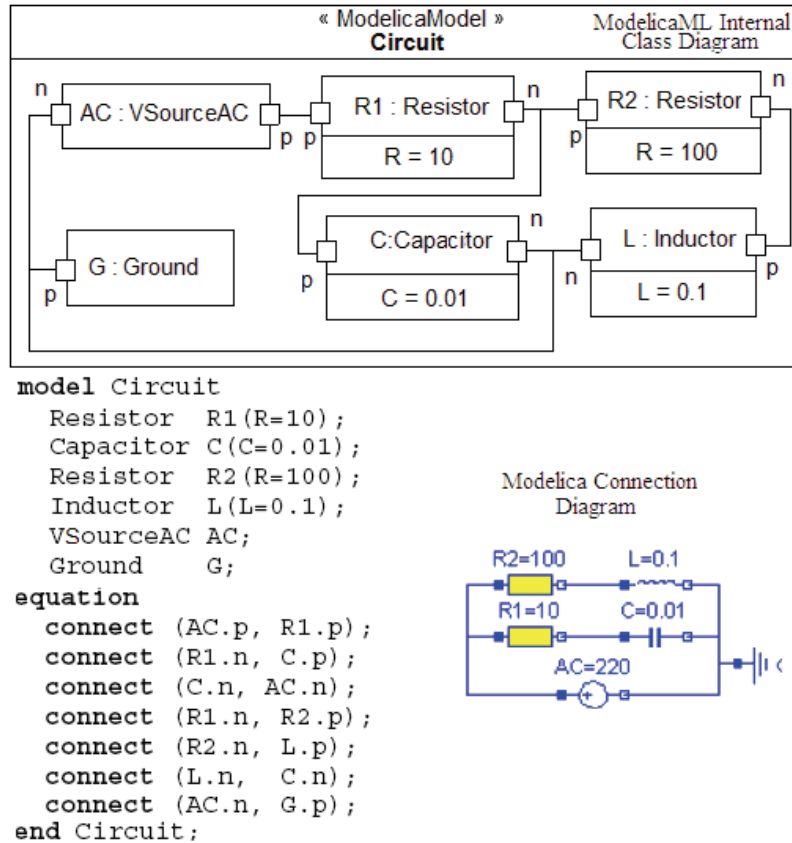


Figure 10-3. ModelicaML Internal Class vs. Modelica Connection Diagram.

Usually Modelica models are presented graphically via Modelica connection diagrams (Figure 10-3, bottom). Such diagrams are created by the modeler using a graphic connection editor by connecting together components from available libraries. Since both diagram types are used to compose models and serve the same purpose, we briefly compare the Modelica connection diagram to the Modelica Internal Class Diagram. The main advantage of the Modelica connection diagram over the Internal Class Diagram is that it has better visual comprehension as components are shown via domain-specific icons known to application modelers. Another advantage is that Modelica library developers are able to predefine connector locations on an icon, which are related to the semantics of the component. In the case of a ModelicaML Internal Class Diagram a SysML/ModelicaML tool should somehow point out at which side of a rectangular presentation of a part to place a port (connector).

One of the advantages of the Internal Class Diagram is that it directly supports nested structures. However, nested structures are also available behind the icons in a Modelica connection diagram, thus using the drawing area more effectively.

The main advantage of the Internal Class Diagram is that it highlights top-level Modelica model parameters and variables specification in separate compartments.

Other SysML elements, such as Activities and Requirements which do not exist in Modelica but are very important for additional model specification can be combined with both Internal Class Diagram and Modelica connection diagrams.

10.3.1.3 Package Diagram

A UML Package is a general purpose model element for grouping other elements within a separate namespace. With a help of packages, designers are able group elements to correspond to different structures/views of a system. ModelicaML extends SysML packages in order to support Modelica packaging features, in particular: package inheritance, generic packages, constant declaration within a package, package “instantiation” and renaming import (see (Fritzson 2004 [44]) for Modelica packages details).

A diagram which contains package elements and their relationships is called a Package Diagram. Modelica packages have a hierarchical structure containing package elements as nodes. In Modelica, packages are used to structure model elements into libraries. A snapshot of the Modelica Standard Library hierarchy is shown in Figure 10-4 using UML notation. Package nodes in the hierarchy are connected via the package containment link.

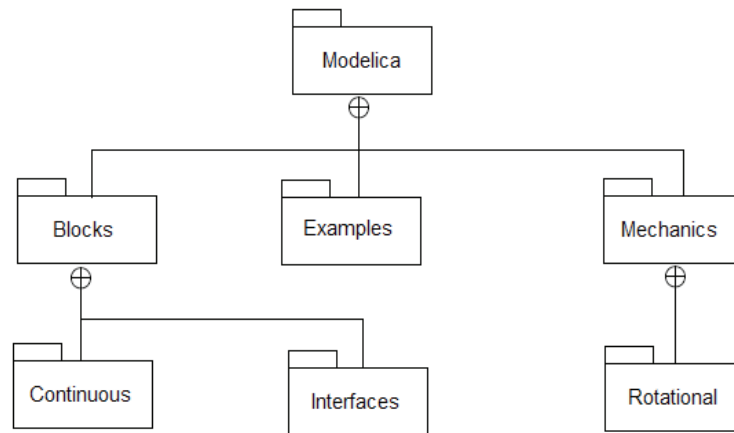


Figure 10-4. Package hierarchy modeling.

10.3.1.4 Parametric Diagrams

SysML defines Constraint blocks which specify mathematical expressions, like equations, to constrain physical properties of a system. Constraint blocks are defined in the Block Definition diagram and can be packaged into domain-specific libraries for later reuse. There is a special diagram type called Parametric Diagram

which relates block parameters with certain constraints blocks. The Parametric Diagram is included in ModelicaML without any modifications.

The Modelica class behavior is usually described by equations, which also constrain Modelica class parameters, and have a domain-specific usage. SysML constraint blocks are less powerful means of domain model description than Modelica equations. Since models in Modelica are expressed by equations, definition complexity of Constraint blocks with parameters for each of equations may results in limited use for Modelica designers. However, grouping constraint blocks into libraries can be useful for system engineers who use Modelica and SysML. SysML Parametric diagram may be used during the initial design phase, when equations related to a class are being identified using Parametric Diagrams and finally associated (via an Equation Diagram) with a Modelica class or set of classes.

10.3.1.5 Equation Diagrams

As was stated previously, model behavior in Modelica is primarily expressed by equations, see Figure 10-5. Compared to traditional programming constructs such as assignment statements and control structures, equations do not prescribe a certain data flow direction. The order in which equations appear in a model, do not influence their meaning and semantics. The only requirement for a system of equations is that it should be solvable. For further details about Modelica equations, see Chapter 3, section 3.2.

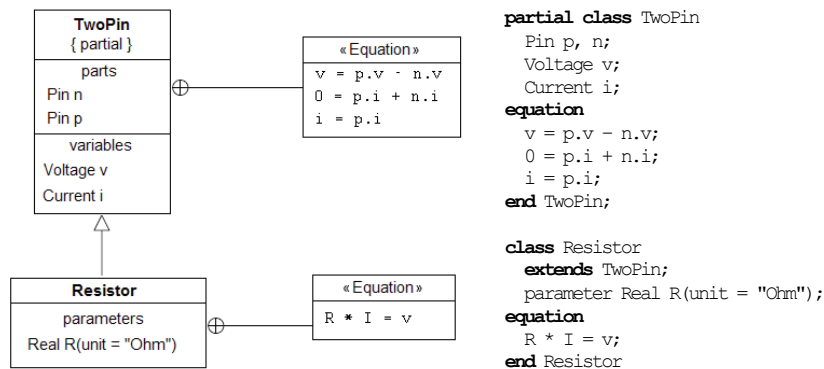


Figure 10-5. Equation modeling example with a Modelica Class Diagram.

Besides simple equality equations, Modelica allows other kind of equations be presented within a model. For each of such kind of equations (i.e. when/if/initial equations) ModelicaML defines a graphical construct. It's up to designer to decide whether to use simple equations block representation or specific construct for equation modeling. Algorithm sections are modeled similar to equations, as text.

10.3.1.6 Simulation Diagram

ModelicaML introduces a new diagram type, called Simulation Diagram, used for simulation modeling. Simulation is usually performed by a simulation tool which allows parameter setting, variable selection for output and plotting. The Simulation Diagram may be used to store any simulation experiment, thus helping to keep the history of simulations and its results.

When integrated with a modeling and simulation environment, a simulation diagram may be automatically generated by a simulation tool. Figure 10-6 shows an example of a Simulation Diagram. The Simulation Diagram provides the following facilities:

- Support for simulation planning.
- Structured presentation of parameter passing and simulation results.
- Running simulations directly from the Simulation Diagram.
- The Simulation Diagram may be generated by a simulation tool.
- Association of simulation results with requirements from a domain expert.
- Additional documentation e.g. by: Note, Problem Rationale text boxes of SysML
- Support for storing model simulation history.

The Simulation Diagram introduces new diagram elements: “Parameter” element and two stereotyped dependency associations, “simParameter” and “simResults”.

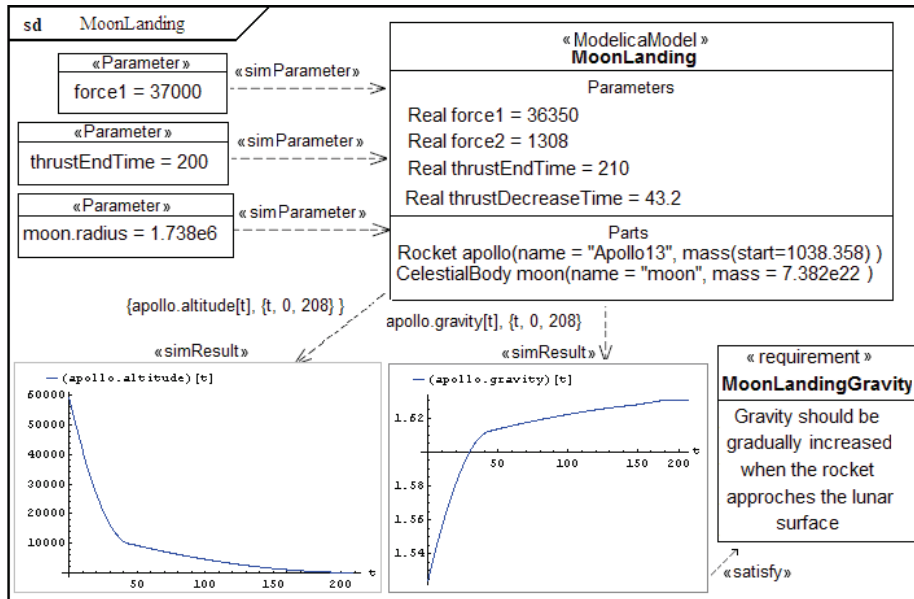


Figure 10-6. Simulation diagram example.

10.3.1.7 Requirement Diagrams

Requirement diagrams have been included in ModelicaML from SysML without any modification. Requirement Diagrams have several attributes: ID, Level, Status, Name and Description. Requirements support hierarchical modeling, i.e., more specific requirements can be derived from more general ones.

Requirements can be linked to any other ModelicaML element via `satisfies` or `satisfiedBy` relations. Any tool implementing the ModelicaML profile can be used to build, query, trace, and manage requirements.

The Modelica language does not support a standard requirements representation. Since the ModelicaML profile supports requirements, we need a way to save these requirements in a Modelica file. Requirements can be represented in Modelica in several ways which we will describe in detail in the next section.

10.3.1.8 Other Diagram Types

Other SysML diagram types such as Use Case Diagram, Activity Diagrams and Allocations, and State Machine Diagrams are included in ModelicaML without modifications. ModelicaML reuses Sequence Diagrams from SysML and changes the semantics of message passing. Modelica doesn't support method declaration within a single class but supports declaration of functions as a restricted class type. In the case of ModelicaML, each lifeline (message passing) represents a Modelica class including block, model, and function restricted classes. Thus, functions are presented as lifelines, and call to a function is modeled as an arrow pointing to it from the caller class (from an algorithm section only). Message name is optional in this case.

10.4 The ModelicaML Integrated Design Environment

Eclipse (Eclipse.Foundation 2001-2008 [29]) is an open source framework for creating extensible integrated development environments (IDEs). One of the goals of the Eclipse platform is to avoid duplicating common code that is needed to implement a powerful integrated environment for development of software. By allowing third parties to easily extend the platform via the plugin concept, the amount of new code that needs to be written is decreased.

For the development of our prototype we used several Eclipse frameworks:

- *EMF – Eclipse Modeling Framework* (Eclipse.Foundation 2008 [30]) is an Eclipse framework for building domain-specific model implementations. The EMF implementation is based on Meta Object Facility (MOF) standard and implements the “Essential MOF” (EMOF) part of a standard. EMF is used by the GMF and UML2 frameworks.
- *GMF – Graphical Modeling Framework* (Eclipse.Foundation 2008 [32]) provides a generative component and runtime infrastructure for developing

graphical editors based on EMF and GEF (Eclipse.Foundation 2008 [31]). GMF consists of tooling, generative and runtime parts, depends on the EMF and GEF frameworks and also on other EMF related tools.

- *The UML2 Eclipse Meta-Model Implementation.* The UML2 Eclipse project is an EMF based implementation of a UML2 meta-model for the Eclipse Platform to support development of UML modeling tools. The UML2 project doesn't aim to provide any graphical modeling or diagram interchange capabilities as it only implements UML abstract syntax. UML2 Tools focus on editing capabilities.

10.4.1 Integrated Design and Development Environment

The Modelica Development Tooling (MDT) (Pop et al. 2006 [131]) and Chapter 8, is part of the OpenModelica system and provides an environment for working with Modelica projects. The following features are available:

- Browsing support and wizards for creating Modelica projects, packages, MSL.
- Syntax color highlighting, syntax and semantic checking.
- Code assistance for packages and function calls
- Support for MetaModelica meta-programming extensions to standard Modelica
- Debugging support for Modelica and MetaModelica algorithmic sections.

We have extended the MDT plugin with a design view to facilitate ModelicaML integration. The ModelicaML integrated design environment where SysML/ModelicaML diagrams are created is shown in Figure 10-7. It consists of a diagram file browser (left), diagram editor (middle), tool palette (right), properties editor (bottom) and a diagram outline (bottom left).

The *Project Browser* lists all Modelica Class and Internal Class diagram files of a project together with existing Modelica files.

The *Diagram Editor* is a tool where diagrams can be created and graphical elements laid out. It has the following graphical features: Graphical elements like Modelica Class or Model can be picked up from a *Tool palette* and created on a Diagram editor pane in a drag-and-drop way. Elements in a palette are grouped by Standard tools (zooming, note, etc), Nodes and Links elements. The tool palette for Modelica Class and Internal Class diagram contains different sets of elements.

The *Property Editor* can be used for changing the properties of the object selected on the diagram editor pane. Property elements vary depending on a type of a chosen object.

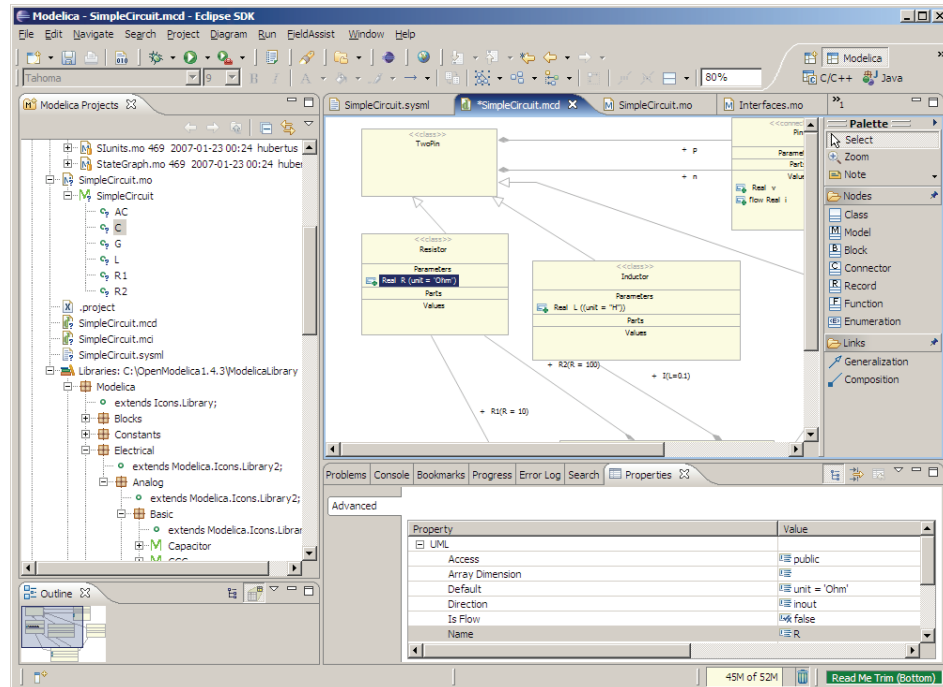


Figure 10-7. ModelicaML Eclipse based design environment with a Class diagram.

The ModelicaML diagrams can be automatically generated from Modelica source files. The integrated tool can also generate Modelica source code from ModelicaML diagrams. However, the implementation of the Modelica code generation and ModelicaML diagram generation is, at the moment, in an experimental stage. In the current implementation ModelicaML diagrams are saved both in Modelica form and also the XMI dialect written to XML files. Further work is needed to save diagram position information within Modelica source code as annotations.

10.4.2 The ModelicaML GMF Model

The Eclipse editor is created from a GMF model of the ModelicaML profile. The model describes the existing elements and their properties. As an example, in Figure 10-8 we present the Requirement Diagram element and its properties.

From the GMF model an editor that supports common operations on that model is automatically generated. The generated code can be extended to deal with issues specific to ModelicaML.

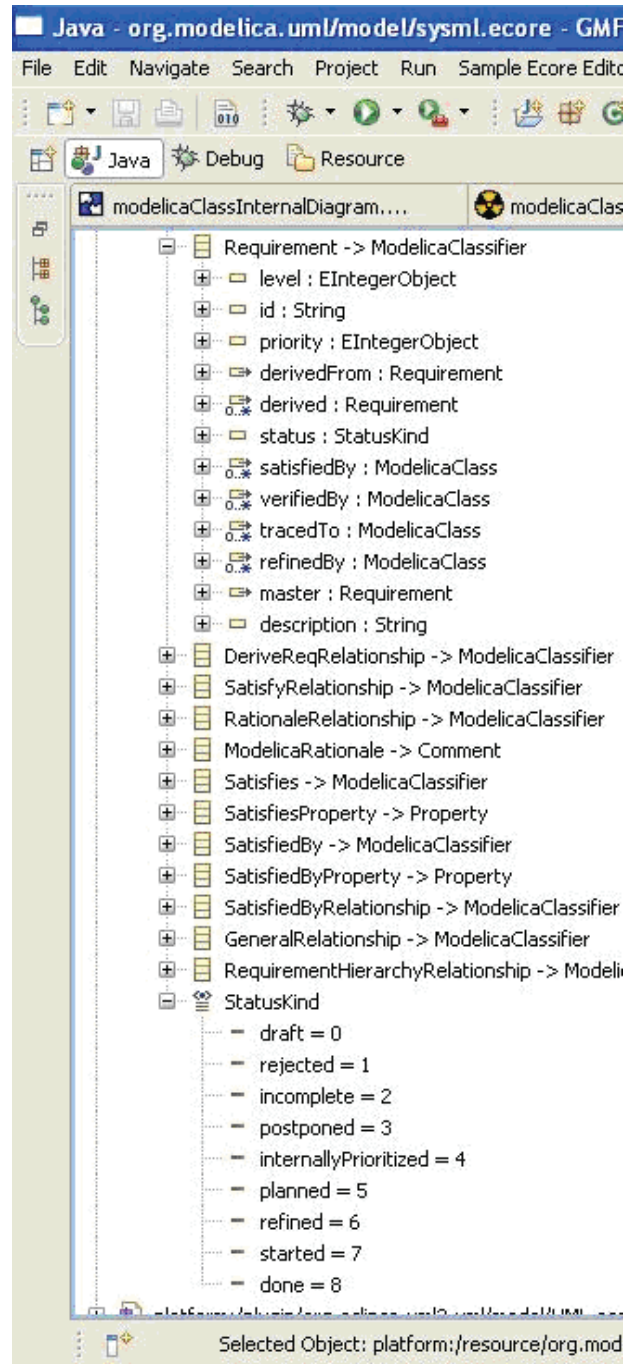


Figure 10-8. ModelicaML GMF Model (Requirements)

10.4.3 Modeling with Requirements

The ModelicaML/MDT Eclipse environment supports modeling with requirements. The following functionality is available in the development environment:

- Hierarchies of requirements can be created.
- Requirements can be traced during the development process.
- Requirements can be queried with respect to any of their attributes.

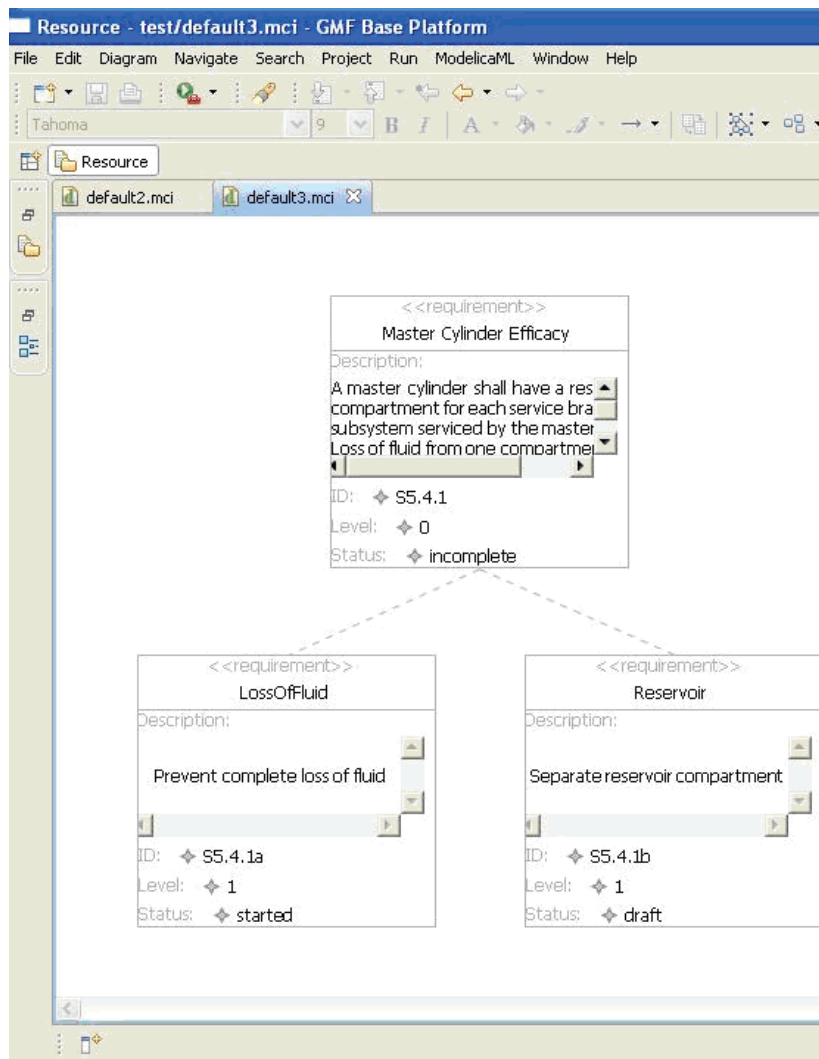


Figure 10-9. Modeling with Requirement Diagrams.

Examples of modeling with Requirement Diagrams are presented in Figure 10-9 and in the Appendix section of this chapter.

10.5 Representing Requirements in Modelica

While the default storage of ModelicaML diagrams is within XML files, our goal is to be able to save this information within normal Modelica files. Having this information within Modelica code would make it accessible to other Modelica and UML tools. Using this information tools could provide additional functionality. For example Modelica tools could display the inheritance hierarchy of a library, display the requirements for a specific class, etc.

To find the best way to encode the ModelicaML diagram information within Modelica we have experienced with several ways of encoding the Requirement Diagrams.

10.5.1 Using Modelica Annotations

A requirement could be saved as an annotation in the following way (the top requirement from Figure 10-9):

```
type RequirementStatus =
  enumeration(Incomplete, Draft, Started);

annotation(
  Requirement(
    id="S5.4.1",
    level=0,
    status=RequirementStatus.Incomplete,
    name="Master Cylinder Efficacy",
    description="A master cylinder..."));
```

The problem with Modelica annotations is that they can only be present at specific places within code and they are usually tied to a class definition. Because requirements are usually cross cutting is impossible to represent all requirements as such annotations. Another problem using annotations is the representation of hierarchies of requirements. Linking of a class with a requirement via a `satisfy` relation could be possible using Modelica `extends`, but would be cumbersome.

10.5.2 Creating a new Restricted Class: requirement

A requirement could be saved as a standard `class` marked with an `isRequirement` annotation or alternatively using a new restricted class in the following way (also in the Figure 10-10 diagrams in the Appendix):

```
requirement R1
  String name="Master Cylinder Efficiency";
  String id="S5.4.1";
  Integer level=0;
  RequirementStatus status=
```

```
        RequirementStatus.Incomplete;  
        String description="A master cylinder  
                           shall have...";  
  
    end R1;
```

Now, we can use extends over requirements to build hierarchies:

```
    requirement R2  
    extends R1;  
    String name"Loss Of Fluid";  
    String id="S5.4.1a";  
    Integer level=1;  
    RequirementStatus status=  
        RequirementStatus.Started;  
    String description="Prevent complete  
                      loss of fluid";  
  
    ...  
end R2;
```

To link requirements to Modelica elements one can use annotations:

```
    model BreakSystem  
        annotation(satisfy=R1);  
    ...  
end BreakSystem;
```

We believe that the best way to encode requirements within Modelica would be to create a new restricted class. Using this new class, requirements can be fully modeled in Modelica.

We will propose in the Modelica Association to introduce the `requirement` restricted class into the Modelica specification. In Modelica, the requirement class could also have equation or algorithm sections that impose constraints to be verified against the class linked with the requirement.

10.6 Conclusion and Future Work

In this chapter we have presented the ModelicaML profile and its prototype integration in the OpenModelica MDT Eclipse plugin. To our knowledge this is the first comprehensive Modelica-UML-SysML integrated environment for product design.

UML Statecharts and Modelica have previously been combined, see e.g.(Ferreira and Oliveira 1999 [38], Nordwig 2002 [107]). SysML is rather new but it has already been adopted for system on chip design (Vanderperren and Dehane 2005 [168]) evaluated for code generation (Vanderperren and Dehane 2006 [169]), and extended with bond graphs support (Turki and Soriano 2005 [156]).

The support for Modelica in ModelicaML allows precisely defining, specifying and simulating physical systems. Modelica provides the means for defining behavior for SysML block diagrams while the additional modeling capabilities of

SysML provides additional modeling and specification power to Modelica (e.g. requirements and inheritance diagrams, etc).

We are currently working on finalizing the implementation details of the Eclipse ModelicaML prototype and releasing the first version for evaluation as part of the OpenModelica environment. Additional functionality such as synchronization of ModelicaML diagrams and Modelica code, storage of ModelicaML information within new Modelica annotations, etc. is also planned.

10.7 Appendix

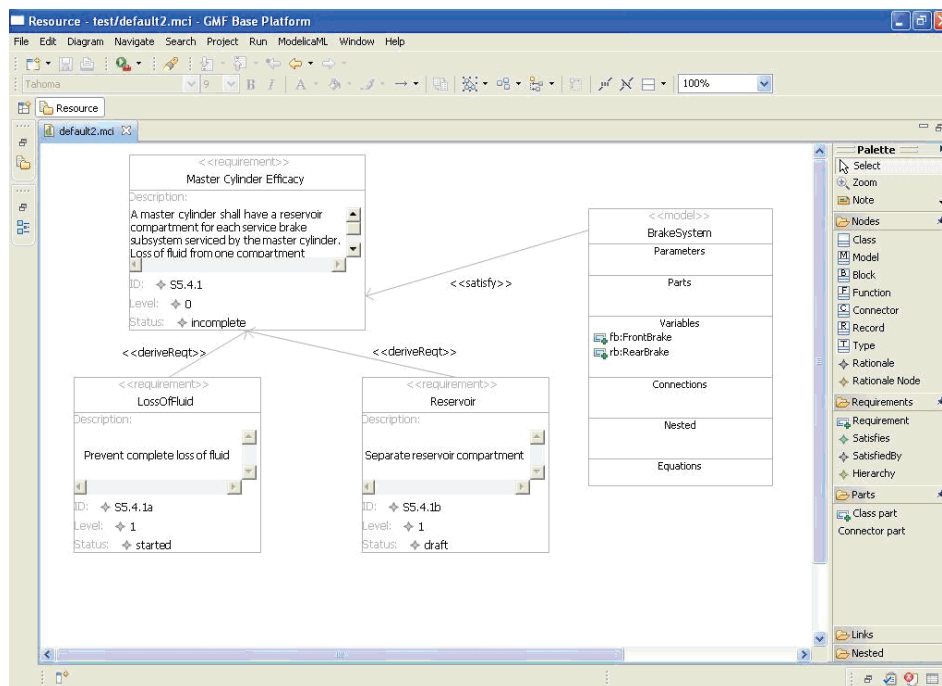


Figure 10-10. Modeling with requirements (Requirements palette).

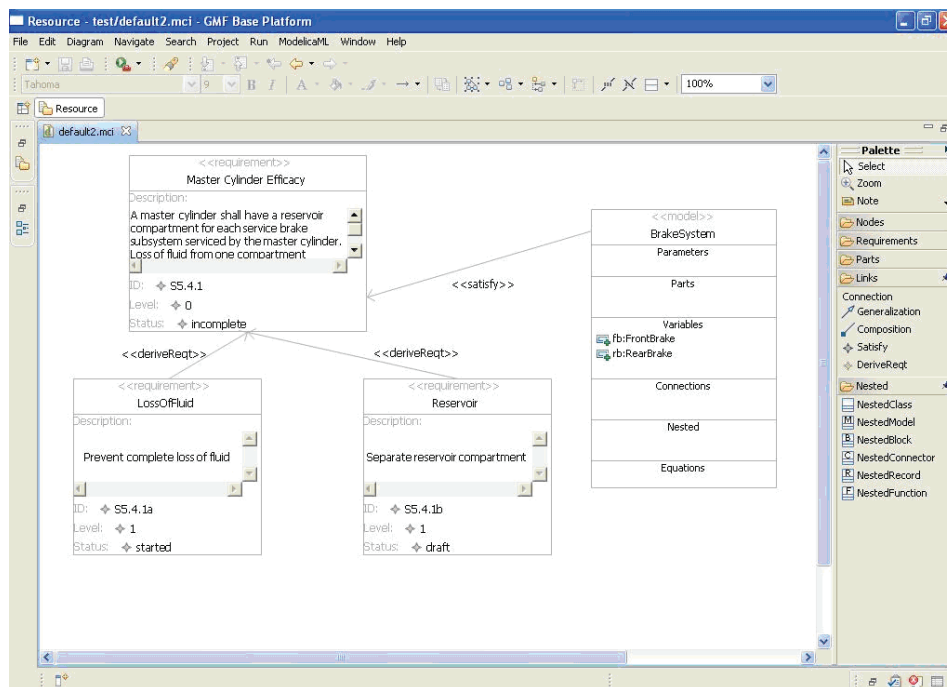


Figure 10-11. Modeling with requirements (Connections).

Chapter 11

An Integrated Framework for Model-driven Product Design and Development Using Modelica

This chapter presents our work in the area of model-driven product development processes. The focus is on the integration of product design tools with modeling and simulation tools. The goal is to provide automatic generation of models from product specifications using a highly integrated set of tools. Also, we provide the designer with the possibility of selecting the best design choice, verified through (automatic) simulation of different implementation alternatives of the same product model. To have a flexible interaction among various tools of the framework an XML representation of the Modelica modeling language called ModelicaXML is used. For efficient search in a large base of simulation models the Modelica Database was designed.

11.1 Introduction

Designing products is a complex process. Highly integrated tools are essential to help a designer to work efficiently. Designing a product includes early design phase product concept modeling and evaluation, physical modeling and simulation and finally the physical product realization. For conceptual modeling and physical modeling and simulation available tools provide advanced functionality. However, the integration of such tools is a resource consuming process that today requires large amounts of manual, and error prone work. Also, the number of physical models available to the designer in the product concept design phase is typically quite large. This has an impact on the selection of the best set of component choices for detailed product concept simulation.

To address these issues we have integrated new product concept design tools with physical modeling and simulation tools in a framework for product design. In our proposed framework, the product concept design phase of the product development process is based on Function-Means tree decomposition (Andreasen 1980 [3]). This phase is implemented in a first version of a prototype tool called FMDesign, (Johansson and Krus 2005 [73]) developed in cooperation with the Machine Design Group, IKP, Linköping University.

As an example of Function-Means tree decomposition we give a landing function in an airplane. This function can be represented by two different means: hydraulic landing gear or electric landing gear. Each of the two alternatives can be selected and configured to simulate its properties.

Starting from FMDesign tool, our integration work extends the framework in two ways:

1. Providing a *Selection and Configuration Tool* that helps the designer to choose a specific implementation for the means in the function-means tree from a Modelica model/ component database. This tool also provides component configuration and has links to a Modelica standard based simulation environment for component editing.
2. Providing an *Automatic Model Generation Tool* that helps the designer to choose the best implementation from different design choices by evaluation through simulation of automatically generated models of candidate product concepts. If the designer is not pleased with the results, he/she can either implement new models for the components that did not perform in the desired way or reiterate in the design process and choose other alternatives for implementing different functions in the product, or change the configuration parameters for models at deeper levels of detail.

The chapter is structured as follows: The next section (section 11.2) presents an overview of our proposed framework. Section 11.3 enters in the details of the framework components and their interaction. Section 11.4 presents our conclusion and future work.

The presented system has similarities with the Schemebuilder tool (Bracewell and D.A.Bradley 1993 [16]) and Modelith framework (Johansson et al. 2002 [72], Larsson et al. 2002 [81]). However our work is more oriented towards the design of advanced complex products that require systems engineering, and targeted to the simulation modeling language Modelica, which to our knowledge has more expressive power in the areas of our research, than many tools for systems engineering that are currently widely used. For details on Systems Engineering, see (INCOSE 1990-2008 [70]).

11.2 Architecture overview

The architecture of our extended framework is presented in Figure 11-1. The entire product concept design process is iterative.

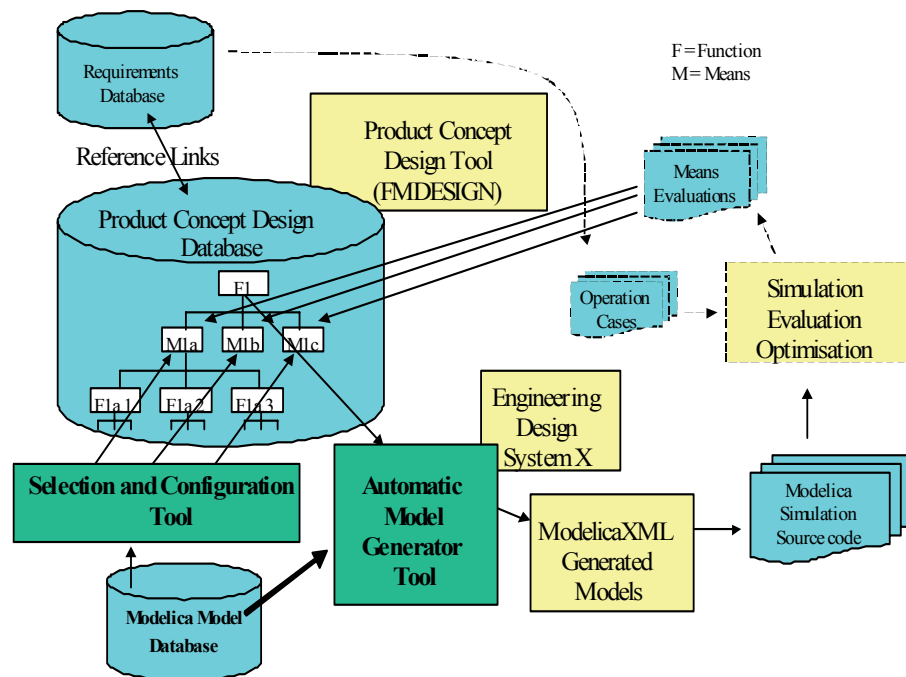


Figure 11-1. Design framework for product development.

Starting from requirements for a product the designer will use the FMDesign prototype for modeling alternative product concepts. The knowledge base for designing a product is organized into function-means trees. A function in the product can be realized by alternative means. A product concept is a set of means that document selected solution alternatives for implementing the functions in a product concept. Example of a function is "Actuator Power Supply", with means "Hydraulic Power Supply" or "Electrical Power Supply". Means must be implemented by (physical) components arranged in a bill-of-material like tree of implementation objects.

One can roughly say that a means and its implementation are the same, but at different levels of detail. Implementation objects (not shown in the figure) may represent existing component products on the market or manufactured components. Implementation objects carry data that is important for the product concept design,

and references to more detailed design information like CAD-drawings, simulation models etc. Some (physical components) may implement several means, like an aircraft wing that creates lift and stores fuel.

To map suitable simulation model implementations to a means, the designer would use the Modelica Database query facility provided by the *Selection and Configuration Tool*. This tool also provides configuration of the simulation components and uses the desired Modelica environment for component editing.

When the product concept design phase of the product is sufficiently complete, the designer can generate code for simulation from the implementation tree using the *Automatic Model Generator Tool*. The generator will output models (different versions for different product concepts) in ModelicaXML. From ModelicaXML the models are translated to Modelica to be simulated. The designer can review the simulation results in any available Modelica tools and then selects (in FMDesign) the desired model alternative for the implementation. If the designer sees that some means do not perform in the desired way, a customized simulation model can be built, or a search conducted for more alternatives for that specific means.

11.3 Detailed framework description

In this section we present the tools from our proposed framework. Also, we briefly explain in each section how they interact.

11.3.1 ModelicaXML

Modelica is translated to ModelicaXML (Pop and Fritzson 2003 [126]) using a Modelica parser (Figure 11-2).

ModelicaXML represents an XML serialization of the Abstract Syntax Tree of the Modelica language obtained after the parsing. In our framework, ModelicaXML is used as an interchange format between the different design tools.

The advantages of having an alternative representation for Modelica in XML are:

- Flexible interaction and translation between different types of physical modeling languages and modeling tools. Also, easy generation of model documentation.
- Basic search and query functionalities over models.
- Easy transformation and composition of models Chapter 13 and (Pop et al. 2004 [133]).

For more information on ModelicaXML the reader is referred to Chapter 12, (Pop and Fritzson 2003 [126]) and (Fritzson 2004 [44]).

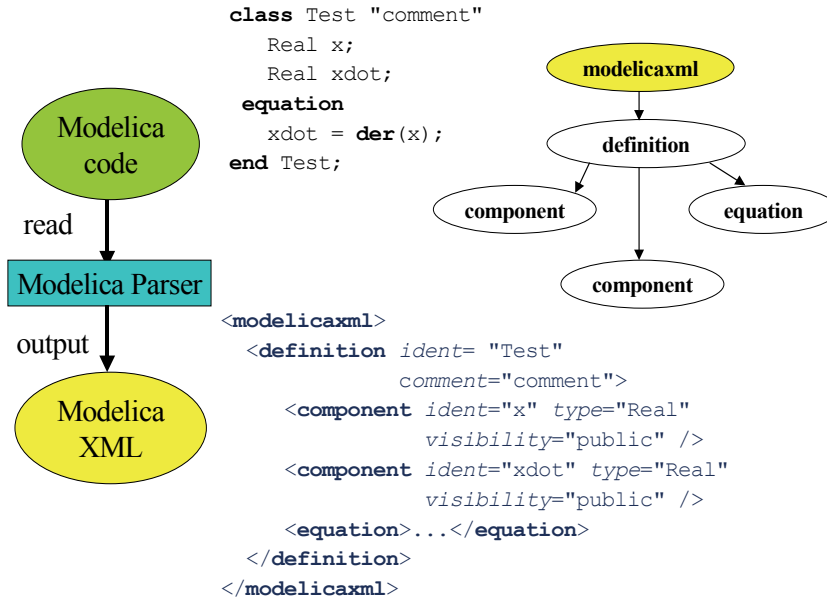


Figure 11-2. Modelica and the corresponding ModelicaXML representation.

11.3.2 Modelica Database (ModelicaDB)

The features of the Modelica language and Modelica tools has made easy for designers to create models. Also, the Modelica community has a growing code-base. In order to cope with interoperability between Modelica and other modeling languages we first developed ModelicaXML. However, scalability and efficient search features for XML require extensive skills in vendor specific products. To quickly get such features without taking on that huge learning effort, we have designed the Modelica Database (ModelicaDB).

The Modelica Database is populated with Modelica models and libraries by importing their ModelicaXML representation. The UML model of this database is presented in the appendix (section 11.5). For space reasons we use a somewhat customized compressed graphical representation of UML class diagrams, where inheritance is represented with a box between the class name and attributes box, where inherited super classes are preceded with a "->". For details on UML see (OMG [115]).

Here we briefly explain the most important structures. They are tightly coupled with the Modelica structure (Fritzson 2004 [44], Pop and Fritzson 2003 [126]):

- *Modelica Repository*: contains several Modelica Models.

- *Class*: A class represents the fundamental model element from the Modelica language. It can include several *Component* clauses, *Equation* and *Algorithm* statements. The component sections can be declared as public or protected in order to provide only the desired interface to the outer world. Specifying that the equation or algorithm sections are only active at the initialization phase they can be declared as initial.
- *Component*: used to define parameters, variables, constants, etc to be used inside a class.
- *Equations and Algorithms* are used to specify the behavior for a class.

In the product design framework the role of ModelicaDB is to provide searching and organization features of a large base of simulation models. This base grows with every product model developed or with the import of additional simulation models from other sources (i.e. the Modelica community). For example, if we want to obtain all the models that have certain parameter names we have to search in the database for all classes that have a component with the attribute `variabilityPrefix` set to "**parameter**" and have the specified name. These searches will be integrated in FMDesign using dialogs and completely transparent for the user.

11.3.3 FMDesign

The FMDesign (Figure 11-3) prototype tool (Johansson and Krus 2005 [73]) helps the designer in creating product specifications using function-means trees.

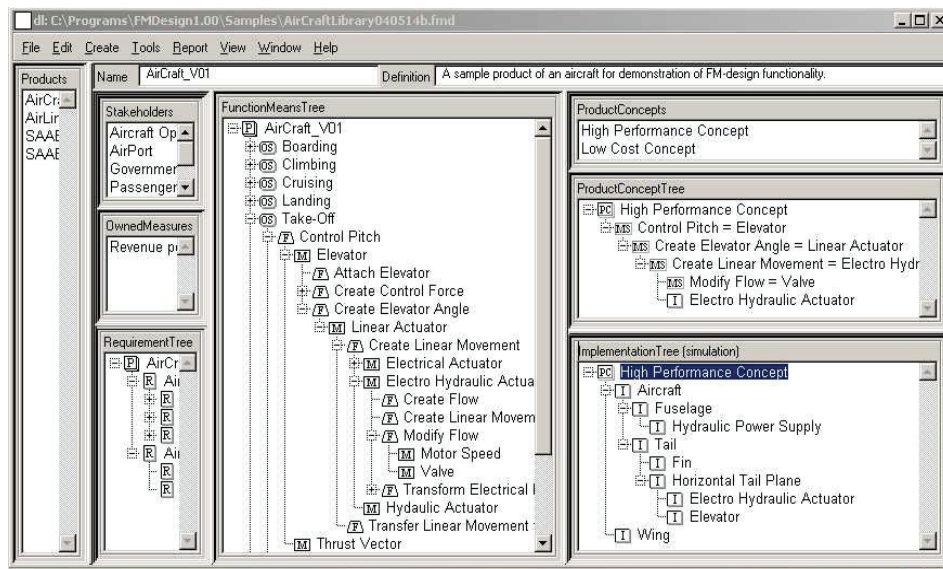


Figure 11-3. FMDesign – a tool for conceptual design of products.

The created product model is stored in a product design library for later reuse. Throughout the product concept design process the designer can use the existing concepts stored in the product design library in order to model the desired product. A somewhat simplified meta-model of the information structure edited in FMDesign is presented as an UML class diagram in the appendix (section 11.5).

In the framework, FMDesign is the central front-end to specific components. FMDesign delegates searches in the ModelicaDB using the *Selection and Configuration Tool* and it uses the *Automatic Model Generation Tool* to generate the models for simulation.

As we can see in Figure 11-3, the work area is divided into several parts:

- *Products*: Here products are created, deleted and selected. When a product is selected, the trees owned by it and described below, are displayed.
- *Requirements Tree*: in this view the requirements for a product can be specified.
- *Function-Means Tree*: in this view the designer can define the operation states, functions, their alternative means etc, of the selected product.
- *Product Concepts*: Allows creating, deleting and selecting product concepts.
- *Product Concept Tree*: displays the currently selected Product Concept Tree, and allows the user to select which means that will implement different functions in the product, using drag-drop. Selected means can be customized for the current product concept by overriding the default values for its design variables owned by a selected means.
- *Implementation Tree*: displays and provides functionality for editing one of many configurable Implementation Trees for the currently selected product concept. These implementation trees organize the implementation objects that represent and refer to more detailed models of physical objects, functional models, simulation models, geometrical layout models etc, and organize them into trees that are useful for interfacing with tools later in the product development process.

We only use the Implementation Tree of type simulation to generate the Modelica simulation model for a product. The Implementation Tree of type geometrical can be used in the visualization of the product.

11.3.4 The Selection and Configuration Tool

The Selection and Configuration Tool extends the framework by adding integrated search capabilities in FMDesign. The tool is coupled with the Implementation Tree for a Product Concept. The designer uses the selection tool to search (query) the Modelica Database for desirable simulation components to implement a certain means. An implementation object in the simulation implementation tree represents

the selected simulation component. Simulation component to means mapping reflects the various design choices made by the designer. In this way, the designer can experiment with different simulation component implementations at various level of detail for a specific means. When choosing alternatives for a specific means the designer has two possibilities: to browse the repository of simulation models classified according to physical concepts or to use the search dialog. The search dialog provides the following functionality:

- Textual/pattern search of components, search for a component in a specific physical domain, search for a component with specific parameters.
- Adding/deleting a product concept specific means to simulation component mapping where the simulation component is referred from an implementation object.

After building the means-component mappings the designer can choose to edit or configure components by using the configuration dialog that provides the following functionality:

- Set implementation component parameters or parameters ranges.
- Edit the simulation component in the desired Modelica environment and use the edited component, which is also automatically added to the Modelica Database.

11.3.5 The Automatic Model Generator Tool

The Automatic Model Generator Tool provides the second extension of the framework.

The model generator tool has as input the Implementation Tree (Figure 11-3, lower right) of a product and as output the complete simulation model with the alternative design choices.

The automatic model generator traverses the Implementation Tree of a Product Concept and outputs ModelicaXML models by choosing the combination of selected components for means. The generated models are then translated to Modelica for means evaluation through simulation. To simulate the models any tool supporting Modelica compiler can be used.

After the simulation of the generated models, the results are used as feedback for the designer. Using this feedback the designer can then choose the best-suited model, based on the simulation results.

11.4 Conclusions and Future Work

As future work we want to explore the use of ontologies for product concept design and for the classification of the available component libraries. The languages

developed by the Semantic Web (Berners-Lee et al. 2001 [12], SemanticWebCommunity [146], W3C [162], W3C [164], W3C [165]) community will be used. Research efforts based on this standard are integrating experience of many promising research areas, for instance declarative rules, which still lack a vendor neutral exchange formats for industrial applications. The semantic web standard lacks important functionality for quality assurance and other necessary functionality, which today is implemented in commercial products, but will open up for sharing of important research results with industry in collaborative environments. Also we would like to improve the *Automatic Model Generator Tool* by using parts of the composition and transformation framework described in Chapter 13 and (Pop et al. 2004 [133]).

In the future we want to provide automatic evaluation through simulation of the generated models based on the constraints collected from the Product's Requirement Tree.

11.5 Appendix

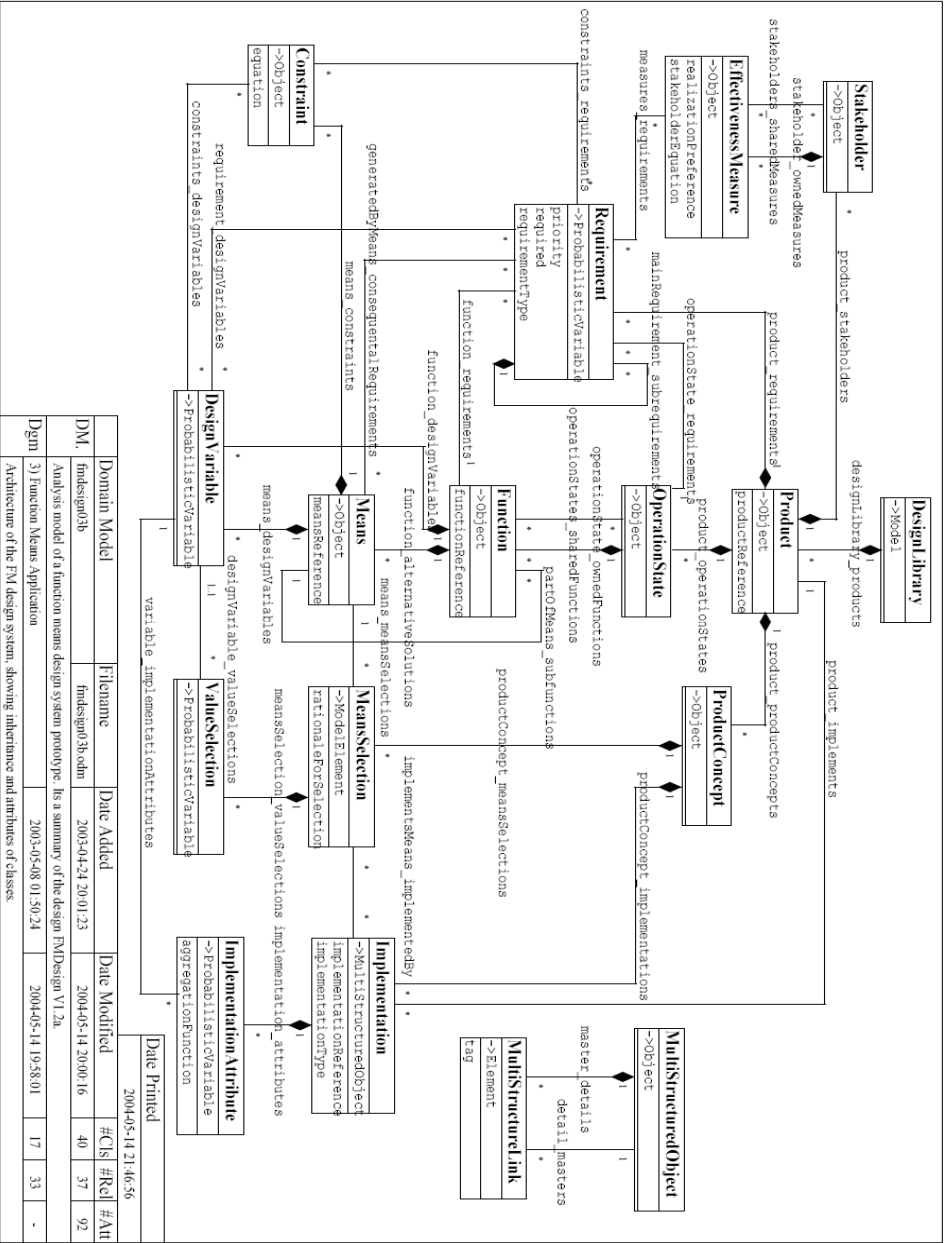


Figure 11-4. FMDesign information model.

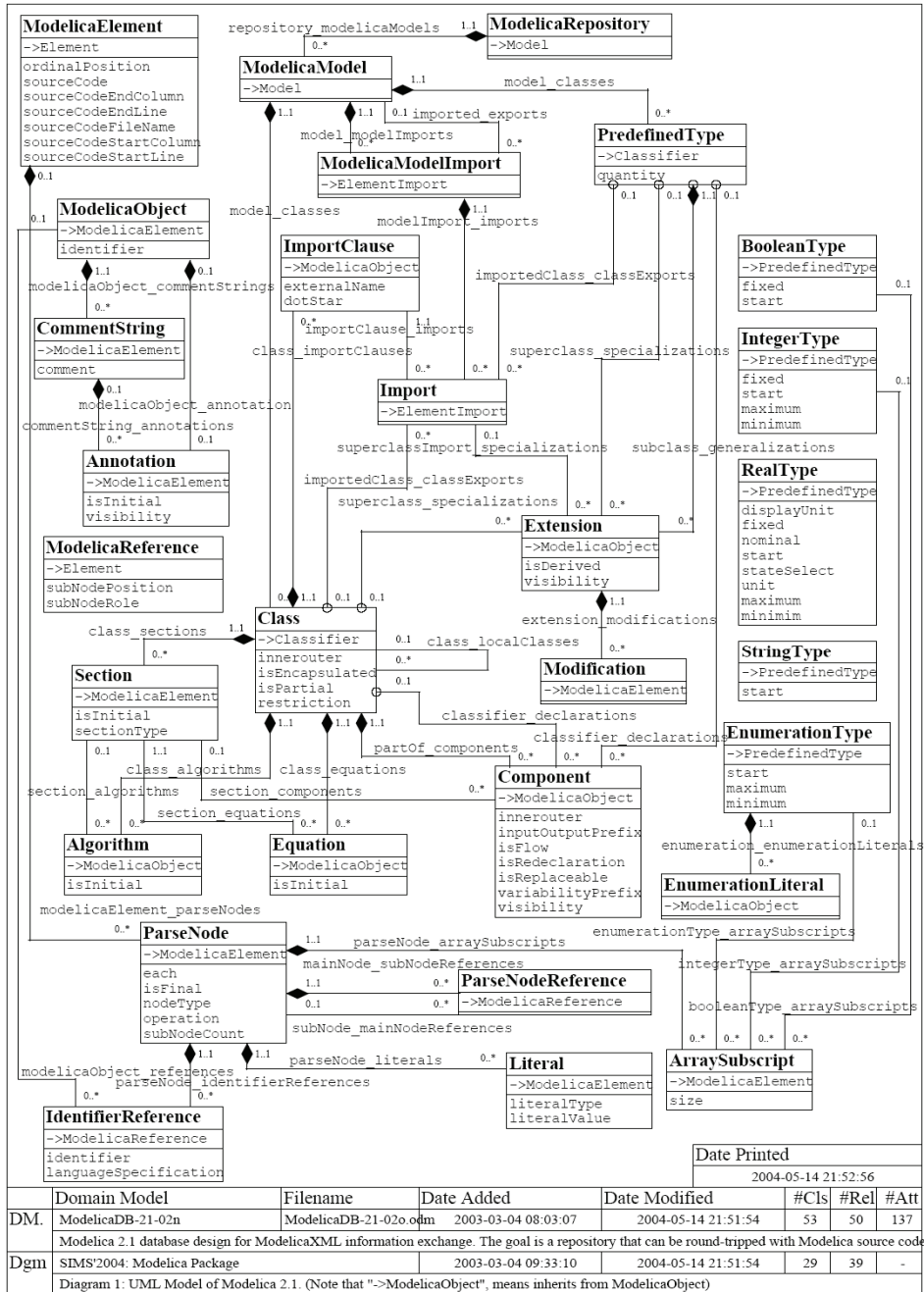


Figure 11-5. ModelicaDB meta-model.

Part V

Meta-programming and Composition of EOO Languages

Chapter 12

ModelicaXML: A ModelicaXML Representation with Applications

This chapter presents the Modelica XML representation with some applications. ModelicaXML provides an Extensible Markup Language (XML) alternative representation of Modelica source code. The language was designed as a standard format for storage, analysis and exchange of models. ModelicaXML represents the structure of the Modelica language as XML trees, similar to Abstract Syntax Trees (AST) generated by a compiler when parsing Modelica source code. The ModelicaXML (DTD/XML-Schema) grammar that validates ModelicaXML documents is introduced. We reflect on the software-engineering analyses one can perform over ModelicaXML documents using standard and general XML tools and techniques. Furthermore we investigate how we can use more powerful markup languages, like the Resource Description Framework (RDF) and the Web Ontology Language (OWL), to express some of the Modelica language semantics.

12.1 Introduction

The structure of a Modelica model can be derived from the source code representation, by using a Modelica compiler front-end (the lexical analyzer and the parser).

The compiler front-end takes the source code representation and transforms it to *abstract syntax trees* (AST), which are easier to handle by the rest of the compiler. As pointed out in (Badros 2000 [11]), a clear disadvantage of this procedure is the need of embedding a compiler front-end in every tool that needs access to the structure of the program. Writing such a front-end for an evolving and advanced language like Modelica is not trivial, even with the support of automated tools like Flex (GNU 2005 [58])/Bison (GNU 2005 [56]) or ANTLR (Parr 2005 [116]).

To overcome these problems, a standard, easily used, structured representation is needed. ModelicaXML is such a representation that defines a structure similar to abstract syntax trees using the XML markup language.

This representation provides more functionality than a typical C++ class library implementing an AST representation of Modelica:

- Declarative query languages for XML can be used to query the XML representation.
- The XML representation can be accessed via standard interfaces like Document Object Model (DOM) (W3C [157]) from practically any programming language.

The usages of the ModelicaXML representation for Modelica models, combined with the power of general XML tools, will ease the implementation of tasks like:

- Analysis of Modelica programs (model checkers and validators).
- Pretty printing (un-parsing).
- Translation between Modelica and other modeling languages (interchange).
- Query and transformation of Modelica models.

Although ModelicaXML captures the structured representation of Modelica source code, the semantics of the Modelica language cannot be expressed without implementing specific XML-based tools. To address this issue we have investigated the benefits of using other markup languages like the Resource Description Framework (RDF) and the Web Ontology Language (OWL). These languages, developed in the Semantic Web Community (Berners-Lee et al. 2001 [12], SemanticWebCommunity [146], W3C [162]), are used to express semantics of data in order to be automatically processed by machines. We believe that using such technology for Modelica models would enable several applications in the future:

- Models could be automatically translated between modeling tools.
- Models could become autonomous (active documents) if they are packaged together with the operational semantics from the compiler, and therefore, they could be simulated in a normal browser.
- Software information systems (SIS) could more easily be constructed for Modelica, facilitating model understanding and information finding.
- Model consistency could be checked using Description Logic (DL) (Baader et al. 2003 [10], DescriptionLogicsWebsite [24]).
- Certain models could be translated to and from the Unified Modeling Language (UML) (OMG [115]).

The chapter is structured as follows: Related work is presented in Section 12.2. Modelica, XML and the ModelicaXML Document Type Definition (DTD) are discussed in Section 12.3. In Section 12.4 we present the software-engineering tasks one can perform on the ModelicaXML representation using XML tools and technologies. Section 12.5 investigates the use of RDF and OWL for representing semantics of Modelica models. Conclusions, future research directions and summary of the work are presented in Section 12.6.

12.2 Related Work

In the field of general programming languages, JavaML (Badros 2000 [11]) has been developed as structured representation of Java source code. JavaML emphasizes the power of such structured representation when leveraging XML tools. When it comes to domain specific modeling languages, there are several (Björn et al. 2002 [14], Freiseisen et al. 2002 [40], Larsson et al. 2002 [81]) approaches to specifying models in XML. These approaches deal with model transformation, exchange and management (regarding adaptation to already existing simulation tools) or with code generation from the intermediate XML representation to C++. Our interest focuses more on providing flexible and general software-engineering tooling support for the Modelica programmer. For this purpose the ModelicaXML is covering the full Modelica language, including algorithm sections and expression operators. Furthermore, we consider more powerful markup languages for defining some of the Modelica static semantics and we discuss future use of such Semantic Web technologies.

12.3 Modelica XML Representation

In section 2.6 we briefly introduced the concepts of XML and DTD. Here we give an example of a Modelica model with its ModelicaXML representation.

12.3.1 ModelicaXML Example

To introduce the Modelica XML representation, we give a Modelica example and show its corresponding representation as ModelicaXML.

Elements are in **bold**, attributes are in *italic* and entities are using underline throughout this section, except from Modelica keywords.

```
class SecondOrderSystem
  parameter Real a=1;
  Real x(start=0); Real xdot(start=0);
  equation
    xdot=der(x);
    der(xdot)+a*der(x)+x=1;
end SecondOrderSystem;
```

For ease of presentation, a ModelicaXML document is split into several parts, each representing a more nested level. The ellipses from one level are detailed in the next level:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE program SYSTEM
          "ModelicaXML.dtd">
<program within="...">
```

```

    <definition ident="SecondOrderSystem"
                restriction="class">
        ...
    </definition>
</program>

```

The root element is a Modelica **program**. The child elements of **program** are a sequence of **definition** elements and an optional *within* attribute (see Figure 12-1, section 12.3.2 for schemata).

```

    <definition ident="SecondOrderSystem"
                restriction="class">
        <component>...</component>
        ...
        <equation>...</equation>
        ...
    </definition>

```

The **definition** element can have **import**, **extends**, **elements**, **equation**, or **algorithm** as sub-elements. In our case we only have **component** (i.e., variable) and **equation** sub-elements inside **definition** (see Figure 12-2, section 12.3.2 for schemata).

```

    <component ident="a" type="Real"
                variability="parameter"
                visibility="public">
        <modification_equals>
            <real_literal value="1"/>
        </modification_equals>
    </component>
    ...
    <component ident="x"
                type="Real"
                visibility="public">
        <modification_arguments>
            <element_modification>
                <component_reference ident="start"/>
                <modification_equals>
                    <real_literal value="0"/>
                </modification_equals>
            </element_modification>
        </modification_arguments>
    </component>

```

The first **component** (i.e., variable, see Figure 12-3, section 12.3.2 for schemata) has the *variability* attribute set to "parameter" as in "parameter Real a=1;". The second **component** declaration (i.e., variable) in the example represents the "Real x(start=0);" line from our Modelica class. All components have the *visibility* attribute set to "public". The last **component** is similar to the second **component** and is not presented.

```

<equation>
  <equ_equal>
    <component_reference ident="xdot"/>
    <call>
      <component_reference ident="der"/>
      <function_arguments>
        <component_reference ident="x"/>
      </function_arguments>
    </call>
  </equ_equal>
</equation>

```

Equations are enclosed in the **equation** element (see Figure 12-4, section 12.3.2 for schemata)

The equation section of the `SecondOrderSystem` model describes two equations. The first equation is quite straightforward. Equality is represented by an **equ_equal** element with two elements inside. The right-hand side is a function call (using the **call** element) to a derivative and the left hand side is a component reference represented with the element with the same name. The second equation below is more complex. It has function calls represented using the **call** element, binary operations (see Figure 12-6, section 12.3.2 for schemata) such as **add**, **mul** for addition (+) and multiplication (*). The **component_reference** elements denote variable references. For the function calls, the arguments are specified using the element **function_arguments** that can contain expressions, named arguments or for indices.

```

<equation>
  <eq_equal>
    <add><call><component_reference ident="der"/>
      <function_arguments>
        <component_reference ident="xdot" />
      </function_arguments>
    </call>
    <add><component_reference ident="x"/>
      <mul>
        <component_reference ident="a"/>
        <call>
          <component_reference ident="der"/>
          <function_arguments>
            <component_reference ident="x" />
          </function_arguments>
        </call>
      </mul>
    </add>
  </add>
  <integer_literal value="1"/>
</eq_equal>
</equation>

```

ModelicaXML Schemata are explained in the next section.

12.3.2 ModelicaXML Schema (DTD/XML-Schema)

When designing the ModelicaXML representation we started from the Modelica grammar. We simplified the common cases to compact the XML representation without loss of information or structure. The Modelica DTD/XML-Schema has a rather close correspondence to the Modelica grammar with the following exceptions: attributes are used to make the XML representation more concise and the DTD/XML-Schema jumps over some non-terminals from the Modelica grammar to make the XML representation more compact.

The OpenModelica Project parser for Modelica source code, written in ANTLR (Parr 2005 [116]), was changed to output the ModelicaXML representation. There are many components in the OpenModelica Project that use the ANTLR Modelica parser. Using ModelicaXML such tools can be decoupled from this parser. One clear advantage of this approach is that only one parser is maintained and future Modelica language extensions or modifications could be easily integrated.

For presentation purposes we translated our first DTD implementation to XML-Schema using XML Spy (Altova 2008 [2]). The purpose of this translation was to generate pictures from the XML-Schema. Also, another reason was to have schemata files in both formats for future use. Perhaps, the DTD variant will be discontinued in the future because the XML-Schema is more widely used now.

All elements from our schema have the optional attributes from the *location* entity (which are *sline*, *scolumn*, *eline* and *ecolumn*) and the *info* attribute, which can be used to store additional information. These *location* attributes are used to generate a mapping between key elements in our schema and the Modelica source code representation. In the following we present some of the important elements from the DTD/XML-Schema.

The content of our ModelicaXML root element, namely **program** is depicted in Figure 12-1. Inside the root element we can have none or several **definition** elements. The optional attribute *within* can be used inside a **program** element. The rounded corner boxes on the line connecting two elements can be sequence (like in Figure 12-1) or choice (like in the bottom part of Figure 12-2).

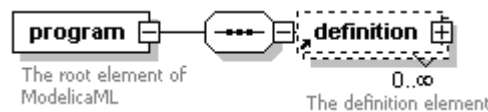


Figure 12-1. The **program** (root) element of the ModelicaXML Schema.

The required attributes for **definition** are *ident* and *restriction* (which can have one of the "class", "model", "record", "block", "connector", "type", "package", or "function" values). Optional attributes are *final*, *partial*, *encapsulated*, *replaceable*, *innerouter*, *visibility* (one of "public", "protected" values) and *string_comment*.

The **definition** element is detailed in Figure 12-2. Presented in the picture at the bottom are the **derived** element (that handles constructs of the type "class X

= Y;") and the **enumeration** element used to declare enumeration types. The upper part of Figure 12-2 shows the other allowed elements that can appear inside the **definition** element. All the elements in the upper part have the *visibility* attribute, taking one of the "public" or "protected" values. The *visibility* attribute values are stating the "public" or "protected" part from the Modelica source code. We can see that the **definition** element is recursive, which allows the declaration of classes inside classes.

The **definition** element can contain **import**, **extends**, **external**, **equation**, **algorithm**, **annotation** and **component** elements. The latter can use **constrain** element for handling statements like "**type** X=Y **extends** Z;".

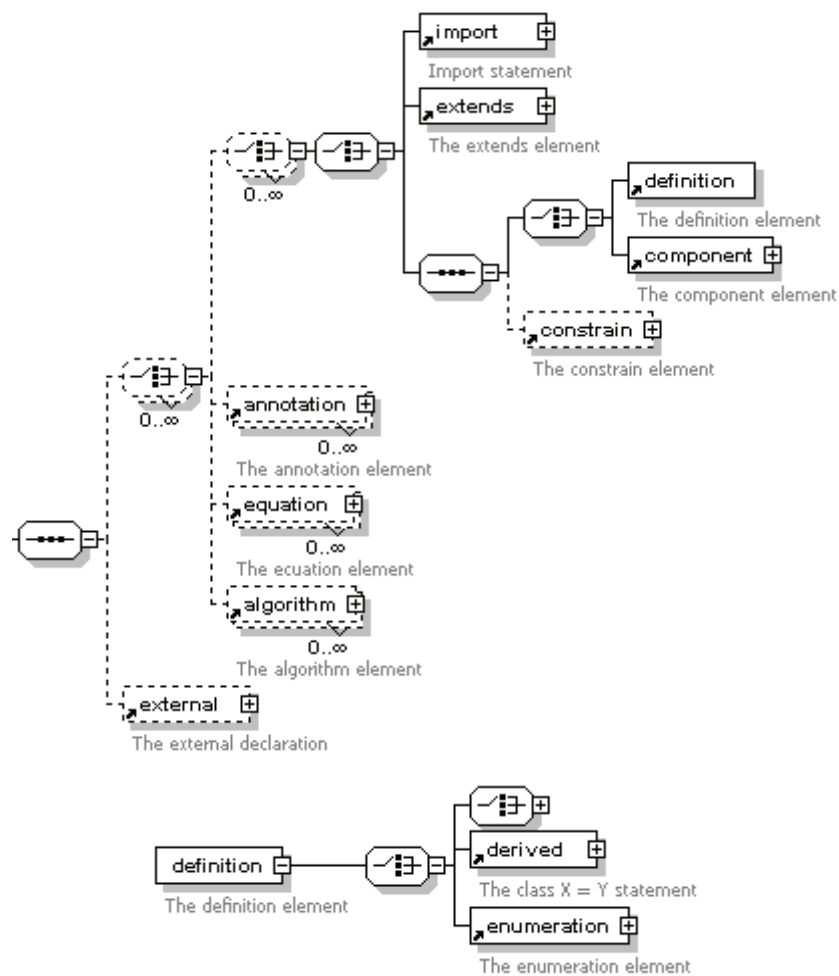


Figure 12-2. The **definition** element from the ModelicaXML Schema.

Component elements, with schemata presented in Figure 12-3, have attributes representing the Modelica type prefix (*flow, variability and direction*), and type name (*type*).

The name of the component is stored in the *ident* attribute. These attributes are important because one can query the ModelicaXML representation for a specific component having desired *type* and *ident*. How XML query languages can be used is explained in section 12.4.

The **type_array_subscripts** element and the **array_subscripts** element are expressing the fact that Modelica array subscripts can be declared either at the type level or at the component level.

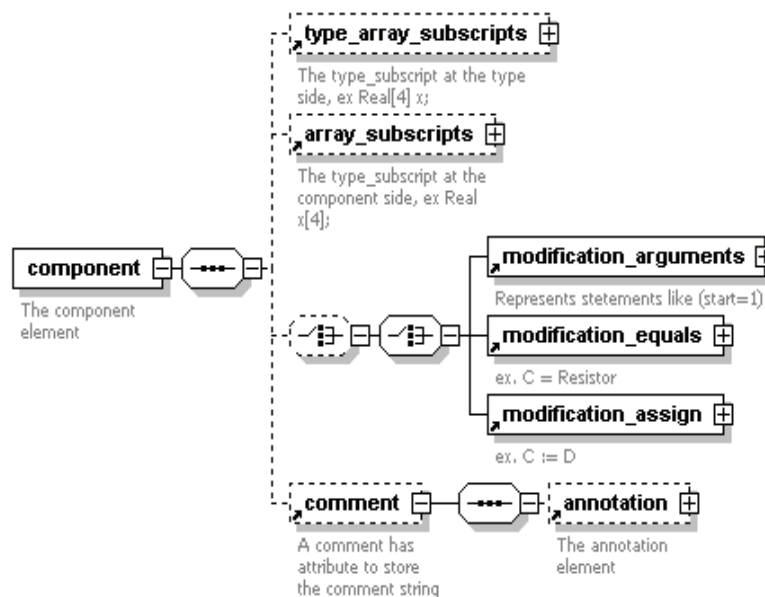


Figure 12-3. The **component** element from the ModelicaXML Schema.

One can use the element **modification_arguments** to further modify the component. Comments for a **component** can be specified with the **comment** element. The elements **modification_equals** and **modification_assign** are used to modify the component; as sub-elements they can have Modelica expressions.

An **equation** element, presented in Figure 12-4, can have *initial* as an attribute to state if it represents a Modelica initial equation.

The content and the structure of the **equation** element are closely following the definition from the Modelica Language Specification. The **equ_connect** element takes component references as arguments here, instead of connect references, as in the version 2.0 of the Modelica Language Specification.

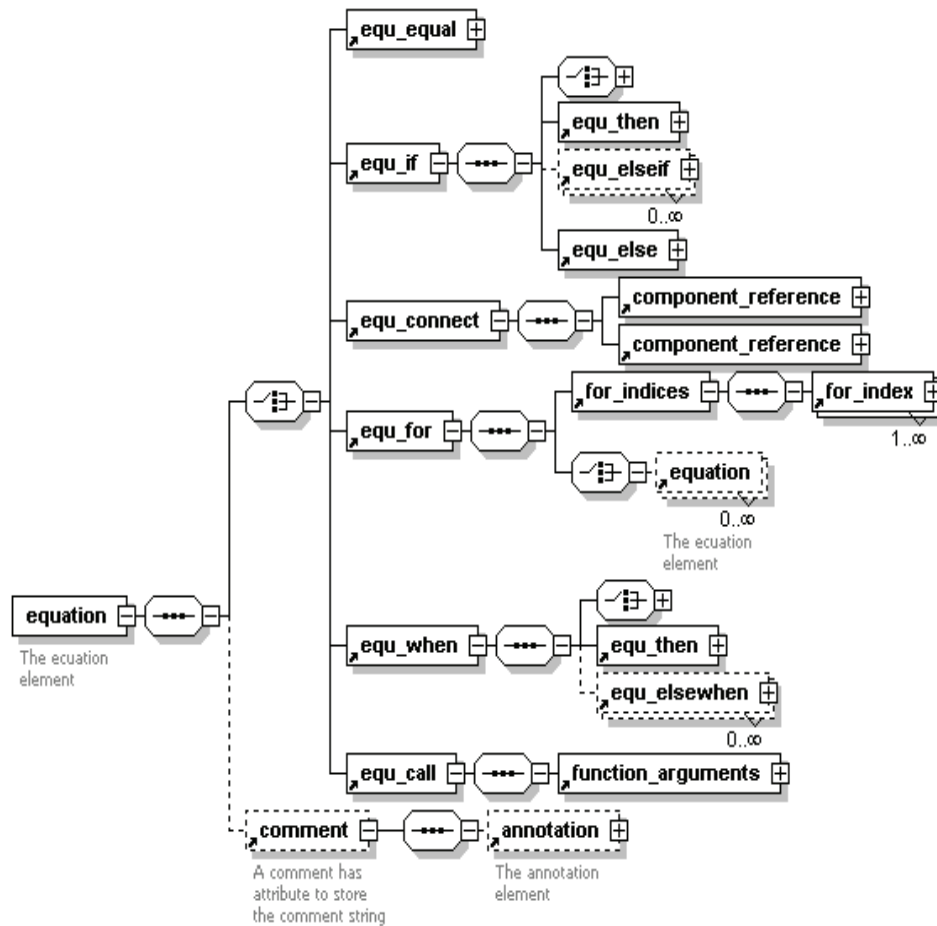


Figure 12-4. The **equation** element from the ModelicaXML Schema.

The collapsed parts from the **equ_if** and **equ_when** elements are the Modelica expressions, detailed in Figure 12-6. The Modelica expressions are present in the collapsed parts of the algorithm elements **alg_if** and **alg_when** and **alg_while**.

The **algorithm** element is presented in Figure 12-5. We point out that the elements **alg_break** and **alg_return** are recently added statements of the algorithm section in the latest version (2.1) Modelica Language Specification.

The elements that can appear in ModelicaXML expressions can be found in Figure 12-6. These are binary operations, literals, component references, array constructions, array operators and logical operations.

The constructs from the ModelicaXML schemata not covered here, along with the full "modelicaXML.xsd" (the XML-Schema version) and "modelicaXML.dtd" (the DTD version), can be found at the OpenModelica Project website.

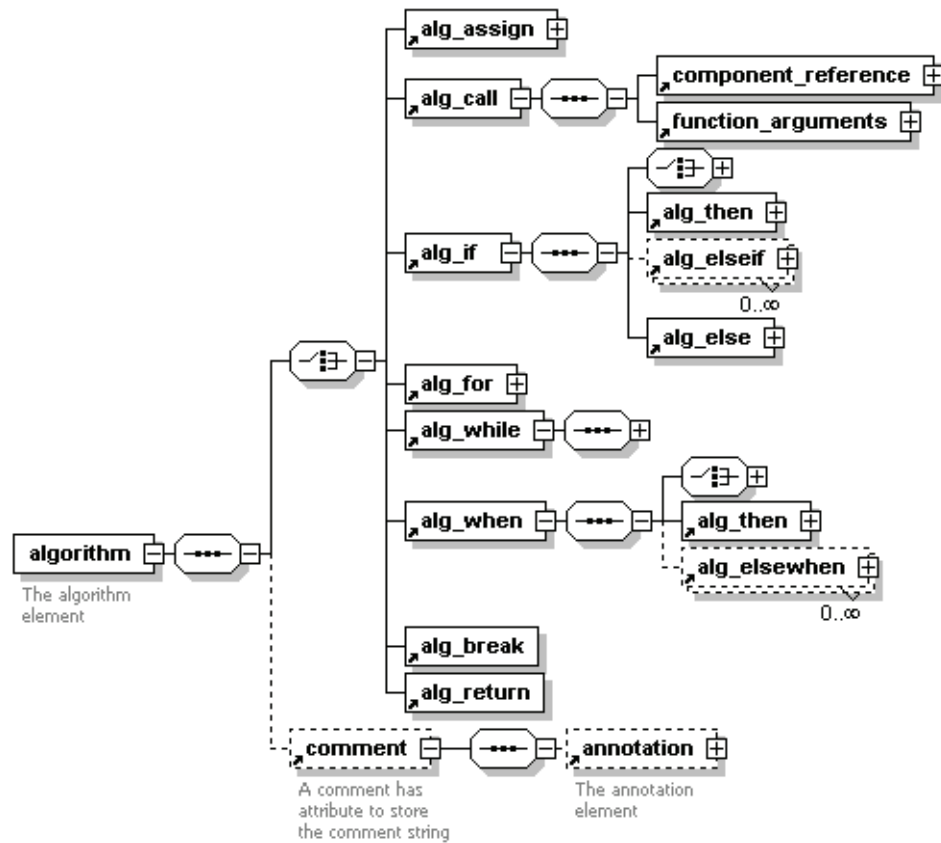


Figure 12-5. The `algorithm` element from the ModelicaXML Schema.

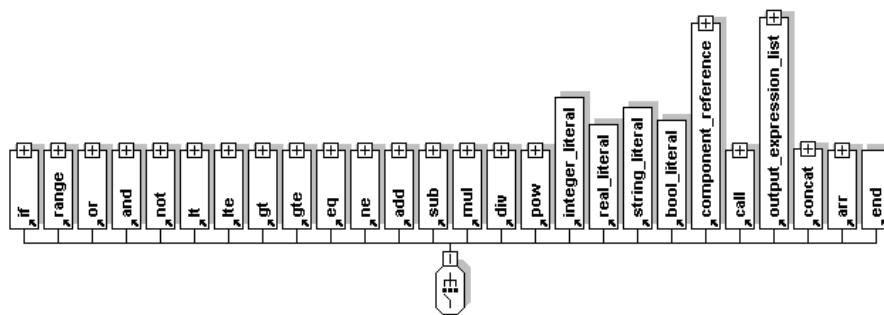


Figure 12-6. The expressions from ModelicaXML schema.

12.4 ModelicaXML and XML Tools

This section introduces various XML tools and explains their usage in conjunction with ModelicaXML. In the following, in different sub-sections we cover: the stylesheet language for transformation (XSLT) (W3C [159]), the query language for XML documents (XQuery) (W3C [166]) and the Document Object Model (DOM) (W3C [157]).

12.4.1 The Stylesheet Language for Transformation (XSLT)

XSL is a stylesheet language for XML. XSLT is the part of XSL that deals with transformation of XML documents.

Using XSLT one can implement pretty printers (un-parsers) that can transform ModelicaXML back into Modelica source code. Alternative transformations could transform ModelicaXML into other general, modeling or markup languages (HTML, XHTML, etc). Transformers that translate other modeling languages (provided that they have an XML representation) into ModelicaXML can also be implemented with XSLT. Using XSLT and ModelicaXML, implementation of HTML documentation generators, similar with what the commercial software Dymola provides, becomes trivial. We cannot provide the HTML documentation generator here because of space reasons, but it will be included in the OpenModelica Project.

We illustrate the usage of XSLT with an example that transforms Modelica code. For this example we assume that Modelica code was already translated to ModelicaXML. After the transformation, one can output the Modelica code from the changed ModelicaXML representation using our "modelicaxml-2modelica.xslt" stylesheet from the OpenModelica Project.

Example of changing a component name, both in the declaration of the component and in the component references:

```
<xsl:stylesheet version="1.0 ...">
  <!-- example of component rename -->
  <xsl:param name="comp_old_name"/>
  <xsl:param name="comp_new_name"/>
  <!-- we echo everything that is not a component or a
  component reference -->
  <xsl:template match="*|@*|text()">
    <xsl:copy>
      <xsl:apply-templates select="*|@*|text()"/>
    </xsl:copy>
  </xsl:template>
  <!-- we match the old component and we output the new name
  -->
  <xsl:template match="component
    [@ident=$comp_old_name]">
    <component ident="{ $comp_new_name }">
```

```

    <xsl:apply-templates/>
  </component>
  <!-- we match the old component reference and we output the
new component name -->
</xsl:template>
<xsl:template match="component_reference
[@ident=$comp_old_name]">
  <component_reference
    ident="{ $comp_new_name }">
    <xsl:apply-templates/>
  </component_reference>
</xsl:template>
</xsl:stylesheet>

```

The XSLT engine is using templates that match on the XML tree structure. The matching is performed by the XPath expression appearing as the value of the *match* attributes. By using **xsl:apply-templates** element we instruct the XSLT engine to apply the rest of the templates on the sub-tree that we already matched. When this stylesheet is applied on our *SecondOrderSystem* example from section 12.3.1 with the parameters "xdot" and "xdot_new" it will change the component name and all the component references of xdot to xdot_new.

XSLT can distinguish between components with the same name defined in different classes by the use of XPath expressions. To rename such occurrences we first match the class in which is defined and then the actual component. This applies for both declarations and component references.

A search-and-replace tool could perform this transformation, but such a tool has no knowledge about the context and it will replace even the occurrences appearing inside comments.

12.4.2 The Query Language for XML (XQuery)

XQuery is a query language similar with what SQL is for relational databases. Using XQuery, one can easily retrieve information from XML documents. The XQuery and XSLT are overlapping in some features, and our example could be implemented in XSLT also.

We give a short example of a query over our "SecondOrderSystem.xml" example from section 12.3.1. In words, "find all parameter components with type Real and show the initialization value":

```

<table border="1">
{
  for $b in
    (document("SecondOrderSystem.xml")/*/
     definition/component)
  where $b/@type = "Real" and
        $b/@variability="parameter"
  return <tr><td>

```

```

        { $b/@* }
        { $b/modification_equals }
      </td></tr>
    }
  </table>

```

We executed this query in the Qexo (GNU 2005 [61]) implementation of XQuery and the result in HTML is as follows:

```

<table border="1">
  <tr><td>
    ident="a" type="Real"
    variability="parameter"
    visibility="public"
    <modification_equals>
      <real_literal value="1" />
    </modification_equals>
  </td></tr>
</table>

```

As expected, the attributes and the set value of the element corresponding to "parameter Real a=1;" from our Modelica example was returned as the answer.

Using XQuery, any types of queries can be asked about the Modelica model. This opens-up the possibility of easily debugging very large models. User interfaces can be implemented to hide the query building from the user. Static type checking can also be implemented as a series of queries on the model, but is not trivial, because the class hierarchy is not explicitly defined in XML.

XQuery uses XPath as sub-language to select the part of tree that matches the XPath expression. In our XML representation one can match an entire component having a specified *ident* attribute. The XPath language can be used to handle scooping.

12.4.3 Document Object Model (DOM)

The Document Object Model (DOM) (W3C [157]) is a standard interface that allows programs to access/update the content, structure and style of XML documents. DOM is similar with a general tree-management library.

There are open-source implementations for DOM APIs in Java, C, C++, Perl, Python and other programming languages.

Any Modelica tool written in various programming languages can use the DOM API to directly access/modify the ModelicaXML representation.

12.5 Towards an Ontology for the Modelica Language

This section investigates the possibility of using the markup languages Resource Description Framework (RDF) (W3C [161]), RDF Vocabulary Description Language (RDFS) (W3C [160]) and OWL (W3C [164], W3C [165]) developed in the Semantic Web (Berners-Lee et al. 2001 [12], SemanticWebCommunity [146], W3C [162]) for development of a Modelica ontology.

An ontology is a description (like a formal specification of a program) of both the objects in a certain domain and the relationships between them. In the context of the Semantic Web there is a layered approach for specifying increasingly richer semantics for the upper layers as in Figure 12-7.

At the bottom, in top of Unicode and Uniform Resource Identifiers (URI) is XML, namespaces (NS) and XML-Schema. XML specifies a term list with no relations. On top of XML comes RDF to define a vocabulary and some relations. RDFS (RDF schema) defines a vocabulary for constructing RDF vocabularies.

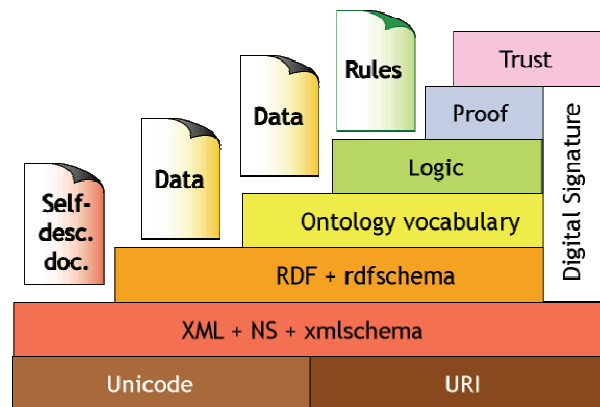


Figure 12-7. The Semantic Web Layers.

The Ontology layer uses languages like OWL to define description logic relationships.

With ModelicaXML we are now only at the XML level! Using RDF we can express graphs and we can model inheritance relationships and place queries over this relation. This can be achieved easily with a smart parser. Using OWL we can place restrictions over relations and concepts and we can reason with inference using Description Logics.

12.5.1 The Semantic Web Languages

This sub-section briefly introduces the Semantic Web Languages: Resource Description Framework (RDF/RDFS) and Web Ontology Language (OWL).

We illustrate the use of Semantic Web Languages by taking a Modelica model and its representation in OWL.

```

class Body "Generic body"
  Real mass;
  String name;
end Body;
class CelestialBody "Celestial body"
  extends Body;
  constant Real g = 6.672e-11;
  parameter Real radius;
end CelestialBody;
CelestialBody moon(name = "moon",
  mass = 7.382e22, radius = 1.738e6);

Body body_instance(name = "some body",
  mass = 7.382e22);

```

Our Modelica model has two classes (concepts) **Body** and **CelestialBody** the latter being a subclass of the former (by using "**extends**" statement).

The encoding in OWL is as follows:

```

<?xml version="1.0" ?>
<rdf:RDF
  <!-- namespaces declaration -->
  xmlns=".../inheritance.owl#"
  xmlns:modelica=".../inheritance.owl#"
  xml:base=".../inheritance.owl">
  <owl:Ontology rdf:about=".../inheritance.owl" />

  <!-- define Body -->
  <owl:Class rdf:ID="Body">
    <rdfs:label>Generic Body</rdfs:label>
  </owl:Class>
  <!-- define mass -->
  <owl:DatatypeProperty rdf:ID="mass">
    <rdfs:domain rdf:resource="#Body"/>
    <rdfs:range rdf:resource="XMLSchema#float"/>
  </owl:DatatypeProperty>
  <!-- define name -->
  <owl:DatatypeProperty rdf:ID="name">
    <rdfs:domain rdf:resource="#Body"/>
    <rdfs:range
      rdf:resource="XMLSchema#string"/>
  </owl:DatatypeProperty>

  <!-- define CelestialBody -->
  <owl:Class rdf:ID="CelestialBody">
    <rdfs:label>Celestial Body</rdfs:label>
    <rdfs:subClassOf

```

```

        rdf:resource="#Body" />
<!-- cardinality restriction on the g constant:
      one and only one in CelestialBody -->
<rdfs:subClassOf>
  <owl:Restriction>
    <owl:onProperty rdf:resource="#g"/>
    <owl:cardinality rdf:datatype
      ="XMLSchema#nonNegativeInteger">
      1
    </owl:cardinality>
  </owl:Restriction>
</rdfs:subClassOf>
</owl:Class>
<!-- define g -->
<owl:DatatypeProperty rdf:ID="g">
  <rdfs:domain
    rdf:resource="#CelestialBody"/>
  <rdfs:range
    rdf:resource=" XMLSchema#float"/>
</owl:DatatypeProperty>
<!-- define radius -->
<owl:DatatypeProperty
  rdf:ID="radius">
  <rdfs:domain rdf:resource="#CelestialBody"/>
  <rdfs:range rdf:resource=" XMLSchema#float"/>
</owl:DatatypeProperty>
<!-- instance declaration of CelestialBody -->
<CelestialBody rdf:ID="moon">
  <name rdf:datatype="XMLSchema#string">moon</name>
  <mass rdf:datatype="XMLSchema#float">7.382e22</mass>
  <radius rdf:datatype="XMLSchema#float">1.738e6</radius>
  <g rdf:datatype="XMLSchema#float">6.672e-11</g>
  <g rdf:datatype="XMLSchema#float">
    intentional error
    (string is not float)
  </g>
</CelestialBody>

<!-- instance declaration of Body -->
<Body rdf:ID="body_instance">
  <name rdf:datatype="XMLSchema#string">
    some body
  </name>
  <mass rdf:datatype="XMLSchema#float">
    7.382e22
  </mass>
  <-- intentional error (Body does not have a radius) -->
  <radius rdf:datatype="XMLSchema#float">1.738e6</radius>
</Body>
</rdf:RDF>

```


In the OWL representation of the Modelica model we first define **Body** as being an **owl:Class** with "Generic body" as label. The attributes of **Body**, namely: **mass** and **name** are represented as **owl:DatatypeProperty**. The datatype is a binary relation having a **range** (type) and a **domain** (in our case the **Body** concept). As **range** we use the datatypes from XML-Schema, in our case, for **mass** we use "float" and for **name** we use "string".

The class **CelestialBody** is defined as **owl:subclassOf** the **Body** class according to the "extends" statement from our Modelica model. As an OWL feature in the definition of **CelestialBody** we show a local cardinality restriction placed on the **g** relation. This means that in the instances of **CelestialBody**, the **g** component has to appear exactly once. The representation of **g** or **radius** components is similar to the representation of **mass** or **name**.

The **moon** instance of the **CelestialBody** class sets the values of the components. We intentionally added the **g** component twice and with a wrong type. We also declare an instance of the **Body** class that has a **radius** component (which is an error).

To verify the model, our file: "inheritance.owl" was fed into an OWL Validator (Rager 2003 [136]).

The validator, as expected, reports the following errors:

- For the **g** component that has a string as value: "Range Type Mismatch. Use of this property implies that object is of type XMLSchema#float".
- For the **radius** component in the **body_instance** declaration: "Domain Type Mismatch. Use of this property implies that subject is of type #CelestialBody. Subject is declared type [Body]"
- For the **moon** instance: "Cardinality Violation. Resource #moon violates the cardinality restriction on class #CelestialBody for property #g. Resource has 2 statements with this property. Maximum cardinality is 1".

The OWL language has more constructs than our example has covered. One can consult the OWL website (W3C [164], W3C [165]) for more details.

12.5.2 The roadmap to a Modelica representation using Semantic Web Languages

In the example above we have presented a small ontology that models our Modelica model, consisting of both classes and instances. With a clever parser, such ontologies could be generated from Modelica libraries and then used for composing Modelica models.

The roadmap to a Modelica representation in OWL has the following steps:

- Define an RDFS vocabulary for Modelica source code constructs. Such a vocabulary should include concepts like class, model, record, block, etc.

- Transform the Modelica libraries in their OWL representation using the above vocabulary.
- An OWL validator can then check the correctness of both the concepts and the instances of these concepts.

At the end of this roadmap we would have Modelica represented in OWL. The future benefits of such a representation were underlined in the Introduction section. Here, we briefly explain how they could be achieved.

12.5.2.1 The Autonomous Models

In the OpenModelica Project, the Modelica compiler is built from the formal specification (expressed in Natural Semantics (Kahn 1988 [75])) of the Modelica Language. This specification can be compiled to executable form using the Relational Meta-Language (RML) system (PELAB 1994-2008 [117], Pettersson 1995 [120], Pettersson 1999 [122]). The rules from Natural Semantics could be translated to OWL or RuleML (RuleML [139]) and shipped together with the model. Using the rules from the model a normal browser could compile and simulate the Modelica model. We assume that the platform should have a C compiler.

12.5.2.2 The Software Information System (SIS)

Having the Modelica ontologies that model the source code one could use the approach detailed in (Welty 1995 [172]) and build the domain model of the problem. Merging them together would result in a Software Information System.

Using such a Software Information System, users can ask queries about the Modelica source code concepts (components, classes, etc) that are classified according to the domain model concepts of the problem.

12.5.2.3 Model consistency could be checked using Description Logic

Modelica models represented in OWL (Description Logics) can be fed into a reasoning tool like FaCT (Horrocks [67]) or Racer (Haarslev et al. 2004 [63]) for consistency checking.

Moreover, such support would be of great help to the Modelica library designers that could formally check relevant properties of the class hierarchies.

The checks one can do using Description Logics on the Modelica OWL representation are the following:

- Ensure that the classes and the class hierarchy are consistent (ensure that a class can have instances and is not over-constrained).
- Find the explicit relations between classes, regarding for example sub-typing or equivalence.

12.5.2.4 Translation of Models to/from Unified Modeling Language

The UML language has its XML representation called XMI (OMG [111]). Translation from Modelica models conforming to a Modelica ontology to XMI could be possible using XSLT.

12.6 Conclusions and Future work

We have presented the ModelicaXML language and some applications of XML technologies. We have shown that there are some missing capabilities with such XML representation and we addressed some of them. We have presented a roadmap to an alternative representation of Modelica in OWL and the use of representation together with the Semantic Web technology.

As future work, we consider completing the ModelicaXML with the definition of all the intermediate steps representations from Modelica to flat Modelica and further to the code generation. This complete representation would allow various open-source tools to act at these formally defined levels, independent of each other. More information could be added in the future to such XML representation, like: model configuration, simulation parameters, etc.

Further insights in the direction of Semantic Web Languages and their use to express Modelica semantics are necessary. Compilation in both directions between OWL and the Relational Meta-Language (RML) is worth considering.

Chapter 13

Composition of XML dialects: A ModelicaXML case study

This chapter investigates how software composition and transformation can be applied to domain specific languages used today in modeling and simulation of physical systems. More specifically, we address the composition and transformation of the Modelica language. The composition targets the ModelicaXML (described in the previous chapter) dialect which is the XML representation of the Modelica language. By extending the COMPOST concrete composition layer with a component model for Modelica, we provide composition and transformation of Modelica. The design of our COMPOST extension is presented together with examples of composition programs for Modelica.

13.1 Introduction

Commercial Modelica tools such as MathModelica and Dymola as well as open-source tools such as the OpenModelica system can be used for modeling with the Modelica language. While all these tools have high capabilities for compilation and simulation of Modelica models, they:

- Provide little support for configuration and generation of components and models from external data sources (databases, XML, etc).
- Provide little support for security, i.e. protection of “intellectual property” through obfuscation of components and models.
- Do not provide automatic composition of models using a composition language. This would be very useful for automatic generation of models from various CAD products.

- Provide little support for library designers (no automatic renaming of components in models, no support for comparison of two version of the same component at the structure level, etc.).

We address these issues by extending the COMPOST framework with a Modelica component model that acts on the ModelicaXML representation.

The use of XML technology for software engineering purposes is highly present in the literature today. The SmartTools system (Attali et al. 2001 [8], Attali et al. 2001 [9]) uses XML technologies to automatically generate programming environments specially tailored to a specific XML dialect that represents the abstract syntax of some desired language. The use of Abstract Syntax Trees represented as XML for aspect-oriented programming and component weaving is presented in (Schonger et al. 2002 [145]). The OpenModelica System project investigates some transformations on Modelica code like meta-programming (Aronsson et al. 2003 [4]). The bases of uniform composition for XML, XHTML dialect and the Java language were developed in the European project Easycomp (EasyComp 2004 [28]). However, the possibilities of this framework can be further extended and tested by supporting composition for an advanced domain specific language like Modelica.

The chapter is structured as follows. The next section introduces Modelica, ModelicaXML, and COMPOST. Section 13.3 presents our COMPOST extension and its usage through various examples of composition and transformation programs for Modelica. Conclusion and future work can be found in Section 13.4. The appendix, gives the ModelicaXML representation for some of the examples.

13.2 Background

In this section give a short description of the COMPOST framework and present a short Modelica model and its ModelicaXML representation.

13.2.1 Modelica and ModelicaXML

Modelica has a structure similar to the Java language, but with equation and algorithm sections for specifying behavior instead of methods. Also, in contrast to Java, where one would use assignment statements, Modelica is primary an equation-based language. We give a short Modelica model and its ModelicaXML representation:

```
class HelloWorld "HelloWorld comment"
  Real x(start = 1);
  parameter Real a = 1;
  equation
    der(x) = -a*x;
end HelloWorld;
```

In the example we have defined a class called `HelloWorld`, which has two components and one equation. The first component declaration (second line) creates a component `x`, with type `Real`. All Modelica variables have a `start` attribute, which can be initialized using a modification equation like (`start = 1`).

The second declaration declares a so called **parameter** named `a`, of type `Real` and set equal to an integer with value 1. The parameters are constant during simulation; they can be changed only during the set-up phase, before the actual simulation.

The software composition is not performed directly on the Modelica code, but instead, on an alternative representation of it: ModelicaXML (Chapter 13 and (Pop and Fritzson 2003 [126])). As an example, the `HelloWorld` class translated to ModelicaXML would have the following representation:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE modelica SYSTEM "modelica.dtd">
<program>
  <definition ident="HelloWorld" restriction="class"
    string_comment="HelloWorld comment">
    <component visibility="public" type="Real" ident="x">
      <modification_arguments>
        <element_modification>
          <component_reference ident="start"/>
          <modification_equals><integer_literal value="1"/>
        </modification_equals>
        </element_modification>
      </modification_arguments>
    </component>
    <component visibility="public" variability="parameter"
      type="Real" ident="a">
      <modification_equals><integer_literal value="1"/>
    </modification_equals>
    </component>
    <equation>
      <equ_equal>
        <call><component_reference ident="der"/>
          <function_arguments>
            <component_reference ident="x"/>
          </function_arguments>
        </call>
        <sub operation="unary">
          <mul><component_reference ident="a"/>
            <component_reference ident="x"/>
          </mul>
        </sub>
      </equ_equal>
    </equation>
  </definition>
</program>
```

The translation of the Modelica into ModelicaXML is straightforward. The abstract syntax tree (AST) of the Modelica code is serialized as XML using the ModelicaXML format. ModelicaXML is validated against the `modelica.dtd` Document Type Definition (DTD) (W3C [158]). Using the XML representation for Modelica, generation of documentation, translation to/from other modeling languages can be simplified.

13.2.2 The Compost Framework

COMPOST is a composition framework for components such as code or document fragments, with special regard to construction time. Its interface layer called UNICOMP for universal composition provides a generic model for fragment components in different languages and different concrete component models.¹

Components are composed by COMPOST as follows. First, the components, i.e., templates containing declared and implicit hooks, are read from file. Then, a *composition program* in Java applies composition operations to the templates, and transforms them towards their final form. (The transformations rely on standard program transformation techniques.) After all hooks have been filled, the components can be pretty-printed to textual form in a file again. They should no longer contain declared hooks so that they can be compiled to binary form.

13.2.2.1 The notions of components and composition

Fragment-based composition with COMPOST (Aßmann and Ludwig 2005 [7]) is based on the observation that the features of a component can be classified in several dimensions. These dimensions are the language of the component, the model of the component, and abstract component features. The dimensions depend on each other and can be ordered into a layer structure of 5 layers (Figure 13-1):

1. **Transformation Engine Layer.** The most basic layer encapsulates knowledge about the contents of the components, i.e., about the concrete language of the component. Fragment-based component composition needs a transformation engine that transforms the representation of components (Aßmann 2003 [5]). For such transformation engines, COMPOST reuses external tools, such as the Java refactoring engine RECODER (Ludwig [88]). This transformation engine layer contains adapters between COMPOST and the external tools.
2. **Concrete Composition Layer.** On top of the pure fragment layer, this layer adds information for a concrete component model, e.g., Java fragment components, or ModelicaXML fragment components. Concrete composition constraints are incorporated that describe valid compositions, which can refer to the contents of the components. For instance, a constraint could be defined

¹ COMPOST and its interface layer UNICOMP can also model runtime and other types of component models.

- that disallows to encapsulating a Java method component into another Java method component.
3. **Time Specific Composition Layer.** On this layer the time of the composition is taken into account: static or runtime composition.
 4. **Abstract Composition Layer.** In this layer, knowledge is modeled that does not depend on the concrete component language, or on the concrete component model. General constraints are modeled, for instance, that each component has a list of subcomponents, the component hierarchy is a tree, or composition expressions employ the same type of component, independently of the concrete type.
 5. **UNICOMP Interface Layer.** The interfaces of the abstract composition layer have been collected into a separate interface layer, UNICOMP. This set of interfaces provides a generic fragment component model, from which different concrete component models can be instantiated.

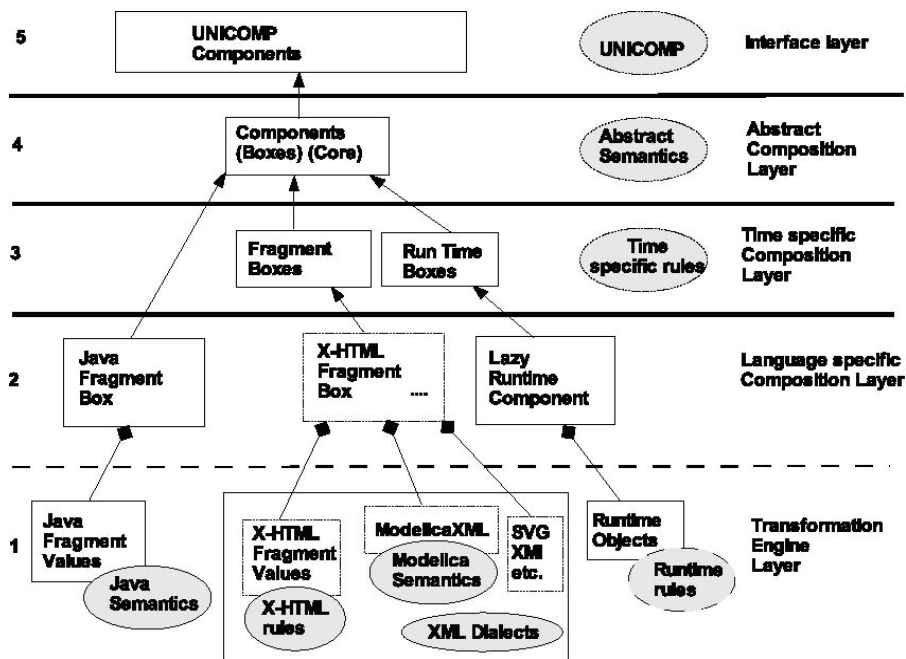


Figure 13-1. The layers of COMPOST.

For COMPOST applications, UNICOMP hides underlying concrete information about the component model to a large extent. An application uses COMPOST in a similar way as a component framework with an Abstract Factory (Gamma et al. 1994 [54]). When a component is created, its concrete type is given to the COMPOST factory. However, after creation, the application only uses the UNICOMP generic interfaces. Hence, generic applications can be developed that work for different component models, but use generic composition operations.

Already on the Abstract Composition Level, the following uniform operations for fragment components are available:

- *Other uniform basic operations.* COMPOST composition operators can address hooks and adapt them during composition for a context. As a basic set of abstract composition operators, *copy*, *extend*, and *rename* are available.
- *Uniform parameterizations.* Template processing works for completely different types of component models. After a semantics for composition points and bind operations has been defined, generic parameterization programs can be executed for template processing.
- *Uniform extensions.* The extension operator works on all types of components.
- *Uniform inheritance.* On the abstract composition layer COMPOST defined several inheritance operators that can be employed to share components, be it Java, or XML-based components. Inheritance is explained as a copy-and-extend operation, and both copy and extend operations are available in the most abstract layer.
- *Uniform connection.* COMPOST allows for uniform connection operations, as well for topologic as well as concrete connections (Aßmann 2003 [5]).
- *Uniform aspect weaving.* Based on these basic uniform operations, uniform aspect weaving operations (Karlsson 2003 [76]), can be defined.

The great advantage of the layer structure is that new component models, e.g., for XML languages, can be added easily as we show in this chapter. In fact, COMPOST is built for extension: adding a new component model is easy, it consists of adding appropriate classes in the concrete composition levels, subclassing from the abstract composition level as we show in Section 13.3.

13.2.2.2 Composition Constraints

Each COMPOST layer contains constraints for composition. These constraints consist of code that validates components and compositions.

- *Composite component constraints.* A component must be composite, i.e., the composed system is a hierarchy of subsystems. A component is the result of a composite composition expression or a composition program.
- *Composition typing constraints.* Composition operations must fit to components and their composition points. For instance, a composer may only bind appropriate values to composition points (fragments to fragments, runtime values to runtime values), or use a specific extension semantics.
- *Constraints on the content of components.* For instance, for a Java composition system, this requires that the static semantics of Java is modeled, and that this semantics controls the composition. For an XML

dialect, semantic constraints can be modeled, for instance, that all links in a document must be valid, i.e., point to a reasonable target. Our extended framework presented in this chapter provides parts of the Modelica semantics in top of the ModelicaXML format.

With these constraints, it should be possible to type-check composition expressions and programs in the UNICOMP framework. Many of these constraints can be specified in a logic language, such as first order logic (Datalog) or OWL (W3C [165]), and can be generated to check objects on every layer.

13.2.2.3 Support for staged composition

COMPOST supports staged composition as follows. Firstly, the UNICOMP layer has been connected to the Component Workbench, the visual component editor of the VCF (Oberleitner and Gschwind 2002 [108]). Composition programs for fragment component models can be edited from the Component Workbench, and executed via COMPOST.

So far, a case study has been build for a web-based conference reviewing system that requires Java and XHTML composition. This chapter shows how to *compose Modelica components* by using its alternative XML representation: ModelicaXML.

Secondly, COMPOST can be used to prepare components such that they fit into component models of stage 2 and 3. For instance, COMPOST connectors can prepare a Java class for use in CORBA context (Aßmann et al. 2000 [6]). They can also be used to insert event-emitting code, to prepare a class for Aspect-Oriented Programming.

13.3 COMPOST extension for Modelica

This section describes the Modelica component model. The architecture of our system is presented. Modelica Box and Hook hierarchies are explained. Finally, various composition programs are given as examples.

13.3.1 Overview

The architecture of the composition system is given in Figure 13-2. A Modelica parser is employed to generate the ModelicaXML representation. ModelicaXML is fed into the COMPOST framework where it can be composed and transformed. The result is transformed back into Modelica code by the use of a ModelicaXML unparser.

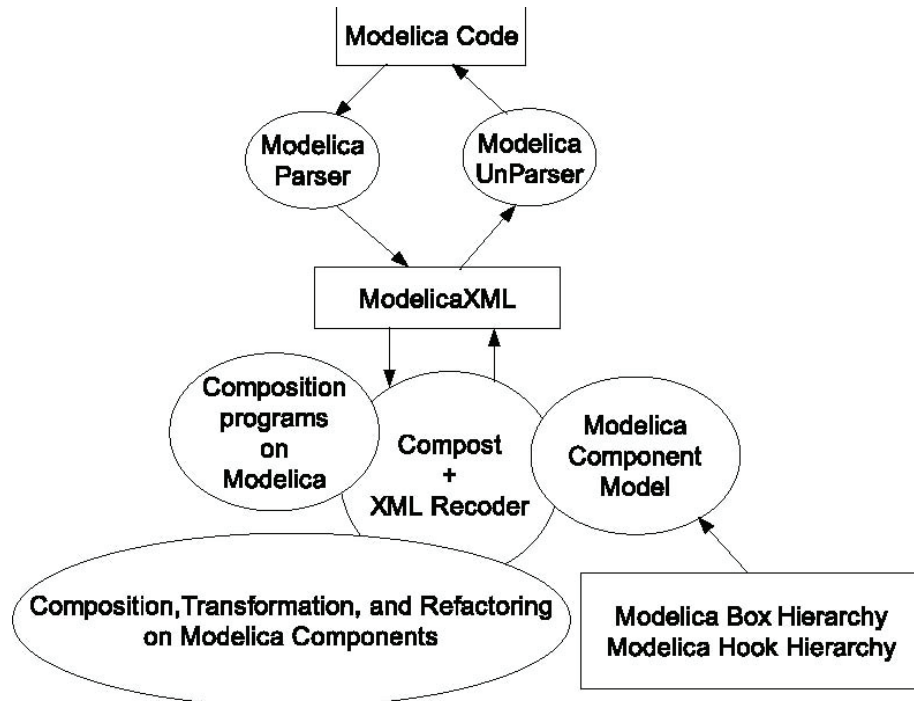


Figure 13-2. The XML composition. System Architecture Overview.

13.3.2 Modelica Box Hierarchy

Besides general classes, Modelica uses so called restricted class constructs to structure information and behavior: models, packages, records, types, functions, connectors and blocks. Restricted classes have most properties in common with general classes, but have some restrictions, e.g. there are no equations in records.

Modelica classes are composed of elements of different kinds, e.g.:

- Import or extends declarations.
- Public or protected variable declarations.
- Equation and algorithm sections.

Each of the Modelica restricted classes and each of the element types have their corresponding box class in the Modelica Box hierarchy (Figure 13-3).

In our case, the boxes (templates) are mapped to their specific element types in the ModelicaXML representation. For example, the `ModelicaClass` box is mapped to a `<define id="ClassName">..</define>` element. The `ModelicaClass` box can contain several `ModelicaElement` boxes and can contain itself in the case that one Modelica class is declared inside another class.

The boxes that inherit from `ModelicaContainer` represent the usual constructs of the Modelica language. The boxes that inherit from `ModelicaElement` are defining the contents of the boxes that inherit from `ModelicaContainer`.

The boxes incorporate constraints derived from Modelica static semantics. For example, constraints specify that inside a `ModelicaRecord` is not allowed to have `ModelicaEquationSections`.

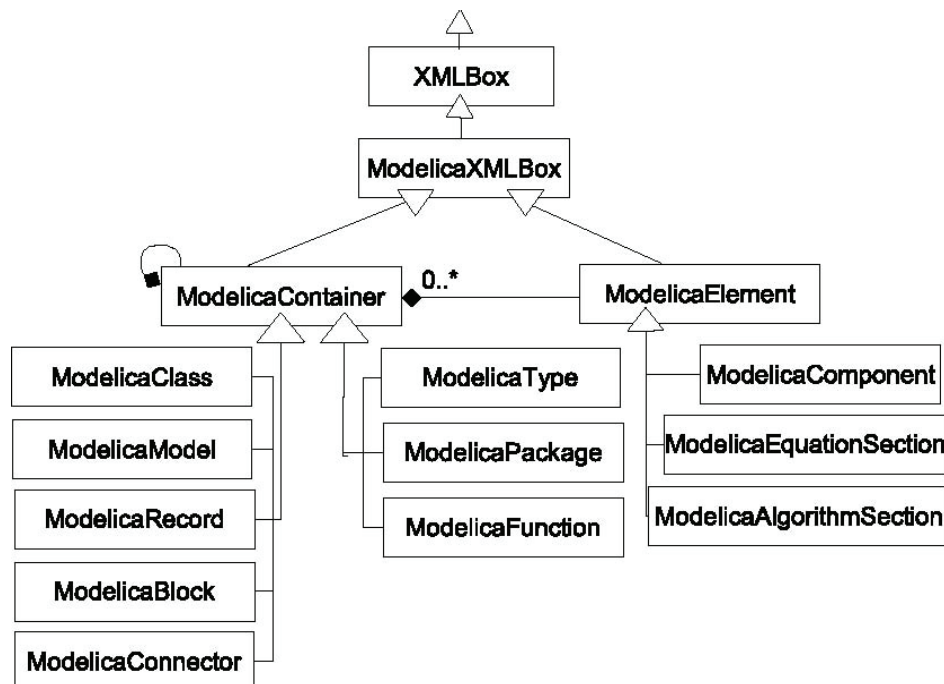


Figure 13-3. The Modelica Box Hierarchy defines a set of templates for each language structure.

While these constraints in our case were specified in the Java code, a future extension will automatically generate these constraints from external specifications expressed in formalisms such as Document Type Definition (DTD) (W3C [158]), Web Ontology Language (OWL) (W3C [164], W3C [165]) or Relational Meta-Language (RML) (PELAB 1994-2008 [117], Pettersson 1995 [120], Pettersson 1999 [122]).

13.3.3 Modelica Hook Hierarchy

Implicit Hooks are fragments of Modelica classes that have specific meaning according to Modelica code structure and semantics. By using Hooks one can easily change/extract parts of the code. In the Modelica Hook Hierarchy presented in (Figure 13-4) only Implicit Hooks are defined for the Modelica code.

There is no need to define Declared Hooks especially for Modelica, because the `XMLDeclaredHook` already performs this operation. One can have an XML declared hook that extracts from the XML document the contents of an element with a specified tag, i.e., `<extract ...>`.

Hooks are used to configure parts of boxes. The `XMLImplicitHook` is specialized as `ModelicaParameterHook` or `ModelicaModificationHook`.

`ModelicaParameterHook` binds variable components in ModelicaXML that have variability attribute set to "parameter". To provide typing constraints, specific hooks for `real_literal`, `integer_literal`, `string_literal` types have been declared. These constraints the binding of the parameters to values of proper type.

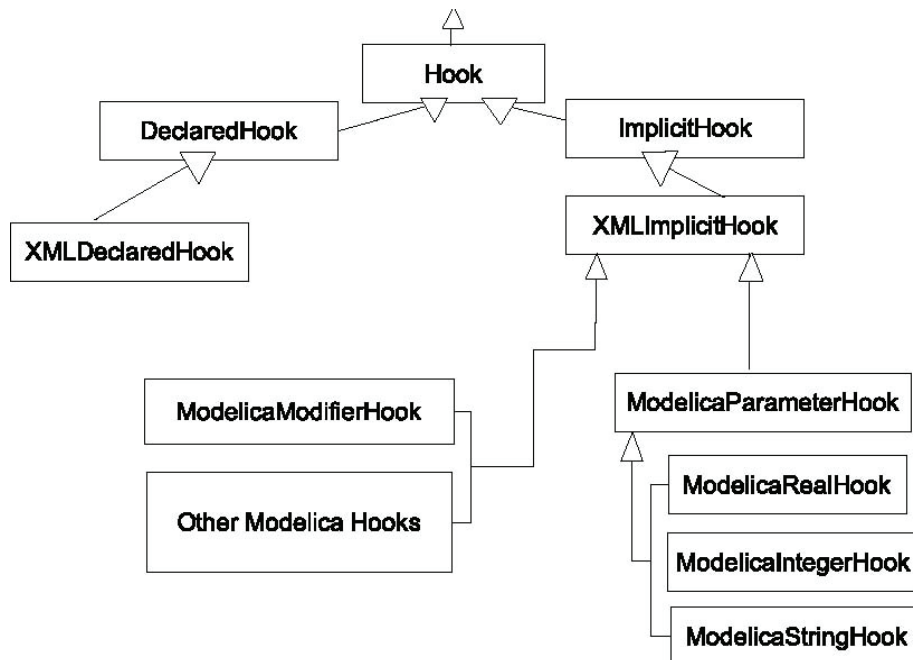


Figure 13-4. The Modelica Hook Hierarchy.

`ModelicaModificationHook` targets component declarations that have their elements changed by modifiers. In the `HelloWorld` example in Section 13.2.1, the modifier is imposing on component `x` to change its start value. At the ModelicaXML level the `ModelicaModificationHook` is searching for XML elements of the form:

```

<component ident="ComponentName">
  <modification_arguments>
    <element_modification>
      <component_reference ident="element"/>
      <modification_equals>value initialization e.g.

```

```

    <integer_literal>1</integer_literal>
  </modification_equals>
</element_modification>
</modification_arguments>
</component>

```

This hook will bind proper values to the modified elements.

Also, other types of implicit hooks can be specified like hooks for the left hand side or the right hand side of an equation hooks that change types of components, hooks that change the documentation part of a class declaration, etc.

13.3.4 Examples of Composition and Transformation Programs

This subsection gives concrete examples on the usages of our framework. The examples are written in Java, but they could easily be performed using a tool that has visual abstractions for the composition operators. For presentation issues only the Modelica code is given in the examples below and their corresponding ModelicaXML representation is presented in Section 13.5.

13.3.4.1 Generic Parameterization with Type Checking

To be able to reuse components into different contexts they should be highly configurable. Configuration of parameters in Modelica is specified in class definitions and can be modified in parameter declaration. The values can be read from external sources using external functions implemented in C or Fortran. In the example below we show how the parameters of a Modelica component can be configured using implicit hooks. Because we use Java, the parameter/value list can be read from any data source (XML, SQL, files, etc). The example is based on the following Modelica class:

```

class Engine
  parameter Integer cylinders = 4;
  Cylinder c[cylinders];
  /* additional parameters, variables and equations */
end Engine;

```

Different versions of the `Engine` class can be automatically generated using a composition script. Also, the parameter values are type checked before they are bound to ensure their compatibility. The composition script is given below partially in Java, partially in pseudo-code:

```

ModelicaCompositionSystem cs = new
    ModelicaCompositionSystem();

ModelicaClass templateBox =
    cs.createModelicaClass("Engine.mo.xml");

/* read parameters from configuration file, XML or SQL */

```

```
foreach engine entry X
{
  ModelicaClass engineX =
    templateBox.cloneBox().rename("Engine_"+X);

  foreach engine parameter
  {
    engineX.findHook("parameterName").bind(parameterValue);
    /* typed parameterization */
  }
  engineX.print();
}
```

Using a similar program, the modification of parameters can be performed in parameter declarations.

13.3.4.2 Class Hierarchy Refinement using Declared Hooks

When designing libraries one would like to split specific classes into a more general part and a more specific part. As an example, one could split the class defined below into two classes that inherit from each other, one more generic and one more specific, in order to exploit reuse. Also if one wants to add a third class, e.g. `RectangularBody`, to the created hierarchy the transformation above would be beneficial. The specific class that should be modified is given below:

```
class CelestialBody "Celestial Body"
  Real mass;
  String name;
  constant Real g = 6.672e-11;
  parameter Real radius;
end CelestialBody;
```

The desired result, the two split classes where one inherits from the other, is shown below:

```
class Body "Generic Body"
  Real mass;
  String name;
end Body;

class CelestialBody "Celestial Body"
  extends Body;
  constant Real g = 6.672e-11;
  parameter Real radius;
end CelestialBody;
```

One can see that this transformation extracts parts of classes and inserts them into a new created class. Also, the old class is modified to inherit from the newly created class.

This transformation is performed with the help of one declared hook (for the extraction part) and an implicit hook for the superclass, with its value bound to the

newly created class. The user will guide this operation by specifying, with a declared hook or visually, which parts should be moved in the new class. The composition program that performs these transformations is as follows:

```
ModelicaCompositionSystem cs = new
    ModelicaCompositionSystem();
ModelicaClass bodyBox = cs.createClass("Body.mo.xml");
ModelicaClass celestialBodyBox =
    cs.createModelicaClass("Celestial.mo.xml");
ModelicaElement extractedPart =
    celestialBody.findHook("extract").getValue();

/* empty the hook contents */
celestialBody.findHook("extract").bind(null);

bodyBox.append(extractedPart)
bodyBox.print();
celestialBody.findHook("superclass").bind("Body");
/* or findSuperclass().bind("Body"); */

celestialBody.print();
```

Similar transformations can be used to compose Modelica models based on the interpretation of other modeling languages. During such composition some classes need to be wrapped to provide a different interface. For example, when there is only a force specified for moving a robot arm, but the available library of components only provides electrical motors that generate a force proportional to a voltage input.

13.3.4.3 Composition of classes or model flattening

Mixin composition of the entire contents of two or more classes into one another is performed when the models are flattened i.e. as the first operation in model obfuscation or at compilation time. The content of the classes composed below is not relevant for this particular operation. The composition program that encapsulates this behavior is as follows:

```
ModelicaCompositionSystem cs = new
    ModelicaCompositionSystem();
ModelicaClass resultBox =
    cs.createModelicaClass("Class1.mo.xml");
ModelicaClass firstMixin =
    cs.createModelicaClass("Class2.mo.xml");
ModelicaClass secondBox =
    cs.createModelicaClass("Result.mo.xml");

resultBox.mixin(firstMixin);
resultBox.mixin(secondMixin);
resultBox.print();
```

It first reads the two classes from files, creates a new result class and pastes the contents of the first classes inside the new class.

13.4 Conclusions and Future work

We have shown how composition on Modelica, using its alternative the ModelicaXML representation, can be achieved with a small extension of the COMPOST framework. While this is a good start, we would like to extend our work in the future with some additional features like:

- More composition operators and more transformations, i.e., obfuscation, symbolic transformation of equations, aspect oriented debugging of component behavior by weaving assert statements in equations, etc.
- Implementation of full Modelica semantics to guide the composition, based on the already existing Modelica compiler implemented in the OpenModelica system.
- Validation of the composed or transformed components with the OpenModelica compiler.
- Automatic composition of Modelica models based on interpretation of other modeling languages.

Modelica should provide additional constraints on composition, based on the domain knowledge. These constraints are specifying, for example, that specific components should not be connected even if their connectors allow it. We would like to further investigate how these constraints could be specified by library developers.

13.5 Appendix

CelestialBody in ModelicaXML format before transformation:

```
<definition ident="CelestialBody" restriction="class"
  string_comment="Celestial Body"/>
  <component visibility="public"
    ident="mass" type="Real"/>
  <component visibility="public"
    ident="name" type="String"/>
  <component visibility="public"
    variability="constant" ident="g"
    type="Real">
    <modification_equals>
      <real_literal value="6.672e-11"/>
    </modification_equals>
  </component>
```

```

    <component visibility="public"
               variability="parameter" ident="radius"
               type="Real"/>
  </definition>

```

CelestialBody and Body in ModelicaXML format after transformation:

```

<definition ident="Body" restriction="class"
            string_comment="Generic Body"/>
  <component visibility="public" ident="mass" type="Real"/>
  <component visibility="public"
            ident="name" type="String"/>
</definition>

<definition ident="CelestialBody" restriction="class"
            string_comment="Celestial Body"/>
  <extends type="Body"/>
  <component visibility="public"
            variability="constant" ident="g"
            type="Real">
    <modification_equals>
      <real_literal value="6.672e-11"/>
    </modification_equals>
  </component>
  <component visibility="public" variability="parameter"
            ident="radius" type="Real"/>
</definition>

```

The Engine class representation in ModelicaXML:

```

<definition ident="Engine" restriction="class">
  <component visibility="public" variability="parameter"
            type="Integer" ident="cylinders">
    <modification_equals>
      <integer_literal value="4"/>
    </modification_equals>
  </component>
  <component visibility="public" type="Cylinder" ident="c">
    <array_subscripts>
      <component_reference ident="cylinders"/>
    </array_subscripts>
  </component>
</definition>

```


Part VI

Conclusions and Future Work

Chapter 14

Conclusions and Future Work

As most of the chapters in this thesis have their own specific conclusions and future work, this final chapter presents our general conclusions to the work presented. A summary of the main results and the main contributions of the thesis are reiterated here. We also provide directions for future research.

14.1 Conclusions

The thesis presents the new MetaModelica language that successfully employs meta-modeling and meta-programming features to address the entire product modeling process. Portable debugging methods and tools that support the new language were also designed, implemented, and analyzed in the thesis.

The design, implementation and evaluation of efficient compilers targeting the MetaModelica language are presented in the thesis. The implemented compilers are publicly available and extensively used in industry and academia for large applications.

Moreover, the tools (compilers, debuggers, model editors, and additional tools) supporting the MetaModelica language were integrated into an advanced development environment based on the Eclipse platform. The integrated development environment was evaluated on non-trivial industrial applications.

The integration of Modelica-based modeling and simulation tools with model-driven product design tools within a flexible framework that supports scalable model selection and configuration is also proposed.

Most of our thesis contributions have been implemented and integrated into open-source development environments for EOO languages. The evaluations performed using several case studies show the efficiency of our meta-modeling and meta-programming methods and tools.

We conclude that the work presented in this thesis supports our research hypothesis:

- EOO languages can be successfully generalized to support software modeling, thus addressing the whole product modeling process.

- Integrated environments that support such a generalized EOO language can be created and effectively used on real-sized applications.

The integrated model-driven environments and the new MetaModelica language presented in the thesis provide efficient and effective methods for designing and developing complex product models. Methods and tools for debugging, management, serialization, and composition of models are also contributed.

To reiterate, the main research contributions of the thesis are:

- The design, implementation and evaluation of a new general executable mathematical modeling and semantics meta-modeling language called MetaModelica. The MetaModelica language extends the existing Modelica language with support for meta-modeling, meta-programming and exception handling facilities.
- The design, implementation and evaluation of advanced portable debugging methods and frameworks for runtime debugging of MetaModelica and semantic specifications.
- The design, implementation, and evaluation of several integrated model-driven environments supporting creation, development, refactoring, debugging, management, composition, serialization and graphical representation of models in EOO languages. Additionally, an integrated model-driven product design and development environment based on EOO languages is also contributed.
- Alternative representation of EOO models based on XML and UML/SysML are investigated and evaluated. Transformation and invasive composition of EOO models has also been investigated.

The thesis also discusses our work in comparison to related work and outlines the differences, the advantages and the weaknesses of our contributions.

14.2 Future Work Directions

While most of the research goals of the thesis have been achieved the presented work can be further improved and extended. In this section we present possible future work directions:

- Most of the language support (pattern matching, exception handling, the high-level data structure extensions, etc) needed for the OpenModelica compiler bootstrapping has been implemented. Our current work targets the integration of the MetaModelica compiler prototype runtime with the OpenModelica compiler runtime to finalize the compiler bootstrapping procedure. The OpenModelica compiler bootstrapping will provide further optimization, simplification, and modularization of the current compiler

specification due to providing full MetaModelica language support, compared to the subset supported by the prototype. When the bootstrapping procedure has been completed, the current MetaModelica compiler prototype will retire and the compilation chain of OpenModelica will be highly simplified. Due to easier programming based on the full MetaModelica language, a simplified compilation procedure, and a simplified compiler specification we expect more contributions from the OpenModelica community developers.

- Further work on the MetaModelica unified language design targeting the equation evaluation strategies is needed. In the current design and implementation the order of equations in the meta-programming functions is important. We intend to remove this restriction in the future.
- The modularity and scalability of the MetaModelica language should be further researched. Investigation of the suitability and possible adaptation of the current Modelica component model with regards to software modeling should be carried out. Alternative formalisms such as attribute grammars (Ekman and Hedin 2007 [33]) can provide ideas for improvements in the language design, modularity, and equation evaluation strategies used in the MetaModelica language and its supporting environments to further extend the expressivity and usefulness of the language.
- Model-driven design and development of whole products is briefly investigated in the thesis. However we consider that more research is needed in this area, especially on the integration of all our existing tools in the product design and development process. The Modelica-UML-SysML (ModelicaML) and the FMDesign environments could be integrated to support several views of the same product model. Another research direction worth investigating is the integration of our Modelica tools with existing SysML tools via the ModelicaML profile. Such integration will provide full system simulation capabilities to existing SysML tools.
- Our general run-time debugging framework for EOO languages should be fully implemented, evaluated and integrated with existing static equation-based debugging frameworks.
- The tools for generation of alternative EOO model representations (XML, ModelicaML) and invasive composition engine should be integrated into our MDT environment.

Bibliography

- [1] David Akhvlediani. *Design and Implementation of a UML profile for Modelica/SysML*, Department of Computer and Information Science, Linköping University, 2007, Master Thesis No: LITH-IDA-EX--06/061—SE
- [2] Altova, *XmlSpy System*. 2008, Altova. www: <http://www.xmlspy.com/>. Last Accessed: 2008.
- [3] Mogens Myrup Andreassen. *Machine Design Methods Based on a Systematic Approach (Syntesemetoder på systemgrundlag)*, Lund Technical University, 1980
- [4] Peter Aronsson, Peter Fritzson, Levon Saldamli, Peter Bunus, and Kaj Nyström. *Meta Programming and Function Overloading in OpenModelica*. in *3rd International Modelica Conference*. 2003. Linköping. (<http://www.modelica.org>). p. 431-440
- [5] Uwe Aßmann, *Invasive Software Composition*. 2003: Springer-Verlag Heidelberg. ISBN: 3540443851.
- [6] Uwe Aßmann, Thomas Genßler, and Holger Bär. *Meta-programming Grey-box Connectors*. in *International Conference on Object-Oriented Languages and Systems (TOOLS Europe)*. 2000. Piscataway, NJ. IEEE Press
- [7] Uwe Aßmann and Andreas Ludwig, *COMPOST (The Software COMPOsition SysTem)*. 2005, 1998-2003 Karlsruhe University, IPD Prof. Goos, 1998-2003 Andreas Ludwig, 2001-2003 Uwe Aßmann, 2001-2003 Linköpings Universitet, IDA, PELAB, RISE. www: <http://www.the-compost-system.org/>. Last Accessed: 2005.
- [8] Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, Didier Parigot, and Claude Pasquier. *SmartTools: A Generator of Interactive Environments Tools*. in *International Conference on Compiler Construction (CC2001)*. 2001. Genova, Italy. www: <http://www-sop.inria.fr/smartool/>.
- [9] Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, Joël Fillon, Didier Parigot, Claude Pasquier, and Claudio Sacerdoti Coen. *SmartTools: a*

- development environment generator based on XML technologies.* in *The XML Technologies and Software Engineering (ICSE'2001)*. 2001. Toronto, Canada. ICSE workshop proceedings. www: <http://www-sop.inria.fr/smartool/>.
- [10] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, *Description Logics Handbook*. 2003, New York, NY: Cambridge University Press.
- [11] Greg Badros. *JavaML: A Markup Language for Java Source Code*. in *Proceedings of The 9th International World Wide Web Conference*. 2000. Amsterdam, Netherlands
- [12] Tim Berners-Lee, James Hendler, and Ora Lassila, *The Semantic Web*, in *Scientific American*. 2001.
- [13] Simon Björklén. *Extending Modelica with High-Level Data Structures*, Department of Computer and Information Science, Linköping University, 2008, Master Thesis
- [14] Johansson Björn, Jonas Larsson, Magnus Sethson, and Petter Krus. *An XML-Based Model Representation for model management, transformation and exchange*. in *ASME International Mechanical Engineering Congress*. 2002. New Orleans, USA
- [15] Patrik Borras, Dominique Clement, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. *CENTAUR: The System*. in *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. 1988. 24 of SIGPLAN. p. 14-24
- [16] R.H. Bracewell and D.A.Bradley. *Schemebuilder, A Design Aid for Conceptual Stages of Product Design*. in *International Conference on Engineering Design, IECD'93*. 1993. The Hague
- [17] Gilad Bracha and W. Cook. *Mixin-based inheritance*. in *OOPSLA/ECOOP'90*. 1990. ACM SIGPLAN Notices. p. 303-311
- [18] David Broman. *Safety, Security, and Semantic Aspects of Equation-Based Object-Oriented Languages and Environments*, Department of Computer and Information Science, Linköping University, 2007, Licentiate Thesis No: 1337, <http://www.ep.liu.se/theses/abstract.xsql?dbid=10134>
- [19] Peter Bunus. *Debugging Techniques for Equation-Based Languages*, Department of Computer and Information Science, Linköping University, 2004, PhD Thesis No: 873
- [20] Peter Bunus and Peter Fritzson. *Semi-Automatic Fault Localization and Behavior Verification for Physical System Simulation Models*. in *18th IEEE*

International Conference on Automated Software Engineering. 2003.
Montreal, Canada

- [21] Emil Carlsson. *Translating Natural Semantics to Meta-Modelica*, Department of Computer and Information Science, Linköping University, 2005, Master's Thesis No: LITH-IDA-EX--05/073--SE
- [22] Ernst Christen and Kenneth Bakalar, *VHDL-AMS - A Hardware Description Language for Analog and Mixed-signal Applications*. IEEE Transactions on Circuits and Systems, Part II: Express Briefs, 1999. **46**(10): p. 1263-1272.
- [23] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. *A Simple Applicative Language: Mini-ML*. in *the ACM Conference on Lisp and Functional Programming*. 1986. also available as research report RR-529, INRIA, Sophia-Antipolis, May 1986.
- [24] DescriptionLogicsWebsite, *Description Logics*, maintained by Carsten Lutz. www: <http://dl.kr.org/>. Last Accessed: 2005.
- [25] Thierry Despeyroux. *Executable Specification of Static Semantics*. in *Semantics of Data Types*. 1984. Berlin, Germany. Springer-Verlag. Lecture Notes in Computer Science (LNCS) No:173. p. 215-233
- [26] Thierry Despeyroux, *TYPOL: A Formalism to Implement Natural Semantics*. 1988, INRIA, Sofia-Antipolis. www: <http://www.inria.fr/rrrt/rt-0094.html>.
- [27] Dynasim, *Dymola*. 2005. www: <http://www.dynasim.se/>. Last Accessed: 2005.
- [28] EasyComp, *The EasyComp EU project website*. 2004. www: <http://www.easycomp.org/>. Last Accessed: 2004.
- [29] Eclipse.Foundation, *Eclipse Development Platform*. 2001-2008. www: <http://www.eclipse.org/>. Last Accessed: 2008.
- [30] Eclipse.Foundation, *Eclipse Modeling Framework (EMF)*. 2008. www: <http://www.eclipse.org/emf>. Last Accessed: 2008.
- [31] Eclipse.Foundation, *Graphical Editing Framework (GEF)*. 2008. www: <http://www.eclipse.org/gef>. Last Accessed: 2008.
- [32] Eclipse.Foundation, *Graphical Modeling Framework (GMF)*. 2008. www: <http://www.eclipse.org/gmf>. Last Accessed: 2008.
- [33] Torbjörn Ekman and Görel Hedin. *The JastAdd Extensible Java Compiler*. in *The 22nd Annual ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 2007. Montreal, Canada

- [34] Hilding Elmqvist, Dag Brück, Sven Erik Mattsson, Hans Olsson, and Martin Otter, *Dymola, Dynamic Modeling Laboratory, User's Manual*. 2003.
- [35] Hilding Elmqvist, Sven Erik Mattsson, and Martin Otter. *Modelica - A Language for Physical System Modeling, Visualization and Interaction*. in *IEEE Symposium on Computer-Aided Control System Design*. 1999. Hawaii, USA
- [36] Burak Emir, Martin Odersky, and John Williams. *Matching Objects With Patterns*. in *Proceeding of 21st European Conference on Object-Oriented Programming (ECOOP)*. 2007. Berlin, Germany. Springer. LNCS 4609
- [37] Georgina Fábíán. *A Language and Simulator for Hybrid Systems*, Technische Universiteit Eindhoven, 1999
- [38] Jorge A. Ferreira and João P. Estima de Oliveira. *Modelling hybrid systems using statecharts and Modelica*. in *7th IEEE International Conference on Emerging Technologies and Factory Automation*. 1999. Barcelona, Spain. 2. p. 1063-1069
- [39] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts, *Refactoring: Improving the Design of Existing Code*. 1999: Addison Wesley. ISBN: 0201485672.
- [40] Wolfgang Freiseisen, Robert Keber, Wihelm Medetz, Petru Pau, and Dietmar Stelzmueller. *Using Modelica for testing embedded systems*. in *the 2nd International Modelica Conference*. 2002. Munich, Germany. (<http://www.modelica.org>)
- [41] Peter Fritzson, *Symbolic Debugging through Incremental Compilation in an Integrated Environment*. Journal of Systems and Software, 1983. **3**: p. 285-294.
- [42] Peter Fritzson. *Towards a Distributed Programming Environment based on Incremental Compilation*, Department of Computer and Information Science, Linköping University, 1984, PhD Thesis No: 109
- [43] Peter Fritzson, *Efficient Language Implementation by Natural Semantics*. 1998.
- [44] Peter Fritzson, *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. 2004: Wiley-IEEE Press. 940. ISBN: 0-471-471631.
- [45] Peter Fritzson, *MathModelica - An Object Oriented Mathematical Modeling and Simulation Environment*. Mathematica Journal, 2006. **10**(1).

-
- [46] Peter Fritzson, Peter Aronsson, Peter Bunus, Vadim Engelson, Levon Saldamli, Henrik Johansson, and Andreas Karstöm. *The Open Source Modelica Project*. in *Proceedings of The 2th International Modelica Conference*. 2002. Munich, Germany
 - [47] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, David Broman, Peter Bunus, Vadim Engelson, Henrik Johansson, and Andreas Karstöm, *The OpenModelica Modeling, Simulation and Software Development Environment*. Simulation News Europe, 2005. **44/45**.
 - [48] Peter Fritzson, Mikhail Auguston, and Nahid Shahmehri, *Using Assertions in Declarative and Operational Models for Automated Debugging*. Journal of Systems and Software, 1994. **25**(3): p. 223-232.
 - [49] Peter Fritzson and Peter Bunus. *Modelica, a General Object-Oriented Language for Continuous and Discrete-Event System Modeling and Simulation*. in *35th Annual Simulation Symposium*. 2002. San Diego, California
 - [50] Peter Fritzson and Vadim Engelson. *Modelica, a general Object-Oriented Language for Continuous and Discrete-Event System Modeling and Simulation*. in *12th European Conference on Object-Oriented Programming (ECOOP'98)*. 1998. Brussels, Belgium
 - [51] Peter Fritzson, Adrian Pop, and Peter Aronsson. *Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica*. in *4th International Modelica Conference*. 2005. Hamburg, Germany. (<http://www.modelica.org>)
 - [52] Peter Fritzson, Adrian Pop, Kristoffer Norling, and Mikael Blom. *Comment- and Indentation Preserving Refactoring and Unparsing for Modelica*. in *6th International Modelica Conference*. 2008. Bielefeld, Germany. (<http://www.modelica.org>)
 - [53] Peter Fritzson, Lars Viklund, Dag Fritzson, and Johan Herber, *High Level Mathematical Modeling and Programming*. Scientific Computing, IEEE Software, 1995: p. 77-87.
 - [54] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994, Reading, MA: Addison Wesley.
 - [55] Sabine Glesner and Wolf Zimmermann, *Natural semantics as a static program analysis framework*. ACM Transactions on Programming Languages and Systems (TOPLAS), 2004. **26**(3): p. 510-577.

- [56] GNU, *Bison (a general-purpose parser generator)*. 2005, The Free Software Foundation. www: <http://www.gnu.org/software/bison>. Last Accessed: 2005.
- [57] GNU, *Emacs, The Grand Unified Debugger (GUD)*. 2005, The Free Software Foundation. www: http://www.gnu.org/software/emacs/manual/html_node/Debuggers.html#Debuggers. Last Accessed: 2005.
- [58] GNU, *Flex (a fast lexical analyser generator)*. 2005, The Free Software Foundation. www: <http://www.gnu.org/software/flex/>. Last Accessed: 2005.
- [59] GNU, *The GNU Project debugger*. 2005, The Free Software Foundation. www: <http://www.gnu.org/software/gdb/gdb.html>. Last Accessed: 2005.
- [60] GNU, *The GNU Readline Library*. 2005, The Free Software Foundation. www: <http://cnswww.cns.cwru.edu/php/chet/readline/rltop.html>. Last Accessed: 2005.
- [61] GNU, *Qexo - The GNU Kawa implementation of XQuery*. 2005, The Free Software Foundation. www: <http://www.gnu.org/software/qexo>. Last Accessed: 2005.
- [62] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *The Java Language Specification*. 3rd edition ed. 2005: Prentice Hall. ISBN: 978-0321246783.
- [63] Volker Haarslev, Ralf Möller, and Michael Wessel, *RACER User's Guide and Reference Manual*. 2004. www: <http://www.sts.tu-harburg.de/~r.f.moeller/racer/>. Last Accessed: 2005.
- [64] George T. Heineman and William T. Councill, *Component-Based Software Engineering*, ed. George T. Heineman and William T. Councill. 2001: Addison Wesley.
- [65] Sir Charles Antony Richard Hoare, *Communicating Sequential Processes*. 1985: Prentice-Hall.
- [66] Mikael Holmén. *Natural Semantics Specification and Frontend Generation for Java 1.2*, Department of Computer and Information Science, Linköping University, 2000, Master Thesis No: LiTH-IDA-Ex-00/60
- [67] Ian Horrocks, *The FaCT System*. www: <http://www.cs.man.ac.uk/~horrocks/FaCT/>. Last Accessed: 2005.
- [68] Paul Hudak, *The Haskell School of Expression*. 2000, New York: Cambridge University Press. ISBN: 0521644089.

-
- [69] IEEE, *IEEE 1000 The Authoritative Dictionary of IEEE Standard Terms*. 2000, IEEE Press: New York, USA.
 - [70] INCOSE, *International Council on System Engineering*. 1990-2008. www: <http://www.incose.org>. Last Accessed: 2008.
 - [71] ITI.GmbH, *SimulationX*. 2008. www: <http://www.iti.de/>. Last Accessed: 2008.
 - [72] Björn Johansson, Jonas Larsson, Magnus Sethson, and Petter Krus. *An XML-Based Model Representation for Model Management Transformation and Exchange*. in *ASME International Mechanical Engineering Congress*. 2002. New Orleans, USA. LiTH-IKP-CR-562
 - [73] Olof Johansson and Petter Krus. *FMDesign - A Tool and Interchange Format for Product Concept Designs*. in *ASME International Design Engineering Technical Conferences & Computers and Information In Engineering Conference*. 2005. Long Beach, California, USA
 - [74] Olof Johansson, Adrian Pop, and Peter Fritzson. *ModelicaDB - A Tool for Searching, Analysing, Crossreferencing and Checking of Modelica Libraries*. in *4th International Modelica Conference*. 2005. Hamburg-Harburg, Germany. (<http://www.modelica.org>)
 - [75] Gilles Kahn, *Natural Semantics*, in *Programming of Future Generation Computers*, Niva M., Editor. 1988, Elsevier Science Publishers: North Holland. p. 237-258.
 - [76] Mattias Karlsson. *Component-Based Aspect Weaving Through Invasive Software Composition*, Department of Computer and Information Science, Linköping University, 2003, Master's thesis
 - [77] Uwe Kastens, William McCastline Waite, and Anthony M. Sloane, *Generating Software from Specifications*. 2007: Jones and Bartlett Publishers. ISBN: 0763741248.
 - [78] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. *Aspect-oriented programming*. in *ECOOP'97*. 1997. Springer Verlag. Lecture Notes in Computer Science (LNCS) No:1241. p. 220-242
 - [79] Simon Lacoste-Julien, Hans Vangheluwe, Juan de Lara, and Pieter J. Mosterman. *Meta-modelling hybrid formalisms*. in *IEEE International Symposium on Computer-Aided Control System Design*. 2004. Taipei, Taiwan. IEEE Computer Society Press, Invited paper. p. 65-70
 - [80] Juan de Lara, Esther Guerra, and Hans Vangheluwe. *Meta-Modelling, Graph Transformation and Model Checking for the Analysis of Hybrid Systems*. in *Applications of Graph Transformations with Industrial Relevance (AGTIVE*

- 2003). 2003. Charlottesville, Virginia, USA. Springer-Verlag. Lecture Notes in Computer Science (LNCS) No:3062. p. 292 - 298
- [81] Jonas Larsson, Björn Johansson, Petter Krus, and Magnus Sethson. *Modelith: A Framework Enabling Tool-Independent Modeling and Simulation*. in *European Simulation Symposium*. 2002. Dresten, Germany
- [82] Akos Ledecz, Arpad Bakay, Miklos Maroti, Peter Volgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gabor Karsai, *Composing Domain-Specific Design Environments*. Computer, 2001. **November**: p. 44-51.
- [83] Akos Ledecz, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. *The Generic Modeling Environment*. in *Workshop on Intelligent Signal Processing*. 2001. Budapest, Hungary. www: <http://www.isis.vanderbilt.edu/Projects/gme/>.
- [84] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon, *The Objective Caml System. Documentation and User's Manual*. 2007. www: <http://caml.inria.fr/pub/docs/manual-ocaml>.
- [85] Henry Liebermann, *The Debugging Scandal and What To Do About It*. Communications of the ACM, 1997. **40**(4): p. 27-29.
- [86] J. Lindskov, Knudsen, M. Lofgren, O. Lehrmann Madsen, and B. Magnusson, *Object-Oriented Environments - The Mjølner Approach*. 1993: Prentice Hall. ISBN: 0-13-009291-6.
- [87] Jed Liu and Andrew C. Myers. *JMatch: Iterable Abstract Pattern Matching for Java*. in *Proceeding of the 5th International Symposium on Practical Aspects of Declarative Languages (PADL)*. 2003. New Orleans, LA, USA. Springer. Lecture Notes in Computer Science (LNCS) No:2562. p. 110-127
- [88] Andreas Ludwig, *The RECODER Refactoring Engine*. www: <http://recoder.sourceforge.net>. Last Accessed: 2008.
- [89] Sarah Mallet and Mireille Ducassé. *Generating deductive database explanations*. in *International Conference on Logic Programming*. 1999. Las Cruces, New Mexico. MIT Press
- [90] John J. Marciniak, *Encyclopedia of software engineering*. Vol. 1 A-N. 1994, New York, NY: Wiley-Interscience.
- [91] MathCore, *MathModelica*, MathCore. www: <http://www.mathcore.se/>. Last Accessed: 2008.

-
- [92] Maude.Team, *The Maude System Website*, University of Illinois. www: <http://maude.cs.uiuc.edu/>. Last Accessed: 2008.
 - [93] Deborah L. McGuinness. *Explaining Reasoning in Description Logics*, Rutgers University, 1996, PhD. Thesis
 - [94] Deborah L. McGuinness and Alex Borgida. *Explaining Subsumption in Description Logics*. in *Fourteenth International Joint Conference on Artificial Intelligence*. 1995
 - [95] Deborah L. McGuinness and Paulo Pinheiro da Silva. *Infrastructure for Web Explanations*. in *2nd International Semantic Web Conference (ISWC2003)*. 2003. USA. Springer-Verlag. Lecture Notes in Computer Science (LNCS) No:2870. p. 113-129
 - [96] M. Douglas (Malcolm) McIlroy. *Mass produced software components*. in *NATO Software Engineering Conference*. 1968. Garmisch, Germany. p. 138-155
 - [97] Robert Milner, Mads Tofte, Robert Harper, and David MacQueen, *The Definition of Standard ML - Revised*. 1997: MIT Press. ISBN: 0-262-63181-4.
 - [98] Oh Min and C. C. Pantelides, *A Modeling and Simulation Language for Combined Lumped and Distributed Parameter System*. Computers and Chemical Engineering, 1996. **20**(6-7): p. 611-633.
 - [99] Modelica.Association, *Modelica: A Unified Object-Oriented Language for Physical Systems Modeling, Language Specification*. 1996-2008. www: <http://www.modelica.org/>. Last Accessed: 2008.
 - [100] Modelica.Association, *The Modelica Language Specification Version 3.0*. 2007, Modelica.Association. www: <http://www.modelica.org/>. Last Accessed: 2008.
 - [101] Modelica-Association, *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Tutorial and Design Rationale Version 2.0*. 2005. www: <http://www.modelica.org/>. Last Accessed: 2005.
 - [102] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. *A Pattern Matching Compiler for Multiple Target Languages*. in *Compiler Construction, part of 12th Joint European International Conferences on Theory and Practice of Software (ETAPS)*. 2003. Warsaw, Poland. Springer. Lecture Notes in Computer Science (LNCS) No:2622. p. 61-76
 - [103] Peter D. Mosses, *Modular structural operational semantics*. Journal of Functional Programming and Algebraic Programming. Special issue on SOS., 2004. **60-61**: p. 195-228.

- [104] D. Musser and A. Stepanov. *Generic Programming*. in *ISSAC: the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation*. 1988
- [105] Hanspeter Mössenböck, Markus Löberbauer, and Albrecht Wöß, *The Compiler Generator Coco/R*. 2000, University of Linz. www: <http://www.ssw.uni-linz.ac.at/coco/>. Last Accessed: 2008.
- [106] Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages*, Department of Computer and Information Science, Linköping University, 1998, PhD. Thesis
- [107] André Nordwig. *Formal Integration of Structural Dynamics into the Object-Oriented Modeling of Hybrid Systems*. in *Proceedings of the 16th European Simulation Multiconference on Modelling and Simulation*. 2002. Fachhochschule Darmstadt, Darmstadt, Germany. SCS Europe. p. 128-134
- [108] Johann Oberleitner and Thomas Gschwind. *Composing distributed components with the Component Workbench*. 2002. Springer-Verlag. Lecture Notes in Computer Science (LNCS) No:2596
- [109] Martin Odersky, *Raising Your Abstraction: In Defense of Pattern Matching*. 2006. www: <http://www.artima.com/weblogs/viewpost.jsp?thread=166742>. Last Accessed: 2008.
- [110] Martin Odersky and Philip Wadler. *Pizza into Java: Translating Theory into Practice*. in *Proceedings of Principles of Programming Languages (POPL)*. 1997. Paris, France. ACM, New York, NY, USA
- [111] OMG, *CORBA, XML and XMI Resource Page*, Object Management Group. www: <http://www.omg.org/xml/>. Last Accessed: 2008.
- [112] OMG, *Meta-Object Facility (MOF)*, Object Management Group. www: <http://www.omg.com/mof>. Last Accessed: 2008.
- [113] OMG, *Model Driven Architecture (MDA)*, Object Management Group. www: <http://www.omg.com/mda>. Last Accessed: 2008.
- [114] OMG, *System Modeling Language (SysML)*, Object Management Group. www: <http://www.omg-sysml.org>. Last Accessed: 2008.
- [115] OMG, *Unified Modeling Language (UML)*, Object Management Group. www: <http://www.omg.com/uml>. Last Accessed: 2008.
- [116] Terence Parr, *ANTLR Practical Computer Language Recognition and Translation*. 2005. www: <http://www.antlr.org/book/>. Last Accessed: 2008.

-
- [117] PELAB, *Relational Meta-Language (RML) Environment*. 1994-2008, Programming Environments Laboratory (PELAB). www: <http://www.ida.liu.se/~pelab/rml>. Last Accessed: 2008.
 - [118] PELAB, *Open Modelica System*. 2002-2008, Programming Environments Laboratory. www: <http://www.openmodelica.org>. Last Accessed: 2008.
 - [119] PELAB, *Modelica Development Tooling (MDT)*. 2006-2008, PELAB. www: <http://www.ida.liu.se/~pelab/modelica/OpenModelica/MDT/>. Last Accessed: 2008.
 - [120] Mikael Pettersson. *Compiling Natural Semantics*, Department of Computer and Information Science, Linköping University, 1995, PhD Thesis No: 413
 - [121] Mikael Pettersson. *Portable Debugging and Profiling*. in *7th International Conference on Compiler Construction*. 1998. Lisbon, Portugal. Springer-Verlag. Lecture Notes in Computer Science (LNCS) No:1383
 - [122] Mikael Pettersson, *Compiling Natural Semantics*. Lecture Notes in Computer Science (LNCS). Vol. 1549. 1999: Springer-Verlag.
 - [123] Mikael Pettersson and Peter Fritzson. *DML - A Meta-language and System for the Generation of Practical and Efficient Compilers from Denotational Specifications*. in *the 1992 International Conference on Computer Languages*. 1992. Oakland, California
 - [124] Benjamin C. Pierce, *Types and Programming Languages*. 2002, Massachusetts, CA, USA: The MIT Press, Massachusetts Institute of Technology. ISBN: 0-262-16209-1.
 - [125] Gordon Plotkin, *A structural approach to operational semantics*. 1981, Århus University: Århus, Denmark.
 - [126] Adrian Pop and Peter Fritzson. *ModelicaXML: A Modelica XML representation with Applications*. in *3rd International Modelica Conference*. 2003. Linköping, Sweden. (<http://www.modelica.org>). www: ModelicaXML Tools: <http://www.ida.liu.se/~adrpo/modelica/>.
 - [127] Adrian Pop and Peter Fritzson. *Debugging Natural Semantics Specifications*. in *Sixth International Symposium on Automated and Analysis-Driven Debugging*. 2005. Monterey, California
 - [128] Adrian Pop and Peter Fritzson. *A Portable Debugger for Algorithmic Modelica Code*. in *4th International Modelica Conference (Modelica2005)*. 2005. Hamburg-Harburg, Germany. (<http://www.modelica.org>)

- [129] Adrian Pop and Peter Fritzson. *An Eclipse-based Integrated Environment for Developing Executable Structural Operational Semantics Specifications*. in *3rd Workshop on Structural Operational Semantics*. 2006. Bonn, Germany. Elsevier Science. Electronic Notes in Theoretical Computer Science (ENTCS) No:175, Issue 1. p. 71-75
- [130] Adrian Pop and Peter Fritzson. *MetaModelica: A Unified Equation-Based Semantical and Mathematical Modeling Language*. in *7th Joint Modular Languages Conference*. 2006. Oxford, UK. Springer. Lecture Notes in Computer Science (LNCS) No:4228. p. 211-229
- [131] Adrian Pop, Peter Fritzson, Andreas Remar, Elmir Jagudin, and David Akhvlediani. *OpenModelica Development Environment with Eclipse Integration for Browsing, Modeling and Debugging*. in *The 5th International Modelica Conference*. 2006. Vienna, Austria. (<http://www.modelica.org>)
- [132] Adrian Pop, Olof Johansson, and Peter Fritzson. *An Integrated Framework for Model-Driven Design and Development using Modelica*. in *the 45th Conference on Simulation and Modelling (SIMS 2004)*. 2004. Copenhagen, Denmark. www: <http://www.scansims.org/sims2004/index.htm>.
- [133] Adrian Pop, Ilie Savga, Uwe Aßmann, and Peter Fritzson. *Composition of XML dialects: A ModelicaXML case study*. in *Software Composition Workshop 2004, affiliated with ETAPS 2004*. 2004. Barcelona, Spain. Elsevier. Electronic Notes in Theoretical Computer Science (ENTCS) No:114. p. 137-152. www: <http://www.elsevier.com/locate/issn/15710661>.
- [134] Adrian Pop, Kristian Stavåker, and Peter Fritzson. *Exception Handling for Modelica*. in *6th International Modelica Conference*. 2008. Bielefeld, Germany. (<http://www.modelica.org>)
- [135] Bernard Pope and Lee Naish. *Practical aspects of Declarative Debugging in Haskell 98*. in *5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. 2003. Uppsala, Sweden. p. 230-240
- [136] Dave Rager, *OWL Validator*. 2003. www: <http://owl.bbn.com/validator/#www>. Last Accessed: 2005.
- [137] Johan Ringström, Peter Fritzson, and Mikael Pettersson. *Generating an Efficient Compiler for a Data Parallel Language from Denotational Specifications*. in *Int. Conf. of Compiler Construction*. 1994. Edinburgh. Springer Verlag. LNCS 786
- [138] Peter van Roy and Seif Haridi, *Concepts, Techniques, and Models of Computer Programming*. 2004, Cambridge, MA, USA: MIT University Press. ISBN: 0-262-22069-5.

-
- [139] RuleML, *The Rule Markup Initiative*, maintained by Harold Boley and Said Tabet. www: <http://www.ruleml.org/>. Last Accessed: 2008.
 - [140] Levon Saldamli. *PDEModelica - Towards a High-Level Language for Modeling with Partial Differential Equations*, Department of Computer and Information Science, Linköping University, 2002, Licenciate Thesis
 - [141] Levon Saldamli, Bernhard Bachmann, Peter Fritzson, and Hansjürg Wiesmann. *A Framework for Describing and Solving PDE Models in Modelica*. in *4th International Modelica Conference*. 2005. Hamburg-Harburg. (<http://www.modelica.org>)
 - [142] Levon Saldamli, Peter Fritzson, and Bernhard Bachmann. *Extending Modelica for Partial Differential Equations*. in *2nd International Modelica Conference*. 2002. Munich, Germany
 - [143] Erik Sandewall, *Programming in an Interactive Environment: the "Lisp" Experience*. ACM Computing Surveys (CSUR), 1978. **10**(1): p. 35-71.
 - [144] Ilie Savga, Adrian Pop, and Peter Fritzson, *Deriving a Component Model from a Language Specification: an Example Using Natural Semantics*. 2004, Linköping University: Linköping. www: <http://www.ida.liu.se/~adrpo/reports>.
 - [145] Stefan Schonger, Elke Pulvermüller, and Stefan Sarstedt. *Aspect-Oriented Programming and Component Weaving: Using XML Representations of Abstract Syntax Trees*. in *Second Workshop on Aspect-Oriented Software Development (In: Technical Report No. IAI-TR-2002-1)*. 2002. Rheinische Friedrich-Wilhelms-Universität Bonn, Institut für Informatik III. p. 59-64
 - [146] SemanticWebCommunity, *Semantic Web Community Portal*, maintained by Stefan Decker and Michael Sintek. www: <http://www.semanticweb.org/>. Last Accessed: 2008.
 - [147] SICS, *SICStus Prolog Website*, Swedish.Institute.of.Computer.Science. www: <http://www.sics.se/sicstus/>. Last Accessed: 2008.
 - [148] SML/NJ-Fellowship, *Standard ML of New Jersey*. 2004-2008. www: <http://www.smlnj.org/>. Last Accessed: 2008.
 - [149] Kristian Stavåker, Adrian Pop, and Peter Fritzson. *Compiling and Using Pattern Matching in Modelica*. in *6th International Modelica Conference*. 2008. Bielefeld, Germany. (<http://www.modelica.org>)
 - [150] Bjarne Stroustrup, *The C++ Programming Language: Special Edition*. 3rd Edition ed. 2000: Addison-Wesley. ISBN: 0-201-88954-4.

- [151] SWI-Prolog, *SWI-Prolog Website*, University of Amsterdam. [www: http://www.swi-prolog.org/](http://www.swi-prolog.org/). Last Accessed: 2006.
- [152] Michael Tiller, *Introduction to Physical Modeling with Modelica*. 2001: Kluwer Academic Publishers.
- [153] Michael M. Tiller, *Introduction to Physical Modeling with Modelica*. 2001: Kluwer Academic Publishers.
- [154] Andrew Tolmach and Andrew W. Appel, *A debugger for Standard ML*. Journal of Functional Programming, 1995. **5**(2).
- [155] Andrew P. Tolmach. *Debugging Standard ML*, Princeton University, 1992, PhD. Thesis
- [156] Skander Turki and Thierry Soriano. *A SysML Extension for Bond Graphs Support*. in *Proceeding of the International Conference on Technology and Automation (ICTA)*. 2005. Thessaloniki, Greece
- [157] W3C, *Document Object Model (DOM)*, World Wide Web Consortium (W3C). [www: http://www.w3.org/DOM/](http://www.w3.org/DOM/). Last Accessed: 2008.
- [158] W3C, *Extensible Markup Language (XML)*, Word Wide Web Consortium (W3C). [www: http://www.w3.org/XML/](http://www.w3.org/XML/). Last Accessed: 2008.
- [159] W3C, *The Extensible Stylesheet Language Family (XSL/XSLT/XPath/XSL-FO)*, Word Wide Web Consortium (W3C). [www: http://www.w3.org/Style/XSL/](http://www.w3.org/Style/XSL/). Last Accessed: 2008.
- [160] W3C, *RDF Vocabulary Description Language (RDFS/RDF-Schema)*, World Wide Web Consortium (W3C). [www: http://www.w3.org/TR/rdf-schema/](http://www.w3.org/TR/rdf-schema/). Last Accessed: 2008.
- [161] W3C, *Resource Description Framework (RDF)*, Word Wide Web Consortium (W3C). [www: http://www.w3c.org/RDF/](http://www.w3c.org/RDF/). Last Accessed: 2008.
- [162] W3C, *Semantic Web*, World Wide Web Consortium (W3C). [www: http://www.w3.org/2001/sw/](http://www.w3.org/2001/sw/). Last Accessed: 2008.
- [163] W3C, *Standard Generalized Markup Language (SGML)*, World Wide Web Consortium (W3C). [www: http://www.w3.org/MarkUp/SGML/](http://www.w3.org/MarkUp/SGML/). Last Accessed: 2008.
- [164] W3C, *Web Ontology Language (OWL)*, Word Wide Web Consortium (W3C). [www: http://www.w3.org/TR/2003/CR-owl-features-20030818/](http://www.w3.org/TR/2003/CR-owl-features-20030818/). Last Accessed: 2008.

-
- [165] W3C, *Web Ontology Language (OWL) Overview*, World Wide Web Consortium (W3C). www: <http://www.w3.org/TR/owl-features/>. Last Accessed: 2008.
 - [166] W3C, *XML Query (XQuery)*, Word Wide Web Consortium (W3C). www: <http://www.w3.org/XML/Query>. Last Accessed: 2008.
 - [167] W3C, *XML Schema (XSchema)*, Word Wide Web Consortium (W3C). www: <http://www.w3.org/XML/Schema>. Last Accessed: 2008.
 - [168] Yves Vanderperren and Wim Dehane. *SysML and Systems Engineering Applied to UML-Based SoC Design*. in *Proceeding of the 2nd UML-SoC Workshop at 42nd Design and Automation Conference (DAC)*. 2005. Anaheim, USA
 - [169] Yves Vanderperren and Wim Dehane. *From UML/SysML to Matlab/Simulink: Current State and Future Perspectives*. in *The Conference on Design, Automation and Test in Europe (DATE)*. 2006. Munich, Germany. p. 93-93
 - [170] Hans Vangheluwe and Juan de Lara. *Domain-Specific Modelling with AToM3*. in *4th OOPSLA Workshop on Domain-Specific Modeling*. 2004. Vancouver, Canada
 - [171] Teitelman Warren, *INTERLISP Reference Manual*. 1974, Xerox Palo Alto Research Center: Palo Alto, CA.
 - [172] Christopher Welty. *An Integrated Representation for Software Development and Discovery*, Rensselaer Polytechnic Institute, 1995, PhD Thesis
 - [173] Lars Viklund, Johan Herber, and Peter Fritzson. *The implementation of ObjectMath - a high-level programming environment for scientific computing*. in *Compiler Construction - 4th International Conference (CC'92)*. 1992. Springer-Verlag. Lecture Notes in Computer Science (LNCS) No:641. p. 312-318
 - [174] Paul R. Wilson. *Uniprocessor Garbage Collection Techniques*. in *The International Workshop on Memory Management*. 1994. Lecture Notes In Computer Science (LNCS) No:637
 - [175] Wolfram, *Mathematica*. 2008. www: <http://www.wolfram.com/>. Last Accessed: 2008.
 - [176] Matthias Zenger and Martin Odersky. *Extensible Algebraic Datatypes with Defaults*. in *Proceedings of the International Conference on Functional Programming*. 2001. Firenze, Italy

Dissertations

Linköping Studies in Science and Technology

- No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.
- No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.
- No 18 **Mats Cedwall:** Semantisk analys av process-beskrivningar i naturligt språk, 1977, ISBN 91-7372-168-9.
- No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.
- No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91-7372-232-4.
- No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.
- No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.
- No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.
- No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.
- No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.
- No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.
- No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91-7372-527-7.
- No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.
- No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.
- No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation, 1984, ISBN 91-7372-801-2.
- No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372-805-5.
- No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.
- No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.
- No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.
- No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.
- No 192 **Dimitar Driankov:** Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.
- No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.
- No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.
- No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.
- No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.
- No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.
- No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies, 1991, ISBN 91-7870-784-6.
- No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.
- No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.
- No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.
- No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.
- No 270 **Ralph Rönnquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.
- No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.
- No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.
- No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.
- No 281 **Christer Bäckström:** Computational Complexity

- of Reasoning about Plans, 1992, ISBN 91-7870-979-2.
- No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.
- No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.
- No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2.
- No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.
- No 338 **Simin Nadjm-Tehrani:** Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.
- No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea/ABB 1968-1993, 1995, ISBN 91-7871-494-X.
- No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.
- No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.
- No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.
- No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.
- No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.
- No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.
- No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.
- No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.
- No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.
- No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.
- No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.
- No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.
- No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.
- No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms, 1997, ISBN 91-7871-857-0.
- No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.
- No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.
- No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.
- No 485 **Göran Forslund:** Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.
- No 494 **Martin Sköld:** Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.
- No 495 **Hans Olsén:** Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.
- No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.
- No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.
- No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Languages from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.
- No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.
- No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.
- No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.
- No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.
- No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.
- No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis, 1998, ISBN 91-7219-369-7.
- No 561 **Ling Lin:** Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.
- No 563 **Eva L Ragnemalm:** Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.
- No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.
- No 582 **Vanja Josifovski:** Design, Implementation and

- Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.
- No 589 **Rita Kovordányi:** Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.
- No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.
- No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.
- No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.
- No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.
- No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.
- No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.
- No 598 **Johan Jenvald:** Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.
- No 607 **Magnus Merkel:** Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.
- No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.
- No 613 **Man Lin:** Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.
- No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.
- No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.
- No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.
- No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.
- No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.
- No 688 **Marcus Bjärelund:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.
- No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.
- No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN-91-7373-126-9.
- No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91 7373 207 9.
- No 725 **Tim Heyer:** Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91 7373 208 7.
- No 726 **Pär Carlshamre:** A Usability Perspective on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91 7373 212 5.
- No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91 7373 258 3.
- No 745 **Johan Åberg:** Live Help Systems: An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.
- No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.
- No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.
- No 747 **Anneli Hagdahl:** Development of IT-supported Inter-organisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.
- No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.
- No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.
- No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.
- No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.
- No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.
- No 774 **Choong-ho Yi:** Modelling Object-Oriented Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.
- No 779 **Mathias Broxvall:** A Study in the Computational Complexity of Temporal Reasoning, 2002, ISBN 91-7373-440-3.
- No 793 **Asmus Pandikow:** A Generic Principle for Enabling Interoperability of Structured and Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.
- No 785 **Lars Hult:** Publika Informationstjänster. En studie av den Internetbaserade encyklopedins bruksegenskaper, 2003, ISBN 91-7373-461-6.
- No 800 **Lars Taxén:** A Framework for the Coordination of Complex Systems' Development, 2003, ISBN 91-7373-604-X
- No 808 **Klas Gäre:** Tre perspektiv på förväntningar och förändringar i samband med införande av informa-

- tionsystem, 2003, ISBN 91-7373-618-X.
- No 821 **Mikael Kindborg:** Concurrent Comics - programming of social agents by children, 2003, ISBN 91-7373-651-1.
- No 823 **Christina Ölvingson:** On Development of Information Systems with GIS Functionality in Public Health Informatics: A Requirements Engineering Approach, 2003, ISBN 91-7373-656-2.
- No 828 **Tobias Ritzau:** Memory Efficient Hard Real-Time Garbage Collection, 2003, ISBN 91-7373-666-X.
- No 833 **Paul Pop:** Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems, 2003, ISBN 91-7373-683-X.
- No 852 **Johan Moe:** Observing the Dynamic Behaviour of Large Distributed Systems to Improve Development and Testing - An Emperical Study in Software Engineering, 2003, ISBN 91-7373-779-8.
- No 867 **Erik Herzog:** An Approach to Systems Engineering Tool Data Representation and Exchange, 2004, ISBN 91-7373-929-4.
- No 872 **Aseel Berglund:** Augmenting the Remote Control: Studies in Complex Information Navigation for Digital TV, 2004, ISBN 91-7373-940-5.
- No 869 **Jo Skåmedal:** Telecommuting's Implications on Travel and Travel Patterns, 2004, ISBN 91-7373-935-9.
- No 870 **Linda Askenäs:** The Roles of IT - Studies of Organising when Implementing and Using Enterprise Systems, 2004, ISBN 91-7373-936-7.
- No 874 **Annika Flycht-Eriksson:** Design and Use of Ontologies in Information-Providing Dialogue Systems, 2004, ISBN 91-7373-947-2.
- No 873 **Peter Bunus:** Debugging Techniques for Equation-Based Languages, 2004, ISBN 91-7373-941-3.
- No 876 **Jonas Mellin:** Resource-Predictable and Efficient Monitoring of Events, 2004, ISBN 91-7373-956-1.
- No 883 **Magnus Bång:** Computing at the Speed of Paper: Ubiquitous Computing Environments for Healthcare Professionals, 2004, ISBN 91-7373-971-5
- No 882 **Robert Eklund:** Disfluency in Swedish human-human and human-machine travel booking dialogues, 2004. ISBN 91-7373-966-9.
- No 887 **Anders Lindström:** English and other Foreign Linguistic Elements in Spoken Swedish. Studies of Productive Processes and their Modelling using Finite-State Tools, 2004, ISBN 91-7373-981-2.
- No 889 **Zhiping Wang:** Capacity-Constrained Production-inventory systems - Modellling and Analysis in both a traditional and an e-business context, 2004, ISBN 91-85295-08-6.
- No 893 **Pernilla Qvarfordt:** Eyes on Multimodal Interaction, 2004, ISBN 91-85295-30-2.
- No 910 **Magnus Kald:** In the Borderland between Strategy and Management Control - Theoretical Framework and Empirical Evidence, 2004, ISBN 91-85295-82-5.
- No 918 **Jonas Lundberg:** Shaping Electronic News: Genre Perspectives on Interaction Design, 2004, ISBN 91-85297-14-3.
- No 900 **Mattias Arvola:** Shades of use: The dynamics of interaction design for sociable use, 2004, ISBN 91-85295-42-6.
- No 920 **Luis Alejandro Cortés:** Verification and Scheduling Techniques for Real-Time Embedded Systems, 2004, ISBN 91-85297-21-6.
- No 929 **Diana Szentivanyi:** Performance Studies of Fault-Tolerant Middleware, 2005, ISBN 91-85297-58-5.
- No 933 **Mikael Cäker:** Management Accounting as Constructing and Opposing Customer Focus: Three Case Studies on Management Accounting and Customer Relations, 2005, ISBN 91-85297-64-X.
- No 937 **Jonas Kvarnström:** TALplanner and Other Extensions to Temporal Action Logic, 2005, ISBN 91-85297-75-5.
- No 938 **Bourhane Kadmiry:** Fuzzy Gain-Scheduled Visual Servoing for Unmanned Helicopter, 2005, ISBN 91-85297-76-3.
- No 945 **Gert Jervan:** Hybrid Built-In Self-Test and Test Generation Techniques for Digital Systems, 2005, ISBN: 91-85297-97-6.
- No 946 **Anders Arpteg:** Intelligent Semi-Structured Information Extraction, 2005, ISBN 91-85297-98-4.
- No 947 **Ola Angelsmark:** Constructing Algorithms for Constraint Satisfaction and Related Problems - Methods and Applications, 2005, ISBN 91-85297-99-2.
- No 963 **Calin Curescu:** Utility-based Optimisation of Resource Allocation for Wireless Networks, 2005. ISBN 91-85457-07-8.
- No 972 **Björn Johansson:** Joint Control in Dynamic Situations, 2005, ISBN 91-85457-31-0.
- No 974 **Dan Lawesson:** An Approach to Diagnosability Analysis for Interacting Finite State Systems, 2005, ISBN 91-85457-39-6.
- No 979 **Claudiu Duma:** Security and Trust Mechanisms for Groups in Distributed Services, 2005, ISBN 91-85457-54-X.
- No 983 **Sorin Manolache:** Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour, 2005, ISBN 91-85457-60-4.
- No 986 **Yuxiao Zhao:** Standards-Based Application Integration for Business-to-Business Communications, 2005, ISBN 91-85457-66-3.
- No 1004 **Patrik Haslum:** Admissible Heuristics for Automated Planning, 2006, ISBN 91-85497-28-2.
- No 1005 **Aleksandra Tešanovic:** Developing Re-usable and Reconfigurable Real-Time Software using Aspects and Components, 2006, ISBN 91-85497-29-0.
- No 1008 **David Dinka:** Role, Identity and Work: Extending the design and development agenda, 2006, ISBN 91-85497-42-8.
- No 1009 **Iakov Nakhimovski:** Contributions to the Modeling and Simulation of Mechanical Systems with Detailed Contact Analysis, 2006, ISBN 91-85497-43-X.
- No 1013 **Wilhelm Dahllöf:** Exact Algorithms for Exact Satisfiability Problems, 2006, ISBN 91-85523-97-6.
- No 1016 **Levon Saldamli:** PDEModelica - A High-Level Language for Modeling with Partial Differential Equations, 2006, ISBN 91-85523-84-4.
- No 1017 **Daniel Karlsson:** Verification of Component-based Embedded System Designs, 2006, ISBN 91-85523-79-8.

- No 1018 **Ioan Chisalita:** Communication and Networking Techniques for Traffic Safety Systems, 2006, ISBN 91-85523-77-1.
- No 1019 **Tarja Susi:** The Puzzle of Social Activity - The Significance of Tools in Cognition and Cooperation, 2006, ISBN 91-85523-71-2.
- No 1021 **Andrzej Bednarski:** Integrated Optimal Code Generation for Digital Signal Processors, 2006, ISBN 91-85523-69-0.
- No 1022 **Peter Aronsson:** Automatic Parallelization of Equation-Based Simulation Programs, 2006, ISBN 91-85523-68-2.
- No 1030 **Robert Nilsson:** A Mutation-based Framework for Automated Testing of Timeliness, 2006, ISBN 91-85523-35-6.
- No 1034 **Jon Edvardsson:** Techniques for Automatic Generation of Tests from Programs and Specifications, 2006, ISBN 91-85523-31-3.
- No 1035 **Vaida Jakoniene:** Integration of Biological Data, 2006, ISBN 91-85523-28-3.
- No 1045 **Genevieve Gorrell:** Generalized Hebbian Algorithms for Dimensionality Reduction in Natural Language Processing, 2006, ISBN 91-85643-88-2.
- No 1051 **Yu-Hsing Huang:** Having a New Pair of Glasses - Applying Systemic Accident Models on Road Safety, 2006, ISBN 91-85643-64-5.
- No 1054 **Åsa Hedenskog:** Perceive those things which cannot be seen - A Cognitive Systems Engineering perspective on requirements management, 2006, ISBN 91-85643-57-2.
- No 1061 **Cécile Åberg:** An Evaluation Platform for Semantic Web Technology, 2007, ISBN 91-85643-31-9.
- No 1073 **Mats Grindal:** Handling Combinatorial Explosion in Software Testing, 2007, ISBN 978-91-85715-74-9.
- No 1075 **Almut Herzog:** Usable Security Policies for Runtime Environments, 2007, ISBN 978-91-85715-65-7.
- No 1079 **Magnus Wahlström:** Algorithms, measures, and upper bounds for satisfiability and related problems, 2007, ISBN 978-91-85715-55-8.
- No 1083 **Jesper Andersson:** Dynamic Software Architectures, 2007, ISBN 978-91-85715-46-6.
- No 1086 **Ulf Johansson:** Obtaining Accurate and Comprehensible Data Mining Models - An Evolutionary Approach, 2007, ISBN 978-91-85715-34-3.
- No 1089 **Traian Pop:** Analysis and Optimisation of Distributed Embedded Systems with Heterogeneous Scheduling Policies, 2007, ISBN 978-91-85715-27-5.
- No 1091 **Gustav Nordh:** Complexity Dichotomies for CSP-related Problems, 2007, ISBN 978-91-85715-20-6.
- No 1106 **Per Ola Kristensson:** Discrete and Continuous Shape Writing for Text Entry and Control, 2007, ISBN 978-91-85831-77-7.
- No 1110 **He Tan:** Aligning Biomedical Ontologies, 2007, ISBN 978-91-85831-56-2.
- No 1112 **Jessica Lindblom:** Minding the body - Interacting socially through embodied action, 2007, ISBN 978-91-85831-48-7.
- No 1113 **Pontus Wärnestål:** Dialogue Behavior Management in Conversational Recommender Systems, 2007, ISBN 978-91-85831-47-0.
- No 1120 **Thomas Gustafsson:** Management of Real-Time Data Consistency and Transient Overloads in Embedded Systems, 2007, ISBN 978-91-85831-33-3.
- No 1127 **Alexandru Andrei:** Energy Efficient and Predictable Design of Real-time Embedded Systems, 2007, ISBN 978-91-85831-06-7.
- No 1139 **Per Wikberg:** Eliciting Knowledge from Experts in Modeling of Complex Systems: Managing Variation and Interactions, 2007, ISBN 978-91-85895-66-3.
- No 1143 **Mehdi Amirijoo:** QoS Control of Real-Time Data Services under Uncertain Workload, 2007, ISBN 978-91-85895-49-6.
- No 1150 **Sanny Syberfeldt:** Optimistic Replication with Forward Conflict Resolution in Distributed Real-Time Databases, 2007, ISBN 978-91-85895-27-4.
- No 1155 **Beatrice Alenljung:** Envisioning a Future Decision Support System for Requirements Engineering - A Holistic and Human-centred Perspective, 2008, ISBN 978-91-85895-11-3.
- No 1156 **Artur Wilk:** Types for XML with Application to Xcerpt, 2008, ISBN 978-91-85895-08-3.
- No 1183 **Adrian Pop:** Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages, 2008, ISBN 978-91-7393-895-2.
- Linköping Studies in Statistics**
- No 9 **Davood Shahsavani:** Computer Experiments Designed to Explore and Approximate Complex Deterministic Models, 2008, ISBN 978-91-7393-976-8.
- Linköping Studies in Information Science**
- No 1 **Karin Axelsson:** Metodisk systemstrukturerings- att skapa samstämmighet mellan informationssystemarkitektur och verksamhet, 1998. ISBN-9172-19-296-8.
- No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998. ISBN-9172-19-299-2.
- No 3 **Anders Avdic:** Användare och utvecklare - om anveckling med kalkylprogram, 1999. ISBN-91-7219-606-8.
- No 4 **Owen Eriksson:** Kommunikationskvalitet hos informationssystem och affärsprocesser, 2000. ISBN 91-7219-811-7.
- No 5 **Mikael Lind:** Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X
- No 6 **Ulf Melin:** Koordination och informationssystem i företag och nätverk, 2002, ISBN 91-7373-278-8.
- No 7 **Pär J. Ågerfalk:** Information Systems Actability - Understanding Information Technology as a Tool for Business Action and Communication, 2003, ISBN 91-7373-628-7.
- No 8 **Ulf Seigerroth:** Att förstå och förändra systemutvecklingsverksamheter - en taxonomi

för metautveckling, 2003, ISBN91-7373-736-4.

- No 9 **Karin Hedström:** Spår av datoriseringens värden - Effekter av IT i äldreomsorg, 2004, ISBN 91-7373-963-4.
- No 10 **Ewa Braf:** Knowledge Demanded for Action - Studies on Knowledge Mediation in Organisations, 2004, ISBN 91-85295-47-7.
- No 11 **Fredrik Karlsson:** Method Configuration - method and computerized tool support, 2005, ISBN 91-85297-48-8.
- No 12 **Malin Nordström:** Styrbar systemförvaltning - Att organisera systemförvaltningsverksamhet med hjälp av effektiva förvaltningsobjekt, 2005, ISBN 91-85297-60-7.
- No 13 **Stefan Holgersson:** Yrke: POLIS - Yrkeskunskap, motivation, IT-system och andra förutsättningar för polisarbete, 2005, ISBN 91-85299-43-X.
- No 14 **Benneth Christiansson, Marie-Therese Christiansson:** Mötet mellan process och komponent - mot ett ramverk för en verksamhetsnära kravspecifikation vid anskaffning av komponentbaserade informationssystem, 2006, ISBN 91-85643-22-X.