D.U.: 2308

E.D.S.P.I.C: 594

Ph.D. Thesis

submitted to the École Doctorale des Sciences pour l'Ingénieur
to obtain the title of

Ph.D. in Computer Science

Submitted by ***Luc Touraille***

Application of Model-Driven Engineering and Metaprogramming to DEVS Modeling & Simulation

Thesis supervisors: *Prof. David R.C. Hill*
Dr. Mamadou K. Traoré

Publicly defended on December 7th, 2012 in front of an examination committee composed of:

Reviewers: *Prof. Jean-Pierre Müller*, Cirad, Montpellier
Prof. Gabriel A. Wainer, Carleton University, Ottawa
Supervisors: *Prof. David R.C. Hill*, Université Blaise Pascal, Clermont-Ferrand
Dr. Mamadou K. Traoré, Université Blaise Pascal, Clermont-Ferrand
Examiners *Dr. Alexandre Muzy*, Università di Corsica Pasquale Paoli, Corte
Prof. Bernard P. Zeigler, University of Arizona, Tucson

D.U.: 2308

E.D.S.P.I.C: 594

Ph.D. Thesis

submitted to the École Doctorale des Sciences pour l'Ingénieur
to obtain the title of

Ph.D. in Computer Science

Submitted by ***Luc Touraille***

Application of Model-Driven Engineering and Metaprogramming to DEVS Modeling & Simulation

---

Thesis supervisors: *Prof. David R.C. Hill*
Dr. Mamadou K. Traoré

Publicly defended on December 7th, 2012 in front of an examination committee composed of:

Reviewers: *Prof. Jean-Pierre Müller*, Cirad, Montpellier
Prof. Gabriel A. Wainer, Carleton University, Ottawa

Supervisors: *Prof. David R.C. Hill*, Université Blaise Pascal, Clermont-Ferrand
Dr. Mamadou K. Traoré, Université Blaise Pascal, Clermont-Ferrand

Examiners *Dr. Alexandre Muzy*, Università di Corsica Pasquale Paoli, Corte
Prof. Bernard P. Zeigler, University of Arizona, Tucson

Abstract

The multiplication of software environments supporting DEVS Modeling & Simulation is becoming a hindrance to scientific collaboration. Indeed, the use of disparate tools in the community makes the exchange, reuse and comparison of models very difficult, preventing practitioners from building on previous works to devise models of ever-increasing complexity.

Tool interoperability is not the only issue raised by the need for models of higher and higher complexity. As models grow, their development becomes more error-prone, and their simulation becomes more resource-consuming. Consequently, it is necessary to devise techniques for improving simulators performance and for providing thorough model verification to assist the practitioner during model design.

In this thesis, we propose two innovative approaches for DEVS Modeling & Simulation that tackle the aforementioned issues. The first contribution described in this document is a model-driven environment for modeling systems with the DEVS formalism, named SimStudio. This environment relies on Model-Driven Engineering to provide a high-level framework where practitioners can create, edit and visualize models, and automatically generate multiple artifacts, most notably model specifications compatible with various DEVS simulators. The core of SimStudio is a platform-independent metamodel of the DEVS formalism, which provides a pivot format for DEVS models. Based on this metamodel, we developed several model verification features as well as many model transformations that can be used to automatically generate documentation, diagrams or code targeting various DEVS platforms. Thus, SimStudio gives a proof of concept of the integration capabilities that a DEVS standard would provide; as a matter of fact, the metamodel presented in this thesis could possibly serve as a basis for such a standard.

The second contribution of this thesis is DEVS-MetaSimulator (DEVS-MS), a DEVS library relying on metaprogramming to generate simulation executables that are specialized and optimized for the model they handle. To do so, the library performs many computations during compilation, resulting in a simulation code where most overhead have been eliminated. The tests we conducted showed that the generated programs were very

efficient, but the performance gain is not the only feature of DEVS-MS. Indeed, through metaprogramming, DEVS-MS can also assert the correctness of models by verifying model characteristics at compile-time, detecting and reporting modeling errors very early in the development cycle and with better confidence than what could be achieved with classical testing.

Keywords: DEVS, Modeling & Simulation, Model-Driven Engineering, Metaprogramming

Résumé

La multiplication des environnements logiciels pour la Modélisation & Simulation DEVS pose un problème de collaboration à la communauté scientifique. En effet, l'utilisation d'outils disparates rend l'échange, la réutilisation et la comparaison de modèles très difficiles, empêchant les scientifiques de s'appuyer sur des travaux précédents pour construire leurs modèles.

L'interopérabilité des outils n'est pas le seul problème soulevé par le besoin de modèles toujours plus complexes. Au fur et à mesure que les modèles grossissent, leur développement devient plus difficile, notamment en termes de détection des erreurs de conception. D'autre part, la simulation de ces modèles demande de plus en plus de ressources. Par conséquent, il est nécessaire de concevoir des techniques pour améliorer la performance des simulateurs et pour fournir des fonctionnalités de vérification de modèle afin d'assister les scientifiques dans la conception de leurs modèles.

Dans cette thèse, nous proposons deux approches innovantes pour la M&S DEVS qui s'attaquent aux problèmes susmentionnés. La première contribution décrite dans ce document est un environnement basé sur les modèles pour modéliser des systèmes avec le formalisme DEVS, intitulé SimStudio. Cet environnement repose sur l'Ingénierie Dirigée par les Modèles pour fournir un cadriciel de haut niveau dans lequel les scientifiques peuvent créer, éditer et visualiser des modèles, et générer automatiquement un ensemble d'artefacts, notamment des spécifications de modèles compatibles avec différents simulateurs DEVS. Le noyau de SimStudio est un métamodèle de DEVS, indépendant de toute plateforme, qui fournit un format pivot pour la représentation des modèles DEVS. En se basant sur ce métamodèle, nous avons développé plusieurs fonctionnalités de vérification de modèle ainsi que plusieurs transformations de modèle pouvant être utilisées pour générer automatiquement de la documentation, des diagrammes ou du code ciblant diverses plateformes DEVS. Ainsi, SimStudio fournit une preuve de concept des capacités d'intégration qu'un standard DEVS pourrait fournir ; en fait, le métamodèle présenté dans cette thèse pourrait potentiellement servir de base de réflexion pour un tel standard.

La seconde contribution de cette thèse est DEVS-MetaSimulateur (DEVS-MS), une bibliothèque DEVS qui utilise la métaprogrammation pour générer des exécutables de simulation spécialisés et optimisés pour le modèle qu'ils traitent. Pour ce faire, la bibliothèque effectue un grand nombre d'opérations durant la compilation, résultant en un code de simulation où une grande partie de l'overhead de simulation a été éliminé. Les tests que nous avons effectués ont montré que les programmes générés étaient très efficaces, mais le gain de performance n'est pas la seule caractéristique intéressante de DEVS-MS. En effet, grâce à la métaprogrammation, DEVS-MS peut également partiellement vérifier à la compilation que les modèles sont corrects, c'est-à-dire que leurs caractéristiques sont bien conformes au formalisme DEVS. Les erreurs de modélisation sont ainsi détectées et signalées très tôt dans le cycle de développement, et avec un taux de détection bien meilleur que ne le permettrait des tests classiques.

Mots-clés : DEVS, Modélisation & Simulation, Ingénierie Dirigée par les Modèles, Métaprogrammation

Acknowledgments/Remerciements

First of all, I would like to thank Jean-Pierre, Gabriel and Bernie for their accessibility and of course for doing me the honor of being part of my committee.

J'adresse également un grand merci à l'enthousiaste Lisandru, non seulement pour m'avoir fait découvrir la beauté de Cargese mais aussi pour les multiples collaborations que nous avons pu effectuer ensemble, et les trop peu nombreuses soirées que nous avons partagés.

Ce travail n'aurait bien sûr pas été possible sans l'indefectible soutien de Benny et Mams, qui ont su me guider durant ces années tout en me laissant une grande liberté. Chaque discussion avec Mamadou aura été une source d'inspiration pour mes travaux, et de bonne humeur pour ma journée (notamment grâce à ce sourire communicatif qui le caractérise !). Quant à Benny, je n'aurais probablement jamais fini cette thèse sans son « coaching » (en fait, je ne l'aurais probablement même pas commencée !). Merci à eux deux pour leur gentillesse, leur disponibilité, et la totale confiance qu'ils m'ont accordée.

Parmi les personnes qui m'ont apporté au niveau scientifique, je tiens à remercier tout particulièrement Hans : en plus d'être agréables, les moments que j'ai pu passer avec lui ont énormément enrichi ce travail.

Merci également à tous mes collègues du LIMOS et de l'ISIMA pour leur sympathie et pour m'avoir formé en tant qu'étudiant puis accueilli au sein de leur établissement. Je pense notamment à Claude, Philippe, Gilles, Christine, Vincent, Michel, Romuald, Christophe, Christophe, Murielle, Loïc, Bruno, Farouk.

Je serais probablement encore perdu dans les méandres de l'administration sans le secours de Françoise, Béa et Isabelle, dont la compétence n'a d'égale que la serviabilité.

Ces années m'auraient paru une éternité sans la compagnie de Corinne, Susan, Nicolas et de mes « compagnons de galère » Mohieddine, Pierre, Romain, Baraa, Faouzi, Guillaume et Jo P.

Ces remerciements seraient bien incomplets si je ne rendais pas hommage à Jo, voisin de bureau et ami de longue date, partenaire de procrastination, de fistball et parfois de réflexion.

Enfin, je tiens à remercier ma famille et mes amis tout simplement pour avoir été là, et bien sûr Mélanie pour avoir supporté mes absences et mes baisses de moral, et pour continuer chaque jour de rendre la vie plus belle.

Table of contents

Abstract 5

Résumé 7

Acknowledgments/Remerciements 9

Table of contents 11

List of figures 19

List of codes 21

List of tables 25

List of abbreviations 26

Chapter 1. General introduction 31

I. Context 31

II. Objectives 32

III. Outline 34

Chapter 2. Modeling and Simulation with Discrete Event System Specifications 37

I. Introduction 37

II. DEVS formalism and tools 38

II.1. Overview 38

II.1.1. Background 38

II.1.1.1. Hierarchy of specifications 39

II.1.1.2. Fundamental formalisms 41

II.1.2. DEVS as an intermediate language for simulating models 42

II.1. Classic and Parallel DEVS 43

II.1.1. Classic DEVS atomic model 43

II.1.2. Classic DEVS coupled model 46

II.1.3. Parallel DEVS 50

II.2. DEVS simulation algorithms 51
II.2.1. Overall architecture 52
II.2.2. Processor for atomic DEVS models (simulator) 53
II.2.3. Processor for coupled DEVS models (coordinator) 55
II.2.4. Top-level processor (root coordinator) 57
II.3. Existing DEVS tools 58
II.3.1. CD++ 58
II.3.2. DEVSJava 59
II.3.3. James II 59
II.3.4. MIMOSA 60
II.3.5. PythonDEVS 61
II.3.6. Virtual Laboratory Environment (VLE) 61
III. Limits of the current DEVS Modeling & Simulation software 62
III.1. The challenge of standardization and interoperability 62
III.1.1. Standardized interoperability middleware 63
III.1.1.1. Simulator-based interoperability 64
III.1.1.2. Model-based interoperability 66
III.1.2. Standardized model representation 69
III.1.3. Interoperability and standardized representation: two complementary approaches 71
III.2. Possible improvements to current DEVS implementations 72
III.2.1. Performance 72
III.2.2. Model verification 74
IV. Conclusion 77
Chapter 3. Model-Driven Engineering 79
I. Introduction 79

II. Concepts and definitions 80
II.1. What is a model 80
II.2. What is a metamodel 82
II.3. What is a transformation 84
II.4. What is Model-Driven Engineering 85
III. Survey of Model-Driven Engineering implementations 88
III.1. Model-Driven Architecture 88
III.2. Eclipse Modeling Project 90
III.3. Software Factories 91
III.4. Other Modeling Driven Engineering initiatives 95
IV. Eclipse Modeling Project 98
IV.1. Abstract syntax development 100
IV.2. Concrete syntax development 102
IV.2.1. Graphical syntax development 103
IV.2.2. Textual syntax development 105
IV.3. Model transformations 107
IV.3.1. Model-to-model transformations 107
IV.3.2. Model-to-text transformation 108
V. Conclusion 109
Chapter 4. Metaprogramming 111
I. Introduction 111
II. Concepts and definitions 112
III. Survey of metaprogramming techniques 114
III.1. Text generation 114
III.1.1. Data generation 115
III.1.2. Generation of instructions 115

III.2. Domain-Specific Languages 116
III.2.1. Embedded Domain Specific Languages 117
III.3. Multi-stage programming languages 120
III.3.1. C++ Template MetaProgramming 122
III.4. Partial evaluation 124
III.5. Comparison of the different approaches 127
IV. Metaprogramming in C++ 130
IV.1. C++ features used for metaprogramming 131
IV.1.1. Preprocessor 131
IV.1.2. Templates 131
IV.1.3. Substitution Failure Is Not An Error 133
IV.2. Data structures and control flow 135
IV.2.1. Typelists 136
IV.2.2. Metafunctions 137
IV.2.3. Static polymorphism 140
IV.3. Metaprogramming libraries 144
IV.3.1. Boost.Preprocessor 144
IV.3.2. Boost.Type Traits 146
IV.3.3. enable_if 147
IV.3.4. Boost.MPL 148
IV.3.5. Boost.Fusion 153
V. Conclusion 156
Chapter 5. SimStudio, a Model-Driven DEVS Environment 157
I. Introduction 157
II. Application of Model-Driven Engineering to DEVS M&S 158
II.1. SimStudio, an extensible and integrating environment 158

II.2. DEVS metamodel 161
II.2.1. Structure 161
II.2.1.1. Types 162
II.2.1.2. Ports 163
II.2.1.3. State variables 164
II.2.1.4. Atomic models 165
II.2.1.5. Components 166
II.2.1.6. Couplings 167
II.2.1.7. Select 169
II.2.1.8. CoupledModel 171
II.2.2. Behavior 172
II.2.2.1. Previous propositions 172
II.2.2.2. Proposal of a “semi-generic” language 174
II.2.3. Additional aspects to consider 180
II.2.3.1. Parameterization and initialization 180
II.2.3.2. Representation of trajectories 184
II.3. Model-to-model transformations 187
II.3.1. DEVS2SVG: Coupled model diagram generation 187
II.3.2. DEVS2XHTML: Documentation generation 190
II.3.3. DDML2DEVS: Graphical edition of DEVS models 191
II.3.4. Inter-formalism transformations 192
II.4. Code generation 193
III. Application to a switch network model 196
III.1. Model creation/edition 196
III.2. Documentation generation 199
III.1. Code generation 200

III.1.1. Generated CD++ code 201
III.1.2. Generated PyDEVS code 203
III.1.3. Generated DEVS-MS code 205
IV. Conclusion 207
Chapter 6. DEVS-MetaSimulator: Enhancing DEVS Simulation through Metaprogramming 211
I. Introduction 211
II. Rationale behind DEVS-MS: specializing simulators for their models 213
II.1. Generic interface versus specific interface 215
II.2. Hierarchical simulation versus flattened simulation 218
II.3. Sample code of a specialized simulation application 223
II.4. Generic modeling and specific simulation: having the best of both worlds 229
III. Implementation of DEVS-MS with C++ Template MetaProgramming 231
III.1. Message 233
III.2. Models 236
III.2.1. Model 237
III.2.2. AtomicModel 238
III.2.3. CoupledModel 240
III.3. Utility macros 243
III.3.1. DECLARE_PORT/DECLARE_COMPONENT 244
III.3.2. PORTS 244
III.3.3. COMPONENTS 244
III.3.4. COUPLINGS 245
III.3.5. SELECT 245
III.4. Processors 245
III.4.1. Simulator 246
III.4.1.1. Initialization 247

III.4.1.2. Internal transition 247
III.4.1.3. External transition 249
III.4.2. Coordinator 250
III.4.2.1. Initialization 253
III.4.2.2. Internal transition 253
III.4.2.3. Input message and output message 257
III.4.3. Root 260
IV. Sample application: semantic pervasive dual cache 261
IV.1. Model of a semantic pervasive dual cache system 263
IV.2. Implementation in DEVS-MS 265
IV.2.1. RandomSwitch model in DEVS-MS 266
IV.2.2. Proxy model in DEVS-MS 269
IV.3. Results 271
IV.3.1. Performance 271
IV.3.1.1. Execution times 272
IV.3.1.1. Compilation time 275
IV.3.2. Model verification 276
V. Conclusion 277
Chapter 7. General conclusion 281
I. Discussion 281
II. Future works 287
References 291
Appendices 309
Appendix A – DEVS metamodel in OCLinEcore 309
Appendix B – Diagram of the DEVS metamodel 312

List of figures

Figure 1. Formalism Transformation Graph [Vangheluwe 2000]. 42
Figure 2. Graphical representation of a sample DEVS coupled model. 49
Figure 3. Mapping between model hierarchy and processor hierarchy..... 52
Figure 4. Heterogeneous model to be distributed over several simulation services. 65
Figure 5. DEVS components interoperability through simulators communication 65
Figure 6. Local simulation of distant and local models 67
Figure 7. Flattening of a processor hierarchy. 73
Figure 8. Extract from [Favre et al. 2006]: metamodeling hierarchy in various technical spaces. 83
Figure 9. Extract from [MOF specification 1.4 2002]: Four layer modeling in MDA..... 89
Figure 10. Development of rich desktop applications with Visual Studio 2010 WPF software factory. 94
Figure 11. EMP logo representing the various subprojects [EMP logo 2006]. 99
Figure 12. Extract from [Gronback 2009]: Ecore model. 101
Figure 13. GMF – sample graphical definition model[18]. 104
Figure 14. GMF – Sample tooling definition model[18]..... 104
Figure 15. GMF – Sample mapping model[18]. 104
Figure 16. GMF – Sample generated editor[18]. 105
Figure 17. GMF-tooling workflow[18]..... 106
Figure 18. Sample editor generated by Xtext. 107
Figure 19. Samples of n-stages executions. 113
Figure 20. Various use cases for a DEVS model. 159
Figure 21. Overview of SimStudio. 160
Figure 22. DEVS metamodel – Ports. 164
Figure 23. DEVS metamodel – State variables. 165
Figure 24. DEVS metamodel – Atomic models..... 165
Figure 25. DEVS metamodel – Components. 167
Figure 26. DEVS metamodel – Couplings. 168
Figure 27. DEVS metamodel – Tie-breaking function (select). 171
Figure 28. DEVS metamodel – Coupled models..... 172

Figure 29. Megamodel of the generation to DEVS-MS. 178
Figure 30. Sample instantiations of the RandomSwitch model. 182
Figure 31. Sample trajectories of a Processor model. 186
Figure 32. Sample coupled model in Eclipse-DDML. [Ughoroje 2010] 191
Figure 33. Sample atomic model in Eclipse-DDML. [Ughoroje 2010] 192
Figure 34. NetSwitch coupled model. 196
Figure 35. DEVS model editor generated by EMF. 197
Figure 36. XHTML documentation generated from the NetSwitch model. 199
Figure 37. SVG diagram generated from the NetSwitch model. 200
Figure 38. Generic simulation of a specific model. 215
Figure 39. Specific simulation of a specific model. 217
Figure 40. Sample coupled model. 219
Figure 41. Sequence diagram of processing an internal event with a hierarchical simulator. 220
Figure 42. Sequence diagram of processing an internal event with a flattened simulator... 221
Figure 43. Sequence diagram of a simulation step after elimination of all processors..... 223
Figure 44. Sample model: NetSwitch with basic experimental frame. 224
Figure 45. Automatic generation of specialized simulators from model specifications..... 230
Figure 46. Architecture of a pervasive dual cache [D'Orazio and Traoré 2009]. 262
Figure 47. Semantic pervasive dual cache model. 264
Figure 48. Client model. 264
Figure 49. Proxy model. 265
Figure 50. Mean execution time for various implementations of the cache model. 272
Figure 51. Mean compilation time of various implementations of the cache model. 275

List of codes

Code 1. DEVS simulator algorithm. 54
Code 2. DEVS coordinator algorithm. 56
Code 3. DEVS root coordinator algorithm. 57
Code 4. C metaprogram generating a C source code printing the string given as argument. 114
Code 5. Sample use of a fluent interface for robot manipulation. 118
Code 6. LISP Code generating the code computing x^n. 121
Code 7. C++ template metaprogram for computing x^n. 123
Code 8. Binary search in C. 125
Code 9. Binary search residual function for a given size (3). 125
Code 10. Two different instantiations of the same class template. 132
Code 11. Generic `begin` function template. 133
Code 12. Sample use of the generic `begin` function. 134
Code 13. Sample use of Substitution Failure Is Not An Error (SFINAE). 135
Code 14. SFINAE with `enable_if` and `disable_if`. 135
Code 15. Basic definition of a typelist. 136
Code 16. Typelist of signed integer types. 136
Code 17. Null-terminated typelist of signed integer types. 137
Code 18. Simple access to elements of a typelist. 137
Code 19. `identity` metafunction. 137
Code 20. Sample trait class provided by the C++ Standard Library. 138
Code 21. `size` metafunction – recursive case. 138
Code 22. `size` metafunction – base class and usage. 139
Code 23. `append` metafunction – recursive case. 140
Code 24. `append` metafunction – base case. 140
Code 25. Representation of serializable objects through inheritance. 141
Code 26. Serialization with inheritance. 141
Code 27. Invocation with dynamic dispatch. 142
Code 28. Representation of serializable objects through genericity. 142
Code 29. Invocation with static dispatch. 143

Code 30. Boost.Preprocessor data structures. 145
Code 31. Function specialization with `enable_if` and `disable_if`. 147
Code 32. Possible implementation of the `is_same` metafunction. 149
Code 33. Encapsulation of a metafunction into a metafunction class. 149
Code 34. Using placeholders to turn a metafunction into a lambda expression. 149
Code 35. Invocation of a lambda expression with actual arguments. 149
Code 36. Partial application of a metafunction. 150
Code 37. Manipulation of an MPL vector. 151
Code 38. Sample use of MPL algorithms. 152
Code 39. Sample use of a Fusion map. 154
Code 40. Sample use of Fusion lazy algorithms. 155
Code 41. Port names uniqueness invariant in OCLinEcore. 166
Code 42. Sample semi-generic code. 175
Code 43. Sample semi-generic code mapped to C++. 176
Code 44. Sample semi-generic code mapped to C#. 176
Code 45. Fragment of the grammar of our "semi-generic" language. 179
Code 46. Updated algorithm for initialization of simulators. 183
Code 47. Updated algorithm for initialization of coordinators. 184
Code 48. Sample textual format for representing trajectories. 186
Code 49. Fragment of the XMI file of the NetSwitch model. 198
Code 50. CD++ – Switch class definition. 202
Code 51. CD++ – Switch constructor definition. 202
Code 52. CD++ – NetSwitch specification. 203
Code 53. PyDEVS – State class for the Switch model. 204
Code 54. PyDEVS – Switch constructor. 204
Code 55. PyDEVS – NetSwitch definition. 205
Code 56. DEVS-MS – Beginning of the Switch class definition. 206
Code 57. DEVS-MS – NetSwitch definition. 207
Code 58. Class specialized for the `s` component. 224
Code 59. Class specialized for the `p1` component. 225
Code 60. Class specialized for the `trans` component. 226
Code 61. Root coordinator specialized for the NetSwitchExperiment model. 228

Code 62. Message class (pseudo-code). 234
Code 63. Using types as identifiers. 235
Code 64. Combining functions and metafunctions. 236
Code 65. Model class (pseudo-code). 237
Code 66. AtomicModel class (pseudo-code). 238
Code 67. CoupledModel class (pseudo-code). 241
Code 68. Possible representation of the NetSwith couplings in DEVS-MS. 243
Code 69. Sample use of the `DECLARE_PORT` macro. 244
Code 70. Conversion from compile-time identifier to runtime string. 244
Code 71. Sample use of the `PORTS` macro. 244
Code 72. Sample use of the `COMPONENTS` macro. 245
Code 73. Sample use of the `COUPLINGS` macro. 245
Code 74. Sample use of the `SELECT` macro. 245
Code 75. Simulator class (pseudo-code). 246
Code 76. Simulator – Initialization algorithm (pseudo-code). 247
Code 77. Simulator – Internal transition algorithm (pseudo-code). 247
Code 78. Simulator – Callback function for generating outputs (pseudo-code). 248
Code 79. Simulator – Modification of the internal transition algorithm (pseudo-code). 248
Code 80. Sample lambda function using the callback mechanism (pseudo-code). 249
Code 81. Simulator – External transition algorithm (pseudo-code). 249
Code 82. Coordinator class (pseudo-code). 250
Code 83. Coordinator – Constructor (pseudo-code). 252
Code 84. Coordinator – Unrolled constructor for a sample coupled model (pseudo-code). 252
Code 85. Coordinator – Initialization algorithm (pseudo-code). 253
Code 86. Coordinator – Internal transition algorithm (pseudo-code). 254
Code 87. Coordinator – Recursive algorithm to construct a static list of imminent components depending on dynamic computations (pseudo-code). 255
Code 88. Coordinator – Combinations generated by the metaprogram and selected by the object program (pseudo-code). 256
Code 89. Coordinator – Algorithm for handling input and output messages (pseudo-code). 258
Code 90. Root class (pseudo-code). 260

Code 91. Root – Main simulation loop (pseudo-code). 261
Code 92. Root – Default handling of "leaking" events (pseudo-code). 261
Code 93. RandomSwitch atomic model in DEVS-MS. 267
Code 94. Definition of the output function and external transition function using the simplified syntax. 268
Code 95. Proxy coupled model in DEVS-MS. 270

List of tables

Table 1. Comparison of metaprogramming approaches. 129

Table 2. Classification of model information in static and dynamic data. 233

Table 3. Comparison of the execution time of various implementations of the cache model. 273

Table 4. Mean compilation time of various implementations of the cache model. 275

List of abbreviations

ACIMS	Arizona Center for Integrative Modeling & Simulation
AMW	ATLAS Model Weaver
AOP	Aspect-Oriented Programming
API	Application Programming Interface
ASL	Action Specification Language
AST	Abstract Syntax Tree
ATL	ATLAS Transformation Language
AToM3	A Tool for Multi-formalism and Meta-Modeling
BPEL	Business Process Execution Language
C++ TMP	C++ Template MetaProgramming
CASE	Computer-Aided Software Engineering
CDEVS	Classic DEVS
CIM	Computation-Independent Model
CIRAD	Centre de Coopération Internationale en Recherche Agronomique pour le Développement
COFF	Common Object File Format
CORBA	Common Object Request Broker Architecture
CSS	Cascading Style Sheets
CSV	Comma-Separated Values
DDML	DEVS-Driven Modeling Language
DESS	Differential Equation System Specification
DEVS	Discrete Event System Specification
DEVS-MS	DEVS-MetaSimulator
DOM	Document Object Model
DS-DEVS	Dynamic Structure DEVS

DSL	Domain-Specific Language
DTSS	Discrete-Time System Specification
EAI	Enterprise Application Integration
EBNF	Extended Backus-Naur Form
EDSL	Embedded Domain-Specific Language
EIC	External Input Coupling
ELF	Executable and Linkable Format
EMF	Eclipse Modeling Framework
EMP	Eclipse Modeling Project
EOC	External Output Coupling
EOV	Eyes of VLE
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
GPPL	General-Purpose Programming Language
GPSS	Global Purpose Simulation System
GVLE	GUI for VLE
HLA	High-Level Architecture
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IC	Internal Coupling
IDE	Integrated Development Environment
IDL	Interface Description Language
INRA	Institut National de Recherche Agronomique
Java EE	Java Enterprise Edition
Java RMI	Java Remote Method Invocation
JET	Java Emitter Templates

KM3	Kernel Meta Meta Model
LISIC	Laboratoire d'Informatique Signal et Image de la Côte d'Opale
M&S	Modeling & Simulation
M2M	Model-to-Model
M2T	Model-to-Text
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
MIMOSA	Méthodes Informatiques de MOdélisation et Simulation Agents
MOF	MetaObject Facility
MPL	MetaProgramming Library
MPS	MetaProgramming System
MS	Mobile Station
MSDL	Modelling, Simulation and Design Lab
MSPL	Multi-Stage Programming Language
MSS	Mobile Support Station
MTL	Model to Text Language
OAL	Object Action Language
OCL	Object Constraint Language
OMG	Object Management Group
PDEVS	Parallel DEVS
PHP	PHP: Hypertext Preprocessor
PIM	Platform-Independent Model
PSM	Platform-Specific Model
QNAP2	Queuing Network Analysis Package II
QVT	Query/View/Transformation
QVTO	Operational QVT

REST	REpresentation State Transfer
SAX	Simple API for XML
SFINAE	Substitution Failure Is Not An Error
SI	International System of Units
SIMAN	SIMulation Analysis
SLAM	Simulation Language for Alternative Modeling
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSJ	Stochastic Simulation in Java
SVG	Scalable Vector Graphics
TCS	Textual Concrete Syntax
TMF	Textual Modeling Framework
UI	User Interface
UML	Unified Modeling Language
URI	Uniform Resource Identifier
VFL	VLE Foundation Libraries
VHDL	VHSIC Hardware Description Language
VLE	Virtual Laboratory Environment
VV&T	Verification, Validation and Testing
WADL	Web Application Description Language
WPF	Windows Presentation Foundation
WS	Web Service
WSCI	Web Service Choreography Interface
WSDL	Web Service Description Language
XAML	eXtensible Application Markup Language

XHTML	eXtensible HyperText Markup Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
XSLT	eXtensible Stylesheet Language Transformations

Chapter 1. General introduction

I. Context

During the last decades, Modeling & Simulation (M&S) has permeated most domains of engineering and science in general, both in academy and in industry. The capacity of M&S to increase our understanding of systems, to evaluate them or to predict their evolution has proved essential in a variety of fields.

However, the need to model and simulate systems that are more and more complex raises a number of challenges, and many scientists agree that the domain of M&S is at a turning point where it must evolve to overcome these new issues, so as to cross the actual frontiers of complexity [GSAT 2005] [PITAC 2005]. More precisely, the US National Science Foundation identified six core challenges faced by the M&S community [NSF 2006]:

1. Multiscale M&S.
2. Verification, Validation and Uncertainty Quantification.
3. Real-time interaction between simulations and experiments.
4. Development of software tools, paradigms and protocols for interdisciplinary M&S.
5. Visualization and data management.
6. Algorithms and computational performance.

The DEVS formalism, by establishing sound mathematical foundations and a clear separation between models and simulators, provides a good basis for tackling these challenges. Notably, it allows practitioners to focus on modeling without worrying about the simulation aspect. The latter can be handled by generic software environments, capable of simulating any DEVS models. Thereby, practitioners can focus on developing accurate models for their systems under study, while toolsmiths can focus on providing new tools, new features or new simulation algorithms. As a result, many environments for DEVS M&S have been developed during the past decades [Bolduc and Vangheluwe 2002] [Himmelspach and Uhrmacher 2007] [Quesnel 2006] [Wainer 2009] [Zeigler and Sarjoughian 2003], giving practitioners a broad set of alternatives to choose from.

However, this profusion of DEVS tools has an unwanted consequence: the collaboration between scientists is hindered by the lack of interoperability between these tools. Due to the absence of a DEVS standard, DEVS models are expressed either in natural language or in formats that are tightly tied to a particular environment. Because of this, exchanging, reusing and comparing models between teams is a costly and error-prone process, hampering the sharing of knowledge essential to scientific advances.

II. Objectives

Our objective in this thesis is to show how advanced software engineering techniques can be used to tackle some of the previously mentioned challenges. More precisely, we will focus on the following aspects: integration of heterogeneous DEVS tools, abstraction of implementation details, model verification and performance.

Our first goal is to create an environment for DEVS M&S that provides high-level abstractions both for the toolsmith (developer of the environment) and for the practitioner (modeler). This environment should allow users to design DEVS models in a high-level language, independent of a particular programming language or platform. Then, it should be able to process such specifications to automate as much tasks as possible, such as writing documentation or verifying model properties.

More importantly, the tool should be able to automatically generate platform-specific implementations of models for various DEVS simulators. This way, the practitioner would

stay away from low-level details, reducing the risk of introducing errors during implementation: he would focus on modeling and not on programming. Moreover, such automatic generation would also greatly facilitate the migration of models from one environment to another, making it easier to experiment with different tools (different algorithms, different features) and hardware (desktop computer, computer cluster, computer grid, etc.). The ability to easily port models would also increase their reusability, notably between teams.

From the toolsmith perspective, the environment should be easily extensible, allowing new features to be added without too much difficulty. For instance, it should be possible to add model analysis features, or visualization, without going through the trouble of modifying a complex software program. Similarly, adding new target simulators for the generation feature should be as simple as possible, so that existing and future tools can be integrated easily. We think that such seamless extensibility is essential for the development of innovative and powerful M&S facilities.

Our second goal is to develop a generic DEVS simulation library providing rigorous model verification features, as well as improved performance compared to existing simulators. More precisely, the library should be developed in such way that it prevents the practitioner from creating models that do not meet the requirements of the DEVS formalism. Design errors should be caught by the library as soon as possible in the development cycle so that it can be quickly corrected.

Regarding performance, we want the library to eliminate as much simulation overhead as possible. Due to their genericity, DEVS simulators usually need to perform several time-consuming operations that are not really relevant to the simulated model, such as handling events, exchanging messages between entities, etc. If a practitioner were to develop a simulation software especially dedicated to his model, such additional computations could be removed, increasing the overall performance of the simulation, but at the expense of the development time. To solve this issue, we aim at developing a library that is generic, i.e. capable of simulating any DEVS models, but which generates a simulation executable that is

as close as possible to the one an experienced programmer would have crafted specifically for the model at hand.

III. Outline

This thesis is structured into five main chapters. Chapter 2 provides an overview of the field of DEVS M&S. The first sections of this chapter focus on the DEVS formalism itself, starting with a brief presentation of its foundations before describing in detail model specifications and simulation algorithms. The next sections are dedicated to DEVS tools: a presentation of some of the most prominent DEVS environments is given, followed by an analysis of some limits and potentially improvable aspects we identified in these tools.

Chapter 3 and Chapter 4 thoroughly describe the two software engineering techniques we used in this work. Chapter 3 is dedicated to Model-Driven Engineering (MDE). It begins with some background theory, i.e. explanations of the most important concepts and definitions needed to apprehend MDE. Then, it provides an overview of the various implementations of MDE currently available, notably the Model-Driven Architecture, the Eclipse Modeling Project and Microsoft's Software Factories. Finally, it presents in more detail the MDE framework we actually used for our works, namely the Eclipse Modeling Project.

The structure of Chapter 4, which focuses on metaprogramming, is quite similar. Firstly, the main concepts of metaprogramming are explicated. Secondly, multiple metaprogramming techniques are explained and analyzed, along with a comparison of the pros and cons of these various approaches. Lastly, the technique we retained, namely C++ Template MetaProgramming, is described in depth.

The last two chapters present the main contributions of this thesis. In Chapter 5, we introduce SimStudio, our Model-Driven DEVS environment. After presenting the overall architecture of the environment, we detail the platform-independent metamodel we devised for representing DEVS models. Then, we expose the various generation features we included, along with a short description of the model transformations involved. In a second part, we demonstrate the capabilities of SimStudio through a sample study case.

Chapter 6 deals with DEVS-MetaSimulator, our DEVS simulation library relying on C++ Template MetaProgramming. To explain the motivation for this library and give a good understanding of the underlying ideas, we start this chapter with some comparisons between usual simulators and what we strive for. We then proceed to describe the implementation of the library in C++ (without too much technical details). Finally, we present the results obtained when testing our library with a sample model.

To conclude this thesis, we discuss the main contributions exposed in this document, along with some trails for future works.

Chapter 2. Modeling and Simulation with Discrete Event System Specifications

I. Introduction

Discrete Event System Specification (DEVS) is a framework for Modeling & Simulation (M&S) that aims at providing a unified basis for a wide range of simulations. Since its creation, it has been leveraged by many scientific teams to model and simulate all sorts of systems such as industries [Ninios et al. 1995], environmental systems [Wainer 2006] [Filippi et al. 2010], embedded systems [Wainer and Castro 2011], intelligent machines [Zeigler and Louri 1993], diseases [Jammalamadaka et al. 2007] [Shang and Wainer 2005] or human behavior [Seck et al. 2004].

To support DEVS M&S, many environments have been developed to facilitate model design, perform simulation, visualize results, etc. These tools provide different features, and use various languages, formats and algorithms. However, most of them adopt the same approach for simulating DEVS models: they provide a single generic simulator for handling all

models. As we will see, this approach has consequences on various aspects, especially performance and verification.

More importantly, the profusion of DEVS environments greatly hinders the collaboration between scientists: sharing models with a colleague using another tool implies investing a great deal of effort to port the model. As a consequence, the DEVS community works on establishing a set of standards for DEVS M&S, to permit a better interoperability and compatibility between these heterogeneous tools.

In this chapter, we will first of all present DEVS, starting by the foundational theory before explaining the formalism itself along with the corresponding simulation algorithms. Then, we will provide an overview of the main DEVS environments currently available. Finally, we will explain the limitations currently present in DEVS software that we will tackle in this thesis.

II. DEVS formalism and tools

DEVS is a powerful formalism for modeling and simulation of dynamic systems. Since its creation in the seventies [Zeigler 1976], it has been used by many scientists in a wide range of domains, and has gathered a productive community around it. This community has not only studied in depth the original formalism, now named Classic DEVS, but has also developed extensions and variants to tackle different issues.

We will start this section by providing a short overview of DEVS background and strengths. After that, we will explain Classic DEVS in detail, from the definition of DEVS models to their interpretation through simulation. We will then briefly introduce a few DEVS variants before presenting the main DEVS tools currently available.

II.1. Overview

II.1.1. Background

In the late seventies, Bernard P. Zeigler proposed a theory of modeling and simulation, stemming from systems theory, which aimed at providing a rigorous and unified framework for the modeling and simulation of all systems.

To reach this goal, the theory builds up a hierarchy of system specifications, starting at a very low level of abstraction, and progressively adding more and more ways of expressing knowledge about models. The outcome of this incremental construction is three basic formalisms for modeling systems according to three different paradigms: continuous, discrete-time and discrete-event.

II.1.1.1. Hierarchy of specifications

The theory contains a set of specifications corresponding to different levels of knowledge about the system. Each new specification is built on top of the previous one, allowing more and more knowledge to be injected into the model. A feature of the theory is that each specification is proven to be convertible to the lower one, meaning that it is always possible, given a structured description of a system, to obtain a lower-level description. The most useful conversion is from the higher levels, well-suited for human manipulation, to one of the lowest, i.e. the input/output trajectories, usually obtained through simulation.

The lowest level of specification simply describes the system as a black box with an interface for receiving inputs and sending outputs. The aim of this specification, called the I/O Observation Frame, is to define precisely which variables must be taken into account when modeling the system: what are the influences to consider, and what is the data to observe.

This purely structural description is rather limited; the next level builds upon it to provide a way of recording the behavior of the system, as a set of input/output associations. The I/O Relation Observation is akin to a log of the system, where outputs of the system are associated with inputs. As the system is still seen as a black box, one input can correspond to several outputs, since the state of the system is not taken into account yet.

To remove this ambiguity, the next specification introduces the notion of initial state. Instead of using a single relation between inputs and outputs, the I/O Function Observation partitions the behavior of the system in a set of functions, each one providing an unambiguous output for an acceptable input. These functions represent the different states of the system: given an initial state, the system exhibits a functional behavior and generates a unique response to a given stimulus.

The state can be more precisely defined in the next level of specification, the I/O System, which accounts for the intrinsic functioning of the system. This is achieved by adding to the specification a state set as well as a transition function to describe how the system evolves from state to state. Outputs are no longer function of inputs; the input only impacts the state, and the state determines the output. We will see later that atomic DEVS models are largely based on this kind of specification.

Finally, the specification at the higher level of knowledge aims at describing the system as a collection of interacting constituents. The theory proposes two kinds of specifications for modeling the decomposition of a system into smaller components. The first one focuses on multicomponent systems, which are collections of highly coupled components. In this kind of system, each component can directly influence the others: the evolution of a component's state depends not only on the system's input and on its own state, but also on the states of the other components. These high dependencies can make such a specification hard to develop and comprehend. In addition, the components of such models can be hard to reuse in other specifications. This issue can be solved by specifying a system as a network of systems specifications (or coupled system): instead of dealing with highly interdependent components, a coupled system embeds modular components, which are encapsulated models with a well-defined interface. The interactions between components are defined as connections between their output and input interfaces, meaning that a component does not need any knowledge about its "influencees" and "influencers": it only receives inputs and generates outputs on its interface, without being aware of its environment. This decoupling can make the system much easier to understand, and eases the development of new models by enabling the reuse of previously written specifications. It is quite logically that this type of modular specification was retained for coupled DEVS model, as we will see in a few pages.

This hierarchy of specifications shows how a system can be modeled at different levels, depending on the amount of knowledge possessed about this system. Each layer increases the abstraction level, going from raw data to structured and modular specifications. At the highest levels, a system is described by the means of a modeling formalism, a language for representing systems in a concise way. There are many modeling formalisms, which can be divided in three main categories: continuous, discrete-time and discrete-event. Hereafter,

we will shortly present these categories, along with a description of a basic formalism for each one.

II.1.1.2. Fundamental formalisms

A first way to model a system is through the continuous paradigm, using a Differential Equation System Specification (DESS). In this vision of the world, the system is seen as a set of variables evolving continuously through time. The state of the system is modeled with differential equations representing the rate of change of the state values, function of the inputs. The outputs are continuous quantities depending on the state of the system. In this kind of model, the state of the system, its input and its output are all defined over a continuous time base. Such continuous models are well suited to analysis, but are hard to simulate on digital devices such as computers. Usually, the continuous values are approximated using some kind of iterative method. However, this discretization can also be done at the modeling level.

A Discrete-Time System Specification (DTSS) is quite similar to a DESS, with the major difference that the time base used is no longer continuous but discrete. Intuitively, it means that we consider the system's evolution as a sequence of snapshots, taken at fixed intervals of time. The model of a system, in the discrete-time paradigm, defines how the system "jumps" from one state to another at each step, taking into account its input. Simulating such a model is pretty straightforward: it boils down to advancing a virtual clock at a fixed pace, while invoking a transition function on the model at each step of the simulation. However, this kind of simulation can be quite inefficient when the simulated system tends to remain in stable states for a certain amount of time. Similarly, the modeling of such systems using discrete-time specification can sometimes be a bit awkward. The discrete-event paradigm provides a different approach to modeling, which can be more natural and usually more efficient in terms of computing resources needed for simulation.

The main idea behind discrete-event modeling is to focus on the significant changes of the system's state. Many systems stay in a given state for some amount of time, before evolving into a new state either as a result of an autonomous/intrinsic activity, or because some external stimulus triggered a change. The discrete-event world view concentrates on these state-changing events, and is formalized by the Discrete-EVent system Specification (DEVS),

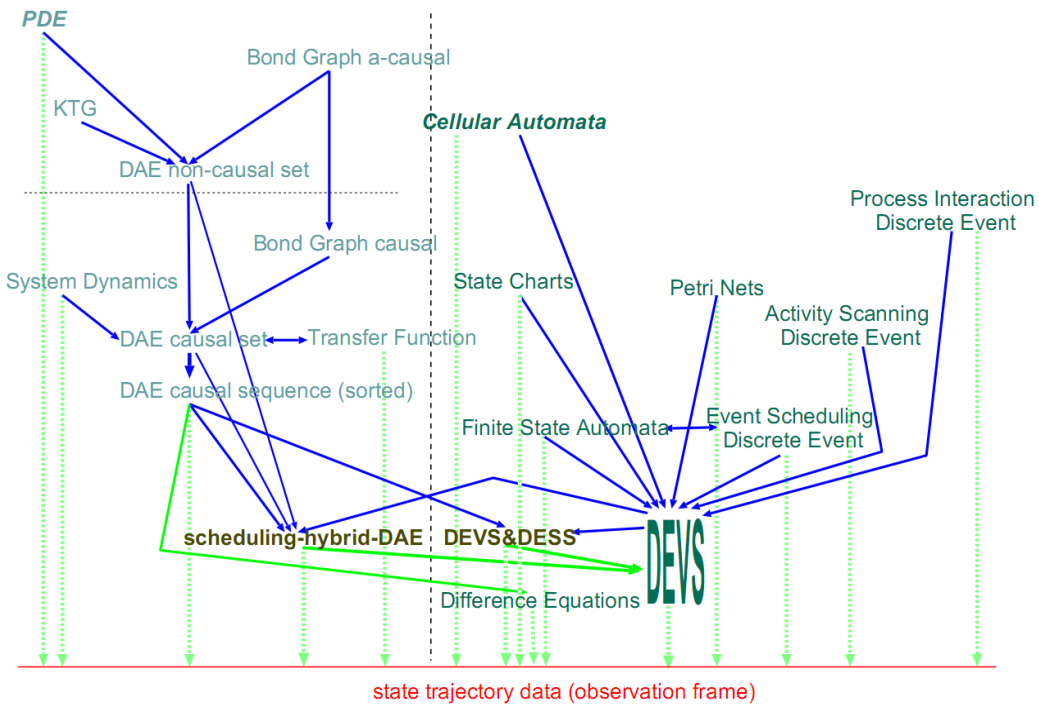


Figure 1. Formalism Transformation Graph [Vangheluwe 2000].

the formalism at the core of this thesis, which we will present in depth in the following pages.

II.1.2. DEVS as an intermediate language for simulating models

The strength of DEVS lies in its universality. Bernard Zeigler describes it as a "computational basis for implementing behaviors that are expressed in the other basic systems formalisms" [Zeigler et al. 2000]. Indeed, DEVS subsumes DTSS and can efficiently approximate DESS: discrete-time models are just a specific case of discrete-event models where events occur at fixed intervals; continuous models such as differential equations are often approximated using iterative integration methods (discrete-time), so they can also be represented by discrete-event models. However, a better alternative to represent DESS as DEVS is quantization [Zeigler 1998], where the continuous inputs and outputs of the continuous model are mapped to a discrete set of values, and events are triggered when a change of value occurs (i.e., the input or output of the continuous model crosses a threshold).

In [Vangheluwe 2000], Hans Vangheluwe provides a formalism transformation graph, reproduced in Figure 1, which shows the possible transformations between several well-known formalisms. The left part of the graph contains continuous models, while the right

part contains discrete models. The important thing to notice is that all formalisms can eventually be reduced to DEVS, making it a perfect candidate for a unifying computation model for simulation.

II.1. Classic and Parallel DEVS

DEVS comes in several flavors: over the years, the M&S community developed many variants and extensions to the original formalism, now named Classic DEVS (CDEVS). The most notable evolution is Parallel DEVS (PDEVS), which is widely used by the community. In this section, we give an in-depth description of CDEVS with its limits, and a small overview of PDEVS, its direct successor improving simultaneous events handling.

CDEVS defines two kinds of models for representing a system: atomic and coupled models.

II.1.1. Classic DEVS atomic model

An *atomic model* is an entity holding a state, which evolves autonomously (in response to internal events) and in response to extrinsic stimuli (external events). More precisely, an atomic model is composed of several state variables that describe the state of the system at some point in time. The model remains in a state for a certain amount of time, determined by the modeler, before undergoing an internal transition to a new state. In addition to modifying the state of model, an internal transition generates some outputs that will eventually become inputs for other models (external events). When a model receives an input, an external transition is triggered and the model goes into a new state.

Formally, an atomic DEVS model is defined by a tuple

$$M = \langle X, Y, S, ta, \delta_{int}, \delta_{ext}, \lambda \rangle$$

where

- *X* is the set of inputs, meaning the set of values that can be carried by external events impacting the model.
- *Y* is the set of outputs, meaning the set of values that can be carried by events generated by the model.
- *S* is the state set, containing all the possible characterizations of the system.

- ta is the time advance function: $ta(S) \rightarrow [0, +\infty]$. It determines the amount of time the model should stay in a given state before undergoing an internal transition, if no external event occurs.
- δ_{int} is the internal state transition function: $\delta_{\mathrm{int}}(\mathrm{S}) \rightarrow \mathrm{S}$. It defines what state the model should go in when it undergoes an internal transition, depending on the state it is currently in.
- δ_{ext} is the external state transition function: $\delta_{\mathrm{ext}}(\mathrm{Q} \times \mathrm{X}) \rightarrow \mathrm{S}$. Q is the set of total states; the total state of the model contains both the state and the time elapsed since the last transition (internal or external): $Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$. Depending on this total state and on the input received, δ_{ext} determines the new state of the model.
- λ is the output function: $\lambda(\mathrm{S}) \rightarrow \mathrm{Y}$. Invoked before each internal transition, it provides the value generated by the model, functions of the state it is in.

To be more easily manipulable, S is often defined with a multivariable/structured set [Zeigler et al. 2000]: it is described as the Cartesian product of several sets denoted by variables, providing additional semantics to the model. Therefore, S is described with several state variables representing different constituents of the model's state:

$$S = \{(v_1, v_2, \dots, v_n) | v_1 \in S_1, v_2 \in S_2, \dots, v_n \in S_n\}$$

To simplify model definitions, we will often use an abuse of notation and describe state sets as follows: $S = v_1 \in S_1 \times v_2 \in S_2 \times \dots \times v_n \in S_n$. For instance, a state composed of one natural number "n" and one real number "r" should be defined as $S = \{(n, r) | n \in \mathbb{N}, r \in \mathbb{R}\}$, but we will write it $S = n \in \mathbb{N} \times r \in \mathbb{R}$.

A similar approach is used for inputs and outputs: instead of representing the interface of a model with a single set containing every single potential input value (resp. output), we can decompose the input (resp. output) set into several sets denoted by a name, corresponding to model ports. This greatly facilitates modeling and provides another way of injecting additional knowledge into the model. Formally, the input set becomes a set of pairs containing a port name and a value:

$$X = \{(p, v) | p \in InPorts, v \in X_p\}$$

The output set can be defined in a similar way. Once again, to facilitate notation in the rest of this document, we will define input and output sets in the following way: $X = port_1 \in X_{port_1} \cup port_2 \in X_{port_2} \cup \ldots \cup port_n \in X_{port_n}$. For instance, an input set decomposed in two ports "n" and "r", where "n" accepts natural numbers and "r" real numbers, should be defined as $X = \{(p, v) | p \in InPorts, v \in X_p\}$ $where\ InPorts = \{n, r\}, X_n = \mathbb{N}\ and\ X_r = \mathbb{R}$; our notation allows this definition to be shorten into $X = n \in \mathbb{N} \cup r \in \mathbb{R}$.

Sample atomic DEVS model

To illustrate the definition of atomic DEVS models, we will present a simple model that will be used in later chapters: a random switch. The aim of this model is to randomly dispatch the values it receives to one of its two output ports. Since the choice of the output port is akin to a random experiment with two possible outcomes, we will name the outputs ports "success" and "failure". In order to be quite generic, the model is parameterized by the type of values it handles and the probability of success; to choose the output port on which to send the value, the model draws a random number uniformly between 0 and 1 and compares it with the success probability. Here is the complete definition:

$$RandomSwitch_{valueType,successProbability} = \langle X, Y, S, ta, \delta_{int}, \delta_{ext}, \lambda \rangle$$

where

- $\mathrm{X} = \mathrm{input} \in \mathrm{valueType}$
- $\mathrm{Y} =$

  $\mathrm{success} \in \mathrm{valueType} \cup$

  $\mathrm{failure} \in \mathrm{valueType}$
- $\mathrm{S} =$

  $\mathrm{currentValue} \in \mathrm{valueType} \times$

  $\mathrm{state} \in \{waiting, outputting\} \times$

  $\mathrm{isSuccess} \in \{true, false\}$
- $\mathrm{ta}(\mathrm{currentValue}, \mathrm{state}, \mathrm{isSuccess})$

  $= 0 \quad \text{if state} = outputting$
  $= \infty \quad \text{if state} = waiting$
- $\delta_{\mathrm{int}}(\mathrm{currentValue}, \mathrm{state}, \mathrm{isSuccess})$

  $= (\mathrm{currentValue}, waiting, \mathrm{isSuccess}) \quad \text{if state} = outputting$
  $\text{is not defined} \quad \text{otherwise}$

- $\delta_{\text{ext}}(\text{currentValue}, \text{state}, \text{isSuccess}, \text{e}, (\text{port}, \text{value}))$
  $= (\text{value}, outputting, \text{rand}(0,1) < successProbability)$
- $\lambda(\text{currentValue}, \text{state}, \text{isSuccess})$
  $= (\text{success}, \text{currentValue}) \quad \text{if isSuccess}$
  $= (\text{failure}, \text{currentValue}) \quad \text{otherwise}$

This definition states that the random switch model has one input port named *input* and two output ports named *success* and *failure*. The values received and sent on these ports belong to the set *valueType* given as a parameter of the model. The state is composed of three variables: *currentValue*, the value to dispatch; *state*, which is either *waiting* if the model is waiting for an input or *outputting* if it is in the process of dispatching a received value; and *isSuccess*, a boolean value that determines on which port the current value is to be sent. The model stays in the state *waiting* until some external event impacts it. When such an event occurs, the model stores the corresponding input in *currentValue*, determines on which port to send the value, and goes into state *outputting*. The duration of this state is 0 in order to immediately trigger the internal event needed for generating the output. The internal transition simply puts back the model into waiting mode, and the value stored is sent on one of the two output ports, depending on the value of *isSuccess*.

Atomic models are not sufficient to easily represent systems. DEVS provides a way to combine them into coupled models, which embed several models (components) interacting through their ports.

II.1.2. Classic DEVS coupled model

In DEVS, models can be combined together to construct more complex models. This is achieved by defining coupled models, which provide an interface similar to atomic models but whose behavior is obtained by connecting several sub-models through their output and input ports. Events received by the coupled model or generated by its components are forwarded to other components, becoming inputs to the sub-models. The components' outputs can also be connected to the output interface of the coupled model. Thus, a coupled model can be seen as a graph of components (sub-models) exchanging events through their ports. An important point to note is that from the outside, coupled models and atomic models are similar: they all expose an interface made of input and output ports. Thanks to

that, a coupled model can embed both atomic and coupled models, thus enabling the construction of hierarchical, nested models with arbitrary depths.

Formally, a coupled DEVS model is defined by a tuple

$$N = \langle X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC, Select \rangle$$

where

- *X* is the input set, defined using ports as we saw for atomic models.
- *Y* is the output set, defined using ports.
- *D* is a set containing the names of the components embedded by the coupled model.
- $\{M_d\}$ is the set of components.
- *EIC* stands for External Input Coupling. It describes how the input ports of the coupled model are connected to the input ports of some of its components.
- *EOC* stands for External Output Coupling. It describes the coupling between the output ports of components and those of the coupled model.
- *IC* stands for Internal Coupling. It defines the connections between components.
- *Select* is the tie-breaking function. In Classic DEVS, simultaneous events are handled by serializing them in the simulation: even though they occur at the same simulation time, they are treated in several simulation steps. To do so, the modeler must prioritize the components so that only one gets activated at each simulation step. Given a set of components about to undergo an internal event (abusively called imminent components), this function indicates which one should be activated, i.e. generate its output and change its state according to its internal transition function:

  $Select$: $\mathcal{P}(D) - \emptyset - \{\{d\} | d \in D\} \rightarrow D$ ($\mathcal{P}$(D), also noted 2^D, is the power set of D, the set of all its subsets.)[1]

[1] The tie-breaking function is only needed when two or more components have the same time of next event. Consequently, *Select* does not need to be defined for the entire power set of D, only for subsets whose cardinality is greater than two. This is why we remove the empty set and the set of singletons from the domain of *Select*.

All three types of couplings are defined as sets of connections, a connection being a pair source-destination. Source and destination are denoted by a component name and a port name. For instance, the formal definition of IC is

$$IC \subseteq \left\{ \left((srcComp, srcPort), (destComp, destPort) \right) \middle| \begin{array}{l} srcComp, destComp \in D, \\ srcPort \in OutPorts_{srcComp}, \\ destPort \in InPorts_{destComp} \end{array} \right\}$$

In the case of EIC, the source component is necessarily the coupled model itself, hence it is often omitted. Similarly, the destination component in EOC is always the coupled model, and can be left out the specification.

Sample coupled DEVS model

Let us illustrate the definition of coupled DEVS models by assuming that we are developing an elementary model of the human perception, which tries to model the motor response to some sensory inputs. For the sake of simplicity, we will only consider two senses, eyesight and hearing. Imagine that the following models already exist:

- a model of the brain, which receives nervous signals from the sense organs and processes them, producing in response a motor signal intended for the muscular system;
- a model of the eye, receiving light and converting it into nerve impulses;
- a model of the ear, converting sound to nerve impulses.

The intrinsics of these models are not relevant: they could be simple atomic models, or complex coupled models with many submodels. The thing of interest is how we are going to put them together to form a larger model, by the means of a coupled model.

A coupled DEVS model can easily be represented graphically, using boxes to denote components and lines or arrows to represent couplings. Figure 2 shows the perception model we want to define, composed of two instances of the eye model, two instances of the ear model, and one instance of the brain model. The perception model has input ports for receiving information from the environment, in our case light and sound, and an output port to provide the brain response to this information. Before reaching the brain, the information is processed by the sense organs, which translate it into nerve impulses. The response

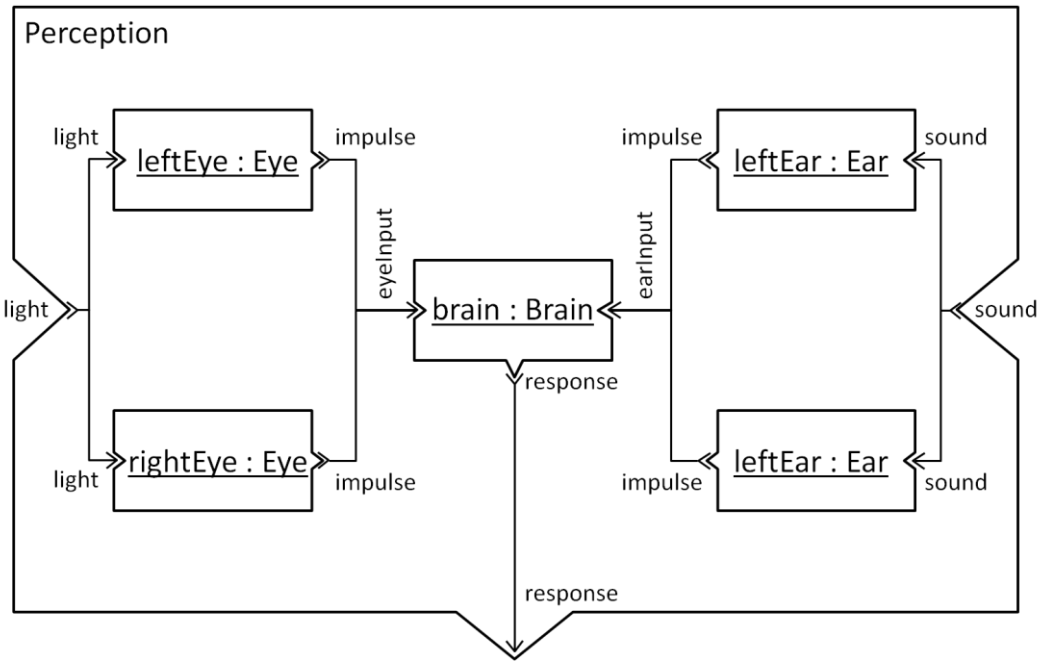


Figure 2. Graphical representation of a sample DEVS coupled model.

generated by the brain is forwarded to the outside of the coupled model for further processing by some other model.

Formally, this model would be described as follows:

$$Perception = \langle X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC, Select \rangle$$

where

- $\mathrm{X} =$

  $\mathrm{ligth} \in \mathrm{T(some\ set\ for\ representing\ light)} \cup$

  $\mathrm{sound} \in \mathrm{U(some\ set\ for\ representing\ sound)}$
- $\mathrm{Y} = \mathrm{response} \in \mathrm{V(some\ set\ for\ representing\ the\ brain\ response)}$
- $\mathrm{D} = \{\mathrm{leftEye, rightEye, leftEar, rightEar, brain}\}$
- $\mathrm{M_{leftEye}} = \mathrm{M_{rightEye}} = \mathrm{Eye}$

  $\mathrm{M_{leftEar}} = \mathrm{M_{rightEar}} = \mathrm{Ear}$

  $\mathrm{M_{brain}} = \mathrm{Brain}$
- $\mathrm{EIC} = \left\{ \begin{matrix} (\mathrm{light}, (\mathrm{leftEye, light})), (\mathrm{light}, (\mathrm{rightEye, light})), \\ (\mathrm{sound}, (\mathrm{leftEar, sound})), (\mathrm{sound}, (\mathrm{rightEar, sound})) \end{matrix} \right\}$
- $\mathrm{EOC} = \{((\mathrm{brain, response}), \mathrm{response})\}$

- $\mathrm{IC} = \left\{ \begin{matrix} \big((\mathrm{leftEye}, \mathrm{impulse}), (\mathrm{brain}, \mathrm{eyeInput})\big), \\ \big((\mathrm{rightEye}, \mathrm{impulse}), (\mathrm{brain}, \mathrm{eyeInput})\big), \\ \big((\mathrm{leftEar}, \mathrm{impulse}), (\mathrm{brain}, \mathrm{earInput})\big), \\ \big((\mathrm{rightEar}, \mathrm{impulse}), (\mathrm{brain}, \mathrm{earInput})\big) \end{matrix} \right\}$
- $\mathrm{Select}(\mathrm{Imminents}) = \text{the first of these models belonging to Imminents:}$

  $brain > leftEye > rightEye > leftEar > rightEar$

The only thing this specification adds to the graphical representation, apart from the declaration of input and output sets, is the definition of the select function. Here, we simply gave a priority to each component: the brain has the priority over the eyes, which have priority over the ears. For instance, if at some point in the simulation both the brain and the right eye should undergo an internal transition, the simulator will activate the brain component and not the eye. Depending on the events triggered by the brain and their impact on other components, the right eye will be activated at the next step of simulation, later, or not at all. This not-so-intuitive behavior is not adapted to a consistent modeling of many real life systems. This point motivated the development of Parallel DEVS, shortly described hereafter.

II.1.3. Parallel DEVS

Parallel DEVS is a DEVS variant which modifies the way simultaneous events are handled. We saw that Classic DEVS's approach is to serialize simultaneous events so that only one model undergoes an internal transition at each simulation cycle. The serialization is performed according to a tie-breaking function that selects the prior component among a set of imminent components. For some systems, this modeling approach is not very intuitive; in these cases, PDEVS provides an alternative that can be more appropriate.

PDEVS is very close to CDEVS, and only differs in a few aspects. The first one is that the inputs and outputs of models are now bags (i.e. unordered collections) of values instead of single elements, meaning that a model can receive/generate several events at the same time. For instance, a model could receive two simultaneous events on one of its port, or generate events on two distinct ports. This impacts the specification of atomic models, where the external transition function now determines the next state functions of the total state and a bag of inputs, and the output function now returns a bag of outputs.

The other modification of PDEVS is the addition of a confluent transition function to atomic models. This function specifies how to handle the simultaneity of an internal event and an external event. Such collisions cannot happen in CDEVS, where only one model undergo an internal transition during a simulation step, but can occur (possibly quite often) in PDEVS. Consequently, the modeler must specify how to handle such conflicts. This is achieved through the confluent transition function, which provides the next state functions of the inputs and the current state (note that there is no need for the elapsed time since the last event, since it is necessarily ta(s)).

Since PDEVS allows simultaneous events, the select function of coupled models is no longer needed. When several components attain their time of next event at the same simulation time, they all generate their outputs, and perform their transition, according to either the internal or confluent transition function, depending on whether they are simultaneously influenced by other components. The influenced components which were still waiting for their next internal event evolves according to their external transition function, as in CDEVS.

Even though CDEVS and PDEVS are a bit different, the class of systems they can model is the same. It has been proven that CDEVS includes PDEVS [Zeigler et al. 2000]; the reverse relation is still an open problem, but many scientists suspect it actually holds. Therefore, the choice of one or the other is mainly a matter of taste, or rather of adequacy to the system to model. CDEVS prevents simultaneous events by prioritizing components, while PDEVS allows them and lets the modeler resolve conflicts as he sees fit. The latter maps more closely to what happens in natural systems, hence a certain preference of the community for it.

All the propositions we make in this dissertation relates to CDEVS; at the beginning of this PhD, we had to choose between the two main formalisms and went for the one which appeared first, historically. However, all the ideas exposed in this document can be adapted to PDEVS and some other DEVS variants, more or less easily.

II.2. DEVS simulation algorithms

In the previous section, we explained how to specify a system using DEVS, either as an atomic model or a coupled model. However, the ultimate goal of these models is to simulate the behavior of the system, i.e. to obtain a temporal trace of its state and/or outputs over a

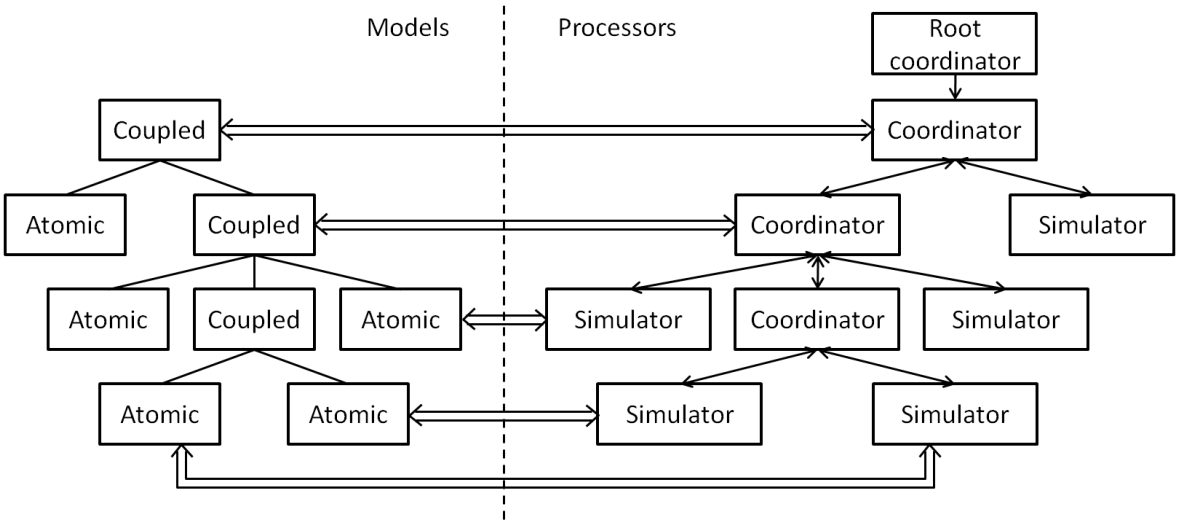


Figure 3. Mapping between model hierarchy and processor hierarchy

given period of time. This transformation from an abstract representation to a data trajectory must be done in a way that preserves the semantics of the model. [Zeigler et al. 2000] provides algorithms (called abstract simulators) that correctly "interpret" DEVS models. Since then, other algorithms have been invented, but the works presented in this thesis, especially the DEVS-MetaSimulator described in Chapter 6, are based on the original ones, so we will only present these.

II.2.1. Overall architecture

We saw previously that DEVS models can be composed in coupled models with a tree-like structure. The simplest way to simulate such models is to map them to a hierarchy of processors, each processor being in charge of handling one component. These processors perform the simulation by querying their models and exchanging simulation messages. Depending on their kind, atomic or coupled, components must be handled differently; therefore, the simulation hierarchy contains two types of processors: coordinators to process coupled models, and simulators to process atomic models. In addition to these, a particular processor at the top of the hierarchy, called root coordinator, handles the main simulation loop. A sample mapping between a coupled model and its simulator is provided in Figure 3.

Before beginning the simulation, the root coordinator sends an initialization message (i-message) to its child processor, which is forwarded down the hierarchy so that every

simulator initializes itself and the model it handles. After this, the root coordinator starts the simulation loop. At each step of the simulation, it sends an activation message (**-message*), which is forwarded to the simulator of the most imminent component. This component undergoes an internal transition, after having generated some output events. These outputs are embedded into output messages (*y-message*) that are sent up the hierarchy to the coordinators. These latter forward them to their parents, or convert them into input messages (*x-message*) for some of their children, according to the couplings specified by the coupled models. Eventually, input messages are received by simulators, which trigger external transitions in their corresponding component. When all the influences have been dealt with, the root coordinator updates the simulation clock and advances to the next simulation step, where it repeats the same operations, i.e. sending an activation message and updating time, until the end of the simulation.

In the following paragraphs, we detail the algorithms for each kind of processor: simulator, coordinator and root coordinator.

II.2.2. Processor for atomic DEVS models (simulator)

A DEVS simulator is associated with an atomic component (`component`) holding a total state `(s,e)`. It keeps a reference to its parent coordinator in the hierarchy (`parent`), in order to forward the output events generated by the component. It also stores the time of the last event undergone by the component (`tl`), and the time of the next internal event scheduled (`tn`).

The simulator's behavior is driven by the messages sent by its parent coordinator. Depending on the type of message received, the simulator performs different operations over its component, and possibly generates new simulation messages. The pseudo-code in Code 1 describes the algorithm of a DEVS simulator.

At the beginning of the simulation, all processors are initialized; in the case of the simulator, this boils down to initializing the times of last and next event, depending on the initial simulation time and the state of the component. The theory [Zeigler et al. 2000] does not specify how the initialization of components should be handled, notably how their initial state should be determined. The convention in most DEVS simulation tools is to either hard-

```
when receiving an initialization message at time t
  | tl = t - component.e;
  | tn = tl + component.ta(s)

when receiving an activation message at time t
  | assert t == tn // time of next event must be reached
  |
  | send output message to parent with value (component.id, component.λ(s))
  | component.s = component.δint(s)
  |
  | tl = t
  | tn = t + component.ta(s)

when receiving an input message at time t with value (iport, x)
  | assert tl ≤ t ≤ tn // last event must have happened;
  |                    // internal event must be pending
  | component.e = t - tl
  | component.s = component.δext(component.s, component.e, (iport, x))
  |
  | tl = t
  | tn = t + component.ta(s)
```

Code 1. DEVS simulator algorithm.

code the initialization in the atomic model, or to provide an initialization function to allow different initializations and the reset of a component to its initial state. In this work, we will use an approach akin to the one proposed in [Quesnel 2006], where the author defines a small addition to the formalism to include parameterization and initialization.

The reception of an activation message means that the component must undergo an internal event. The simulator starts by querying the component's output, through its output function, and forwards it to its parent coordinator, along with the component identifier. Then, the state of the component is changed according to its internal transition function. Finally, the simulator updates the times of last and next event.

Between a transition and the next internal transition, a component can receive an external event, coming from some other components. Such events are encapsulated into input messages exchanged between processors. When a simulator receives an input message, it updates the elapsed time since the last event (e) and modifies the state of the component according to the next external transition function. Then, it updates the times of last and next event.

II.2.3. Processor for coupled DEVS models (coordinator)

A DEVS coordinator is associated with a coupled component (i.e. an instance of a coupled model) (`component`). Like a simulator, it keeps a reference to its parent coordinator in the hierarchy (`parent`), but also a list of children processors corresponding to the children components, indexed by component names (`children`). Finally, it also keeps track of the time of last event (`tl`) and time of next event (`tn`).

A coordinator receives from its parent the same kind of messages as a simulator, namely initialization, activation and input messages, but handles them differently. In addition, it can also receive output messages from its children processors. The pseudo-code in Code 2 describes the algorithm of a DEVS coordinator.

At initialization, the coordinator simply triggers the initialization of each of its child processor, and initializes the times of last and next event. To determine these times, the coordinator queries its children and takes the time of the latest event, and the time of the soonest scheduled event.

When the coordinator receives an activation message, it must determine which of its children to activate. To do so, it starts by retrieving the set of processors whose delay is expired, in other words those whose time of next event is equal to the current simulation time. Then, the coupled model's tie-breaking function is used to select which of the imminent components has priority. The coordinator activates the processor associated with this component, and updates the times of last and next event.

Upon reception of an input message, the coordinator queries the external input coupling to construct a list of destinations, meaning a list of influenced components, along with the ports on which the value must be sent. The coordinator looks up the processors associated with these components, and sends each of them an input message, constructed with the destination port and the value received. If the receiving processor is a coordinator, it carries out the same sequence of operations, sending input messages to some of its children. Thereby, the external event gets forwarded down the processor/model hierarchy until reaching simulators/atomic models.

```
when receiving an initialization message at time t
  | send initialization message to children with value t
  |
  | tl = max(child.tl | child in children)
  | tn = min(child.tn | child in children)

when receiving an activation message at time t
  | assert t == tn // time of next event must be reached
  |
  | imminent processors = {child | child in children and child.tn == t}
  | imminent components = {processor.component.id |
  |                        processor in imminent processors}
  | component to activate = component.select(imminent components)
  | processor to activate = children[component to activate]
  | send activation message to processor to activate
  |
  | tl = t
  | tn = min(child.tn | child in children)

when receiving an input message at time t with value (iport, x)
  | assert tl ≤ t ≤ tn // last event must have happened;
  |                    // internal event must be pending
  | for each destination in component.EIC(iport)
  |   | receiver = child | child in children and
  |   |                    child.component.id == destination.component
  |   | send input message to receiver with value (destination.port, x)
  |
  | tl = t
  | tn = min(child.tn | child in children)

when receiving an output message at time t with value (source, (oport, y))
  | for each destination in component.IC(source, oport)
  |   | receiver = child | child in children and
  |   |                    child.component.id == destination.component
  |   | send input message to receiver with value (destination.port, y)
  |
  | if self-port = component.EOC(oport) exists
  |   | send output message to parent with value
  |   |     (component.id, (self-port, y))
```

Code 2. DEVS coordinator algorithm.

The coordinator must also handle another type of forwarding, namely that of the outputs of its components. These outputs are represented in the simulation by output messages sent to the coordinator by children processors. When the coordinator receives such a message, it determines the list of component ports where the event must be sent, according to the internal coupling and the source port. The coordinator then retrieves the processors associated with these destination components, and sends each of them an input message,

containing the correct destination port. Output ports of components can also be connected to output ports of the coupled model. Hence, the coordinator must query the external output coupling to (possibly) get the port where the output event must be forwarded. If such a coupling exists, an output message is sent to the parent coordinator, which will in turn forward the event according to the coupling information specified by its component, and so on until all the influenced components have received the event.

Some implementations use an alternative approach for determining the component to activate and the time of next event. Instead of querying the `tn` of each child every time an activation or input message is received, the coordinator can keep a list of processors ordered by their `tn` and the select function. With such a sorted list, the activation of a component boils down to taking the first element of the list and sending it a message. However, the list must be kept up to date after each transition (internal or external) of the model. We are not aware of any studies about which approach is more efficient; we chose to describe and use just-in-time selection of the imminent component because it is much simpler and possibly a bit more efficient. Indeed, sorting implies many comparisons and swaps, in addition to the small overhead memory incurred by keeping a separate list of processors. Moreover, the sorting is hard to do correctly since there is no order over the component set, due to the select function not being a binary relation (it operates on an arbitrary-sized set of components).

II.2.4. Top-level processor (root coordinator)

At the top of the processor hierarchy lies a root coordinator, in charge of handling the main simulation loop. The algorithm of this processor, very simple, is summarized in Code 3:

```
send initialization message to child with initial simulation time t
t = child.tn
while simulation is not over
  | send activation message to child with time t
  | t = child.tn
```

Code 3. DEVS root coordinator algorithm.

The root coordinator has a child processor, usually a coordinator. After sending its subordinate an initialization order (that will trigger the initialization of the entire hierarchy), the root coordinator starts the main simulation loop. At each simulation step, it sends an

activation message to its child with the time of next event, this time being determined by the child, as we saw previously.

The condition for stopping the simulation is not specified, so the implementations are free to decide how to handle the halt of the simulation. Common conditions include:

- a given number of steps has been performed;
- a given simulation time has been attained;
- there are no more events scheduled (tn = ∞).

In addition to this main algorithm, the root coordinator can also provide a default behavior for processing unhandled output events. Indeed, it can happen that the simulated model is not "closed", that it has some input and output ports. In this case, since the output ports are not connected to anything, it is the root coordinator which will receive output messages corresponding to events on these ports. An implementation is free to forbid the simulation of non-closed models, or to provide a default processing of unhandled events (which can be ignoring them).

These simulation algorithms, as well as some alternative ones, have been implemented in many DEVS simulation tools. In the next section, we provide a short description of the most prominent ones.

II.3. Existing DEVS tools

II.3.1. CD++

CD++ [Wainer 2009] is a DEVS M&S toolkit developed at the Carlton University under Gabriel Wainer supervision. It provides a library of C++ classes to specify models in several DEVS formalisms and simulate them. The supported formalisms include CDEVS and PDEVS, but the focus is on Cell-DEVS, an extension integrating cellular automata and DEVS.

CD++ can handle in a single simulation several types of models, e.g. parallel DEVS and Cell-DEVS models. Several simulation algorithms are implemented, and simulations can be performed either locally or remotely, by sending model specifications to a simulation server. Depending on their types, CD++ models are specified either as C++ classes (atomic models) or through text files following a custom format (coupled models, Cell-DEVS models).

In addition to the simulation kernel, CD++ provides an Eclipse plugin allowing the edition of DEVS and Cell-DEVS models both textually and graphically, as well as the graphical visualization of Cell-DEVS simulation results.

II.3.2. DEVSJava

DEVSJava [DEVSJava 2004] [Zeigler and Sarjoughian 2003] is a Java library for modeling and simulating PDEVS, Dynamic-Structure DEVS and Real-Time-DEVS models. It is developed at the Arizona Center for Integrative Modeling & Simulation (ACIMS) (University of Arizona; Arizona State University), co-directed by Bernard P. Zeigler, Hessam Sarjoughian and Roman Lysecky.

DEVSJava provides a set of custom container classes for storing entities manipulated by models. These containers are used to develop DEVS models according to the class hierarchy defined by the library. Models are specified as Java classes deriving from one of the base DEVS classes provided. These models can then be simulated with the simulation processors implemented in the library. Several algorithms are included: local simulation using a direct implementation of the abstract simulators, distributed simulation over a network, real-time simulation.

Later versions of DEVSJava include the possibility to perform dynamic-structure modeling, meaning that the structure of models (ports, components, couplings) can be modified at runtime, during the simulation.

The DEVSJava library is now included in a larger software suite for M&S called DEVS-Suite, which provides some graphical facilities for editing models, controlling simulation and visualizing results [Kim et al. 2009].

II.3.3. James II

James II [Himmelspach and Uhrmacher 2007] [Himmelspach 2007] is a generic M&S platform, written in Java, which is developed at the Rostock University under the coordination of Adelinde M. Uhrmacher. Its aim is to provide an extensible platform that can integrate any modeling formalism, simulation algorithms, and tools. To do so, it uses a flexible plugin system that makes the addition of new features in the platform quite seamless.

The architecture of James II focuses on minimizing coupling between modules. The core of the platform provides a set of services and classes for use by the other packages, such as random number generation, data structures, mathematical functions, serialization, etc.). It also handles the graphical user interface and the plugin system.

The other features are implemented as plugins providing a suitable interface for integration into the platform, and are usually bundled in cohesive packages. The most important types of plugins are modeling formalisms, simulation algorithms, editors and visualizers, but other plugin types can be defined. Many plugins have already been developed (more than 500 in late 2009 according to James II documentation); for instance, the last version at the time of this writing (v0.8.6alpha) comes bundled with several formalisms, among others DEVS, PDEVS, PdynDEVS and cellular automata, along with various simulation algorithms (sequential, multi-threaded, etc.), editors and visualizers for each.

II.3.4. MIMOSA

Like James II, MIMOSA [Müller 2010] (Méthodes Informatiques de MOdélisation et Simulation Agents – computer science methods for agent-based modeling & simulation) is an extensible M&S environment that allows multi-formalism modeling. It is developed mainly by Jean-Pierre Müller, at the International Cooperation Center in Agricultural Research for Development (CIRAD).

The main characteristic of MIMOSA is its focus on ontologies. In MIMOSA, the practitioner starts by identifying the entities at stake in his domain, along with their relations. This knowledge is captured in ontologies, which defines the vocabulary of the domain considered. This high-level representation can then be adorned with dynamics that describe the behavior of the various entities. To do so, several paradigms are supported, ranging from basic scripting to more structured modeling using various formalisms such as state charts or differential equations. Thanks to a plugin system, MIMOSA can easily be extended with new formalisms.

Once this is done, the practitioner can define concrete models, which are composed of instances (individuals) of the categories specified in the ontologies. These models are then

processed by a DEVS-based simulation kernel, in charge of coordinating the various components and handling the communication between them.

MIMOSA is not really an environment for DEVS M&S *per se*: it is rather a tool for multi-formalism modeling that uses DEVS as a simulation middleware. Nevertheless, even though MIMOSA encourages the use of more abstract formalims, it still provides support for direct DEVS modeling to some extent, by allowing operational semantics to be defined in a way akin to DEVS atomic models.

II.3.5. PythonDEVS

PythonDEVS [Bolduc and Vangheluwe 2002] is a minimalist Classic DEVS simulator written in Python. It was developed at the Modelling, Simulation and Design Lab (McGill University) under Hans Vangheluwe supervision.

PythonDEVS is a pretty straight-forward implementation of the Classic DEVS abstract simulators. It provides base classes for atomic and coupled models, and the corresponding solvers to perform simulations.

II.3.6. Virtual Laboratory Environment (VLE)

The Virtual Laboratory Environment [Quesnel et al. 2009] [Quesnel 2006] is an M&S platform based on PDEVS, written in C++ and mainly developed by Gauthier Quesnel for the French National Institute for Agricultural Research (INRA) and the Signal and Image Computer Science Laboratory of the Opal Coast (LISIC). The initial development was done under the supervision of Eric Ramat, who is still involved in the evolution of VLE. VLE is now integrated in the RECORD platform supported by the Applied Mathematics and Computer Science department of INRA (named MIA).

Like James II, its architecture is quite modular to facilitate the addition of new features in the platform. The core of VLE consists in a set of class libraries, the VLE Foundation Libraries (VFL), which provides a collection of classes that form the base layer of the platform. These classes implement several modeling formalisms (Petri nets, 2D/3D cellular automata, Quantized State Systems, etc.) in a way that allow heterogeneous models to be seamlessly integrated in a PDEVS coupled model, thereby providing multi-modeling facilities.

To simulate these models, VLE provides a PDEVS simulator, which handles the initialization of models through initialization ports, and their observation through observation ports that can be connected to different views. These views are used to provide simulation results in various formats, which can be visualized graphically and in real-time with the provided display tool, the Eyes Of VLE (EOV).

For editing models, VLE includes a graphical user interface (GUI for VLE (GVLE)); it allows designing coupled models in a graphical way, those models being serialized in a custom format. Atomic models must be defined as C++ classes implementing one of the formalisms provided.

III. Limits of the current DEVS Modeling & Simulation software

III.1. The challenge of standardization and interoperability

In the previous section, we presented some of the prominent tools for DEVS M&S, but there are many others. The consequence of this multiplication of frameworks, platforms and simulators, written in different programming environments, is that model sharing between scientists is very hard and often implies rewriting models from scratch to accommodate some tool.

In an effort to overcome this limitation, the DEVS community formed a standardization group [DEVS Standardization Group] to discuss and propose standards for the DEVS formalism. This group identified three main issues to tackle: define a “DEVS kernel”, i.e. the set of minimum requirements a tool should fulfill to be labeled “DEVS-compliant”, make existing DEVS tool interoperate, e.g. by relying on standard interfaces and some underlying communication protocol; and come up with a standardized format for representing DEVS models, to facilitate their exchange between scientists and their use in different tools.

Until now, most propositions have focused on the two latter tasks. [Wainer and Mosterman 2010] provides a comprehensive overview of the different approaches that have been explored. These approaches aimed at attaining two different and complementary objectives: defining a standard simulation middleware to integrate heterogeneous DEVS applications,

and defining a standard format for representing DEVS models, to make them tool-independent.

III.1.1. Standardized interoperability middleware

The aim of a standardized interoperability middleware is to allow heterogeneous models, i.e. models written in different frameworks, different programming languages, to be coupled and simulated seamlessly, without the need to port them to a given environment. These models can reside on the same machine, on servers distributed over the world, or can be stored on public repositories accessible by the community.

This issue is not new in software engineering: the integration of heterogeneous applications that use various technologies and tools has been studied quite thoroughly for several years. Enterprise Application Integration (EAI) provides a set of methods, technologies and tools for making a set of disparate applications operate together, usually by relying on an hub-and-spoke scheme or a bus architecture. Later, a more general and flexible domain emerged, which focused more on interoperability than integration: Service Oriented Architecture (SOA).

One of the current trends in software engineering is to develop applications not as monolithic entities like they were a few years ago, but as a collection of loosely-coupled/highly-cohesive components that can be easily combined, substituted and reused, named services. In addition to the notion of "Software As A Service" introduced a decade ago [Keith at al. 2000] and now knowing business success, the Service Oriented Architecture (SOA) is now commonly implemented through web services (WS), a widely used technology based on several accepted standards [Newcomer and Lomow 2004] that provides an appealing alternative to the more complex Common Object Request Broker Architecture (CORBA). We leave aside language-specific APIs such as Java RMI, Java EE or .Net Remoting, as they limit their applicability to single languages.

Web services come in two flavors: SOAP-based WS and RESTful WS. SOAP-based services rely on an important stack of standards that standardize interfaces description (Web Service Description Language – WSDL), messages definition (XML Schema) and communication protocol (Service Oriented Access Protocol – SOAP). These last years, the relative complexity

of this architecture has been criticized, giving rise to the emergence of an alternate architecture conforming to the REpresentation State Transfer (REST) constraints. Instead of defining custom interfaces, RESTful WS constrain their interfaces to the standard HTTP operations (GET, PUT, POST, DELETE), making them more integrated in the Web and easier to deploy and access, at some cost (lack of standards for cross-cutting concerns such as security or reliable messaging, dependence on HTPP, …).

In [Wainer et al. 2010], we identified two possible SOA approaches to tackle the issue of DEVS interoperability: *simulator-based interoperability*, where disparate simulators cooperate through standard interfaces, and *model-based interoperability*, where heterogeneous models expose a standard interface, making them queryable by any compliant simulator. To expose these propositions, we will assume the use of SOAP-based WS, but they can easily be adapted to other SOA technologies such as CORBA or RESTful WS. Similarly, we will consider the distributed case, where simulators/models live on distant servers, but the approaches apply equally well to the local case.

III.1.1.1. Simulator-based interoperability

The main idea of this approach, as used in DEVS/SOA [Seo 2009] [Mittal et al. 2009] and DCD++ [Al-Zoubi and Wainer 2008] [Al-Zoubi and Wainer 2009], is to have a collection of simulation services distributed over the internet. These services provide several operations for simulating atomic or coupled DEVS models in a unified manner, by using the DEVS simulation protocol. The overall simulation is coordinated by a main service, which acts like an entry point for the user.

To illustrate this architecture, Figure 4 depicts a heterogeneous coupled model composed of DEVSJava and CD++ models. The simulation of these models is distributed over several simulation services, either implemented as DEVSJava or as CD++, that expose a standard interface to make them interoperable, as Figure 5 shows.

The simulation services can be written in disparate programming languages, using different technologies, as long as they provide a standard interface that can be queried remotely (or a wrapper exposing such an interface). Each service handles one or several models conforming

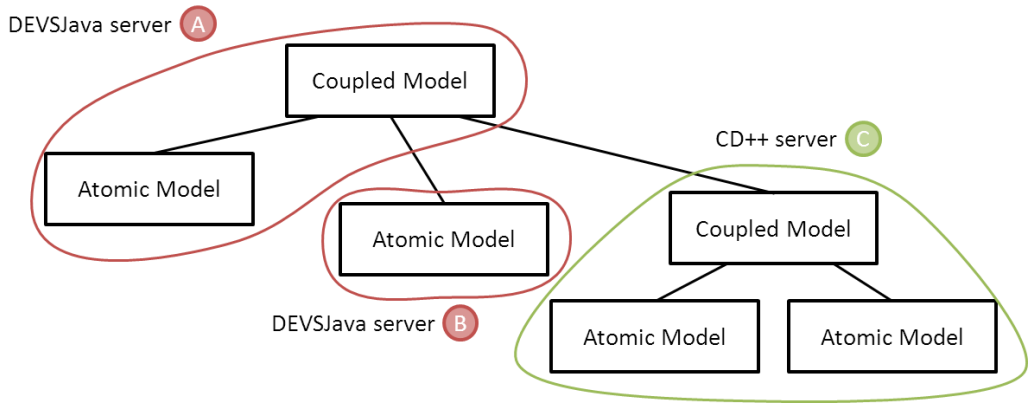


Figure 4. Heterogeneous model to be distributed over several simulation services.

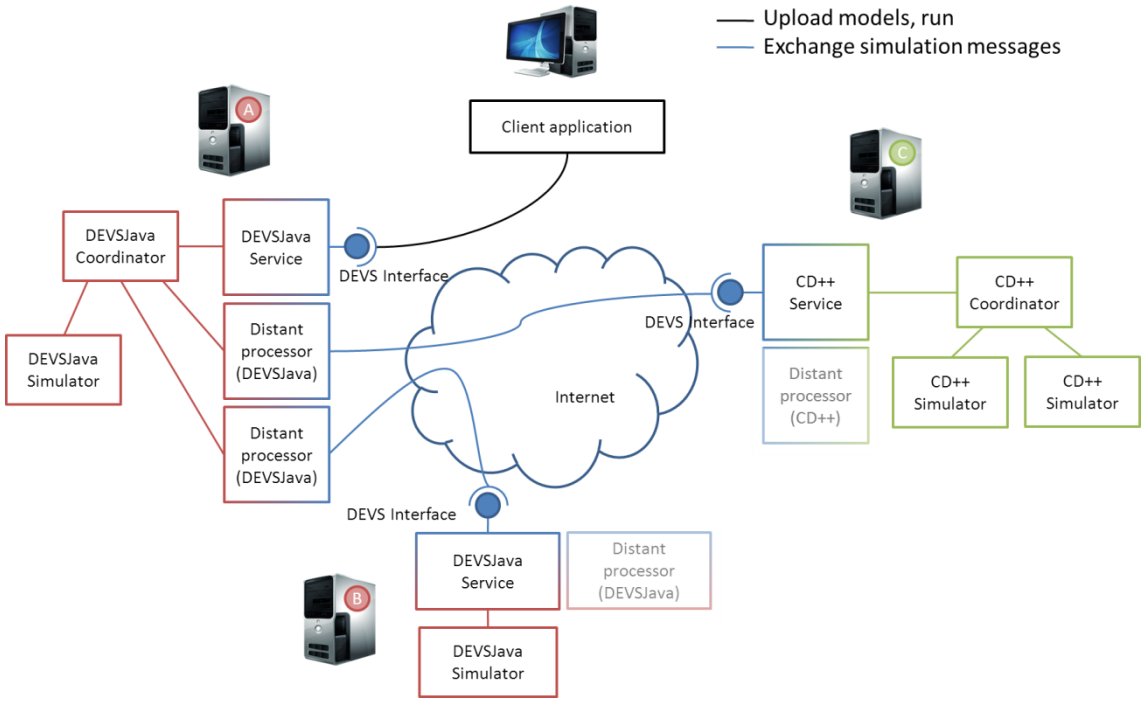


Figure 5. DEVS components interoperability through simulators communication

to the underlying tool; these models can either be already present on the server, or be uploaded by the client, for instance if it was retrieved from a public model repository.

In order to communicate, the DEVS services expose a standard interface that accepts the usual simulation protocol messages: *initialize*, *get time of next event*, *run transition* and so on. These messages must then be translated into requests understandable by the underlying simulator: the tool provider must devise a way of integrating the standard interfaces within its simulation hierarchy.

From a user perspective, the simulation process consists in:

1. Writing a coupled DEVS model, using his favorite framework.
2. Selecting a list of DEVS servers on which he wishes to distribute the simulation.
3. Deploying DEVS models to the appropriate servers (either by uploading them from his location or by activating existing remote models).
4. Running the simulation.

III.1.1.2. Model-based interoperability

This second type of interoperability is based on the fact that most of the time modelers already have or can easily install a DEVS simulator on their machine. However, it is much more difficult for them to retrieve and reuse models already existing on the computers of other scientists. Therefore, solutions are needed to provide access to and share these numerous models.

A solution, described in the next section, is to standardize DEVS models representation so that they can be specified independently of the platform, facilitating their exchange among the community. Another approach is to use a service-oriented architecture, pretty much like the one described in simulator-based interoperability, except that the services deployed in this case are models instead of simulators.

With such architecture, a simulation does no longer need to be distributed over several servers; it can be executed locally, using a single DEVS simulator. The interoperability is done at the model level: by exposing a standard interface, models can be queried by distant simulators in a homogeneous way, allowing them to be included seamlessly in any simulation. Figure 6 shows how a coupled model, written in a given environment, can include local atomic models (in the same environment) and distant atomic models (possibly written in another environment).

The easiest way to make a simulator amenable to this kind of interoperability is to write a simple adapter stub that appears to the simulator as a regular model, but forwards all the requests it receives to a distant model service. As a consequence, the operations invoked through the network are no longer simulation mechanisms, but model functions, such as δ_{int}, δ_{ext}, the time advance function, etc.

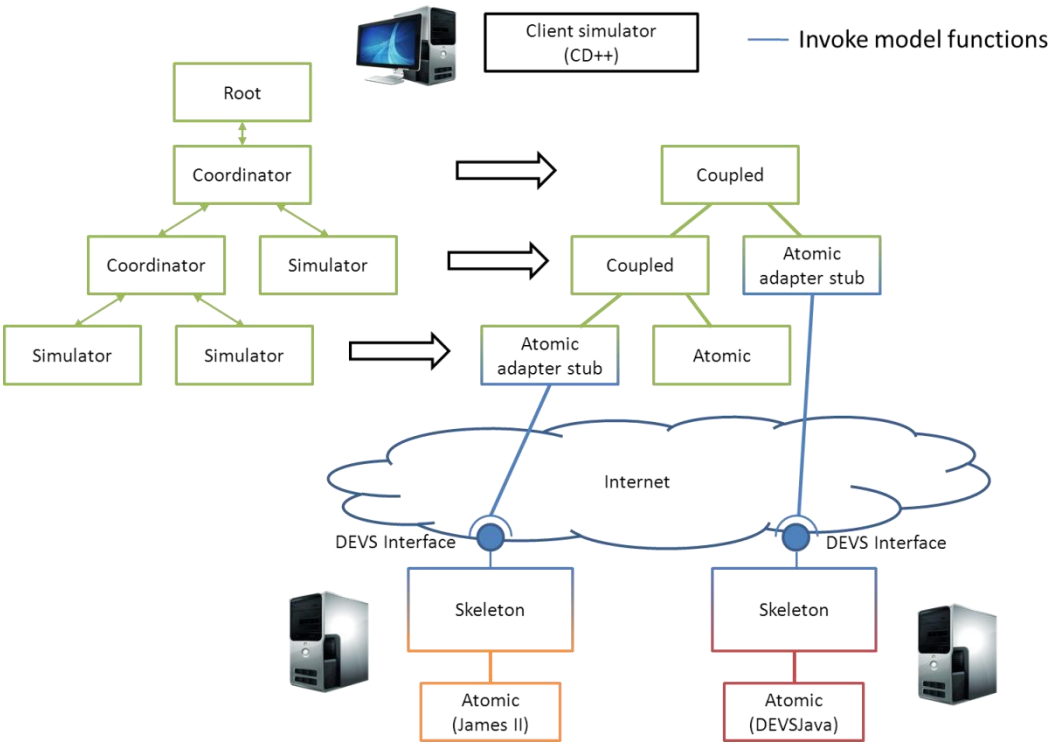


Figure 6. Local simulation of distant and local models

If all model services provide the same standard interface, only one adapter stub needs to be written for each existing simulation tool. Similarly, to make existing models accessible as services, one simply needs to provide a skeleton to receive the standard requests sent by the stubs, and translate them into platform-specific requests understandable by the models (and do the reverse operations for responses). The writing of stubs and skeletons can be greatly simplified by the use of code generation tools.

Regarding distant coupled models, three solutions are possible. The first is to expose them as services too. Like atomic models, they can be wrapped so as to provide a standard interface for coupled models, allowing distant simulators to query their list of components, their couplings and so on. The second solution is to use the closure under coupling property to develop facades making coupled models appear as atomic ones. This way, coupled models are handled transparently, as if they were atomic. The third possibility is to rely on a standardized representation of coupled models. If the coupled model is described in a standardized way, it can be downloaded by the simulation client and integrated into the simulation, either by generating a platform-specific version or by using an appropriate

interpreter. The standardization of models representation is discussed in the following section.

From a practical perspective, once the appropriate stubs/skeletons have been written, a model provider needs to:

1. Write a model in his favorite DEVS framework.
2. Deploy the model as a service, wrapped in the generic skeleton.

For its part, the model consumer has to:

1. Find the location of the models he needs, using some kind of model directory or more classical discovery procedures such as web searches or acquaintances.
2. Write a coupled model in his framework of choice, possibly using both local and distant models, these latter being wrapped in the generic stub.
3. Run the simulation. The framework's simulator is unaware of the presence of both local and distant models.

To sum up the two interoperability solutions we exposed, we can say that the simulator-based approach aims at integrating heterogeneous tools by distributing the simulation over distant servers, whereas the model-based approach aims at integrating distributed heterogeneous models in a local simulation. The two solutions are conceivable, but provide different advantages and drawbacks. The simulator-based approach has an important potential when it comes to parallel simulation, but forces all simulation services to use compatible algorithms. Using a different simulation algorithm is therefore quite difficult, especially if that implies modifying the simulation interface. On the other hand, the model-based approach does not put any restriction on the algorithm, which can be modified very easily since it is localized in one place. Moreover, coming up with a standardized interface for models is arguably simpler than standardizing a simulation protocol. However, this approach makes it harder to integrate non-DEVS models, whose interface cannot be easily mapped onto DEVS'.

III.1.2. Standardized model representation

In the previous section, we showed how SOA could be used to provide an interoperability middleware for integrating heterogeneous DEVS environments. In this section, we present a different approach that aims at representing models in a platform-independent format so that they can be exchanged and used in different tools.

Ideally, a modeler should be allowed to perform the following operations:

- store model specifications and data trajectories for future use;
- perform reasoning on these specifications, e.g. select a model in a repository based on some criteria, or execute analysis on a model structure or a generated output trajectory;
- share models among the community, to enhance productivity and quality by reusing well-tested and validated models;
- visualize simulation results through different interfaces (charts, animations), and graphically create/edit models;
- transform model specifications into executable artifacts for several target simulators, and perform simulation.

Due to the lack of a standard representation, these use cases are currently rather difficult to fulfill. Most actual models are described either in natural language, problematic – if not impossible – to process with a computer, or in a specific programming language, using types and functions peculiar to one DEVS environment. This heterogeneity obstructs the collaboration between modelers, and makes impossible the development of generic tools that could be used by the entire community. Consequently, the need for a standardized way of representing DEVS models has become pressing.

This standard should encompass all the aspects of DEVS M&S: description of model structure (ports, couplings, etc.) and dynamics (δ_{int}, δ_{ext}, etc.), design of experiments (parameterization, input events trajectory), results (output events trajectory, state trajectory). Ideally, it should also be compatible with existing implementations, so that a model written for a given environment could be retro-engineered (preferably automatically) into a standard representation.

Several works have proposed platform-independent language for specifying DEVS models, which can serve as bases for a common reflection of the community, eventually leading to an agreed-upon representation format that could then be formally standardized. Most of these propositions rely on the eXtensible Markup Language (XML [Abiteboul et al. 1999]); indeed, a consensus emerged among the M&S community about XML being a suitable technology for describing models in a standardized way [Brutzman and Zyda 2002] [Fishwick 2002]. There are numerous arguments supporting this choice, such as the wide use of XML in SOA, the great number of existing XML tools, its platform independence and many others.

One of the earliest uses of XML for describing DEVS models can be found in [Wang and Lu 2002]. In this work, the authors present a tool allowing the modeler to graphically build System Entity Structures [18], which can be used to capture the DEVS models structure. The graphical representation is then transformed into an XML document containing the model hierarchy, conforming to some XML Schema.

Based on this premise, [Rhöl and Uhrmacher 2005] define XML Schemas for the DEVS formalism, including both coupled and atomic models. Model structure specification includes ports, couplings and state variables, and the emphasis is put on the use of XML data binding to automatically generate classes corresponding to XML models. However, the representation of the model dynamics is rather limited since each function is defined by a raw string, written in a specific programming language.

This problem of behavior representation is tackled in several works. One approach [Risco-Martín et al. 2007] is to restrict the different functions to simple constructs. For example, the transition functions are described by a set of conditional expressions, each being associated with new values for the state variables. A more general and powerful solution is to use some kind of pseudo-language, as is proposed in [Janoušek et al. 2006]. The advantage of using a simple language is that it can be easily represented in XML and transformed into source code in any programming language. Finally, [Mittal et al. 2007] use an existing XML-based source code representation. In this case, the favored language is Java, but the authors evoke the possibility to use a more generic XML language to describe the model logic.

Based on these propositions, we developed an XML Schema [Touraille et al. 2009] that tries to integrate all these ideas while at the same time being as general as possible, in order to not restrict its applicability. While doing so, we realized that XML was probably not the most appropriate tool for the job, especially when it comes to manipulating model specifications. Therefore, we turned to a full-fledged Model-Driven Engineering (MDE) environment that provides a higher level of abstraction to work with models, metamodels, and transformations, while still relying on XML to store and exchange data.

This work, presented at length in Chapter 5, is akin to the work of Hans Vangheluwe and its team, who leveraged their metamodeling environment Atom^3 [De Lara and Vangheluwe 2002] in the context of DEVS M&S. In [Posse and Bolduc 2003] and [Levytskyy et al. 2003], they show how metamodeling and model transformation through graph rewriting allow the generation, from a DEVS metamodel, of graphical editors, model specifications understandable by external tools (in this case PythonDEVS), or artifacts to be deployed on web servers.

III.1.3. Interoperability and standardized representation: two complementary approaches

At first sight, the two standardization approaches exposed previously seems quite unrelated, as they have quite different objectives: the standardization of a simulation middleware aims at making heterogeneous tools cooperate to co-simulate a model, while the standardization of model representation aims at making models tool-independent to facilitate their sharing and their manipulation by generic software.

However, some interesting advantages can be obtained by combining the two. Indeed, a standard representation of models would greatly facilitate the implementation of model-based interoperability. Assuming that every model written for a specific tool can be retro-engineered into a standard specification, there is no longer any need for standardizing model services interfaces. The standard model representation can be transformed into a model-specific service description (in IDL (CORBA), WSDL (SOAP-based WS) or WADL (RESTful WS), for instance), which can then be used to generate model-specific stubs and skeletons on clients and servers. Such a solution would allow the interfaces to be more

explicit, permitting for example the restriction of events sent to models to certain types, at the cost of a reduced flexibility.

We showed in this section some approaches to DEVS standardization. It is our feeling that this standardization is essential, as it will not only provide interoperability between existing tools, but also facilitate the development of new implementations and transition to those. When developers will be able to rely on a common standard, they will be more eager to provide new algorithms, new libraries and new environments for DEVS, since these latter will be more likely to be adopted by the community. This should greatly benefit DEVS M&S, by allowing implementations of ever-increasing quality to appear.

Even though there is not yet a DEVS standard, we decided to propose “yet another” DEVS simulator, which uses an original approach, namely metaprogramming, to address some of the actual implementations’ shortcomings, which we describe hereafter.

III.2. Possible improvements to current DEVS implementations

III.2.1. Performance

As our knowledge grows, models tend to become more and more complex. Accordingly, the time needed to simulate these models tends to increase, to the point where performance of simulation becomes a concern.

Several works have been made to improve the execution time of DEVS simulators. Most of them focus on parallelizing simulation on different architectures, for example on a single machine [Nutaro 2010], on a computing cluster [Kim and Kang 2004] [Feng et al. 2008], or on a computing grid [Seo et al. 2004]. Several algorithms have been developed, mostly variants of conservative (error-free) and optimistic (error detection and rollback) approaches [Jafer 2011].

Other works focus on analyzing the activity [Muzy and Hill 2011] of models in order to allocate computing resources accordingly. This approach has shown its efficiency, for instance by allowing real-time simulation of fire-spreading [Muzy and Nutaro 2005].

This thesis does not get onto these subjects. However, the DEVS simulator we will present implements to some extent another DEVS simulation optimization called flattening.

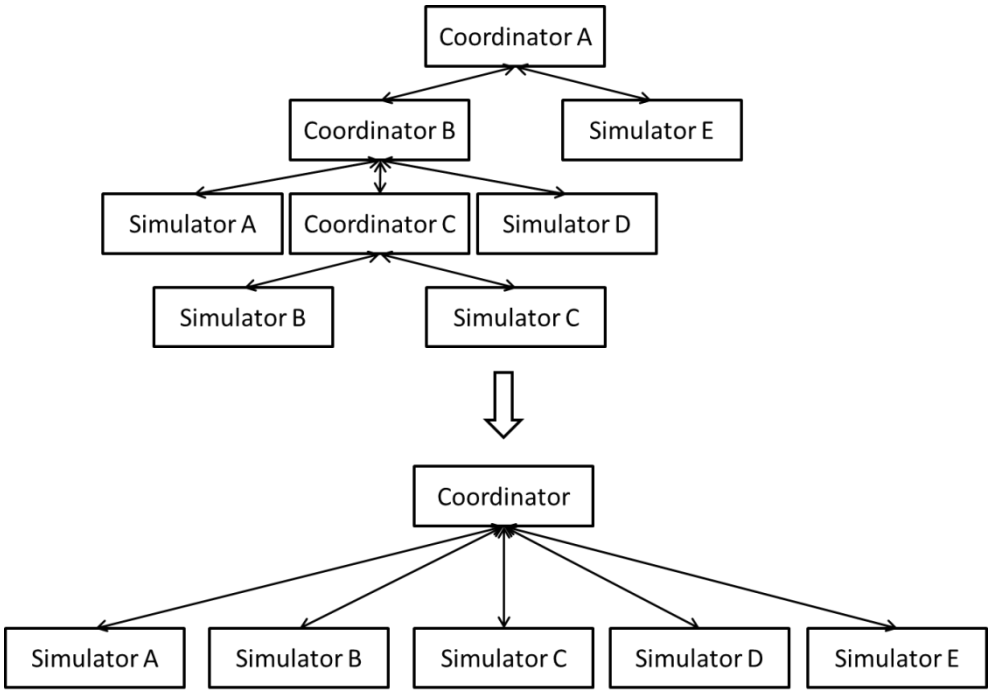


Figure 7. Flattening of a processor hierarchy.

Flattening aims at decreasing the simulation overhead; indeed, a straightforward implementation, which follows the abstract simulators described in Chapter 2.II.2 to the letter, incurs a huge overhead due to message passing. In such implementation, forwarding an event from a model to another can imply passing messages up and down a deep processor hierarchy. To solve this issue, flattening transforms the processor hierarchy into a flat simulator, without any coordinator or with a single coordinator handling all simulators. Figure 7 gives an example of a flattened processor hierarchy, where the depth has been reduced from three to one.

Flattening can be performed either at the modeling level or at the simulation level. For instance, [Kim et al. 2000] transform the structure of DEVS models to limit the message exchange between distributed simulators. Similarly, [Chen and Vangheluwe 2010] apply flattening to the model and not the simulator. An interesting thing about the latter work is that authors provide a formal method for flattening, which is then implemented in Modelica.

The transformation of the processor hierarchy is more common. It is implemented in different ways in CD++, either to limit the number of inter-process messages in the case of parallel simulation [Jafer and Wainer 2009] or to improve performance so as to meet real-

time constraints [Glinsky and Wainer 2002]. In James II, the model structure is not altered in order to keep it observable. The flattening is performed onto the processor hierarchy, to the point where all models are handled by a single processor [Himmelspach and Uhrmacher 2006].

It is important to note that flattening can sometimes be less efficient than advertised above. Indeed, the gain obtained by avoiding unnecessary message exchanges can sometimes be counterbalanced by other operations becoming more complex without a hierarchy of processors. This is notably the case for the determination of the time of next event: after flattening, a single coordinator is responsible for querying all simulators at each simulation step, or for keeping track of the scheduled internal events in some data structure that can be costly to manipulate. On the other hand, when processors are organized hierarchically, the update of the time of next event can be done simply and efficiently, because only some of the processors need to update at each simulation step.

Performance is an important aspect of simulation, especially when modeling complex systems. However, it is not the only interesting property of simulation environments. Another feature that is very important and that we will tackle in this work relates to model verification.

III.2.2. Model verification

Model verification consists in verifying that a model is correctly implemented, that its specification does not contain errors or inconsistencies. This is not to be confused with model validation, which aims at checking the concordance of the model with the system under study.

Model verification can be done in several ways. The first is to analyze the specification – automatically or manually –to detect errors or suspicious properties. Depending on the specification format used, this can be more or less easy; for instance, analyzing a model written in a general-purpose programming language can be quite hard, since the entities manipulated are not specific to the domain. On the other hand, a custom format can be more amenable to manipulation and analysis, thereby facilitating the verification of model properties (another argument in favor of standard formats for representing models).

The second way is to extensively test the model with unit tests and/or integration tests (preferably with both). The former means developing small tests that verify in isolation the behavior of the model components, while the latter implies running simulations with various inputs and checking their results.

According to the experience gathered in software engineering, one approach is usually not sufficient; both should be used, as they complement each other quite nicely. However, it is commonly agreed that it is better to detect errors as early as possible in the development process. For example, most library developers prefer exposing APIs that forbid wrong uses by generating compiler errors rather than failing at runtime. This way, the error is guaranteed to be detected, which is not the case when relying on additional testing, and it can be fixed rapidly.

Verification of DEVS model encompasses many aspects. Leaving aside the verification inherent to specific models, there are several generic properties that can be verified uniformly for all models. The difficulty of asserting these constraints ranges from straight-forward to very difficult. Here is a non-exhaustive list of some requirements that must be fulfilled by all CDEVS models:

- Every identifier used must refer to an existing entity (e.g. port/component names).
- Components appearing in a coupled model's couplings definition must belong to this model; similarly, in a coupling, the source (resp. destination) port must belong to the source (resp. destination) component.
- A component can receive only one event on one of its ports at a given simulation time.
- Coupled ports must have compatible types: a port that send string values should not be connected to a port expecting integer values, for instance.
- A coupled model shall not have zero-delayed loops. This includes direct-feedback loops, where an output port of a component is connected to an input port of the same component.
- The state of an atomic model shall not be modified outside its transition functions.

- The component returned by the select function must be present in the set of imminent components provided.

All these constraints can be verified either statically, by observing the model specification, or dynamically, by inserting checks into the simulation process. For example, [Rhöl and Uhrmacher 2008] provides formal definitions and XML schemas to specify model interfaces, in order to verify the correctness of compositions, while PythonDEVS contains checks to verify at runtime, during the construction of coupled models, that there is no direct feedback.

All in all, most DEVS implementations do some verification, but mainly at runtime, using ad-hoc checks. Only some constraints are sometimes enforced by the API, notably the constraints regarding identifiers. By using variables to represent entities, the languages rules ensure that they will be unique in a given scope and refer to existing objects. Not all implementations use this approach; for instance, identifiers in VLE are strings, which are verified during simulation. As a consequence, an erroneous identifier can go unnoticed for a long time before being detected, unless thorough unit testing is done.

Perhaps the least implemented feature (and, as a matter of fact, the one that initiated the development of DEVS-MS) is the detection of incompatible couplings. Indeed, most tools rely on some kind of type erasure to allow all values exchanged to be manipulated uniformly, usually by having a “universal” base class for all values. In Java implementations, this is often the Object class; all entities manipulated in DEVSJava inherit from Entity; in CD++, all values must be subclasses of Value; etc. This is needed for writing generic simulators: models have specific interfaces, but the processors must be able to handle any model, so they must use generic types. However, the processors are mainly there to make models communicate, but since they are generic, they make the models unaware of their respective interfaces. As a consequence, a wrong coupling is usually detected only when a type error is raised at runtime, when trying to downcast a value to a type which is not his.

In order to provide better error messages, James II includes some type checking before casting values to derived classes. In James II, a port is associated with a type (a Class instance). Whenever there is a request for writing a value to the port, a check is performed

to verify that the value and the port have the same type. This approach, even though it is better than no check at all, has two disadvantages:

- The check is performed at runtime, only when trying to write a value, meaning that a mistake can go unnoticed for a long time before being detected. Moreover, performing the check at each write incurs an overhead. Nevertheless, this approach is very flexible since it allows the verification to be made even if the couplings are modified during simulation, which can occur with DS-DEVS.
- This kind of implementation requires the type of the value and the one of the port to be exactly the same. This is quite restrictive, as it does not allow polymorphism (a port of type Base cannot receive a Derived instance) or conversions (for instance between numeric types (int to long)).

More evolved propositions have been made regarding port compatibility, but most of them focus on formal definitions (see e.g. [Albert et al. 2008]), not on actual implementations. In the end, these propositions often boil down to reinventing a type system dedicated to the ports of DEVS models. In DEVS-MS, we show that we can instead rely on existing type systems.

Our approach, as exposed in Chapter 6, is to leverage the C++ language template mechanism to perform both flattening and model verifications at compile-time. We achieve this by making the compiler perform simulation operations ahead of execution, effectively generating an executable specialized for a given model.

IV. Conclusion

In this chapter, we saw that DEVS is a powerful formalism, with a great potential in terms of universality. Consequently, it has received much attention from the scientific community, which has developed many tools for DEVS M&S. These tools range from simple simulation libraries to full-fledged environments including editors, real-time interaction with the simulation, results visualizers, and so on.

We pinpointed two aspects of these DEVS implementations that could be improved. The first one is performance, a feature which is rarely considered in most existing tools. For instance,

most implementations use simulation algorithms that follow directly from the abstract simulators described in [Zeigler et al. 2000], even though such algorithms incur an important message-passing overhead that can be eliminated using flattening. Flattening is used in some implementations, but there are few other optimizations applied in current DEVS simulators.

The second improvable aspect is model verification, meaning the analysis or testing of models to check whether they contain design errors. Some implementations do a bit of verifications, but rarely systematically. Usually, the verifications are performed during execution, which means that design errors can go undetected for a long time or even forever.

In Chapter 6, we will present a DEVS simulation library that tackles these issues by making the compiler perform some simulation operations and verifications over the model, thereby allowing an immediate detection of design errors and the generation of an executable optimized for the model at hand. We achieve this by using metaprogramming, more precisely a technique peculiar to the C++ language named Template Metaprogramming.

In addition to these implementation issues, we also identified the lack of interoperability between tools as one of the major hindrance for collaboration between modelers. Indeed, the multiplication of DEVS environments makes it close to impossible to share a model between teams using different implementations. We showed that several solutions were conceivable to achieve interoperability, by focusing either on simulators or on models. We think that the most promising approach consists in standardizing the representation of DEVS models, or at least in providing bridges between the different formats. To do so, we propose to use a software engineering technique called Model-Driven Engineering, which facilitate the description of models independently of any simulator/environment, and the automatic generation of representations specific to these environments, thereby obliterating interoperability issues as we will see in Chapter 5.

Before detailing our propositions, we will describe the two software engineering techniques we applied to solve the problems we just exposed, namely metaprogramming and Model-Driven Engineering. The next two chapters are dedicated to these two approaches.

Chapter 3. Model-Driven Engineering

I. Introduction

In the previous chapter, we established the fact that the DEVS software landscape is very rich. There is a profusion of DEVS environments, with various pros and cons. Having such a great offer is rather beneficial, but it also comes with the fact that interactions between scientists can be very difficult: a team can hardly share its models with another one that does not use the same tooling. Indeed, the lack of interoperability between the different software tools suppose that model sharing implies spending time and money porting the models to a new target environment. This difficulty for collaborating with DEVS is one of the major hindrances in the current M&S community, and calls for a collaborative effort to propose approaches for making different environments interoperable and compatible. In this chapter, we introduce a software engineering approach that can be leveraged to tackle this issue, namely Model-Driven Engineering (MDE).

Over the last decade, the development and the maintenance of software improved thanks to the emergence and the constantly growing adoption of a new software engineering approach called MDE [Schmidt 2006] [Bézivin 2006]. The main idea of MDE is to represent software artifacts through well-defined models, and more importantly to make these models

productive. Indeed, software engineers have been using models for a very long time, but these models were often only contemplative: they had a documenting value, but were disconnected from the executable end products.

MDE is an integrative approach that draws inspiration from tried and tested techniques from multiple technical domains to define a unified framework for the development of computer-related artifacts through models, metamodels and transformations. We will start this chapter by explaining the major concepts underlying MDE, from an integrative/generic point of view. Then, we will present the main MDE projects of these last years that focus on software engineering, coming from both academy and industry. Finally, the MDE framework we retained for our work, namely the Eclipse Modeling Project, will be described in more details.

II. Concepts and definitions

To understand MDE, one firstly needs to understand its specific terminology. Hereafter, we will provide definitions for the most important terms and concepts, along with examples and explanations of their role in MDE. A more in-depth presentation of MDE concepts can be found in the excellent book by Favre, Estublier and Blay-Fornarino [Favre et al. 2006].

II.1. What is a model

For thousands of years, mankind have used simplified representations of physical (and later, immaterial) systems to better understand, reason or communicate about them. For instance, prehistoric men used to model mammoths and horses with paintings on their caves' walls; in ancient Egypt, pyramid builders used plans to represent the soon-to-be erected stone buildings[2]; scientists of all ages have relied on drawings, mock-ups, and later on equations to represent numerous aspects of the world.

Most definitions agree on a given set of characteristics that define a model. First of all, a model is a *representation* of a system under study. A model is always associated with a given

[2] For an humoristic but relevant analysis of MDE and its relation to ancient Egypt, see the series "From Ancient Egypt to Model-Driven Engineering" [Favre 2004a,b]

system, which can be physical or not, existing or not, etc. This relation is important, as the sole purpose of a model is to provide information about the represented system.

A second characteristic of models is that they are developed for *a certain purpose*. Indeed, the aim of a model is to obtain a certain kind of information about a system. Depending on the nature of this information, different models can be needed. This property is nicely encompassed by the following definition, proposed by Minsky [Minsky 1965]:

> *"To an observer B, an object A* is a model of an object A to the extent that B can use A* to answer questions that interest him about A."*

However, this definition overlooks a common characteristic of models. If we take Minsky's definition too literally, a system can be considered a model of itself, which goes against the common use of the term. Indeed, an important part of modeling is to obtain a representation that is *simpler* than the system. The goal of modeling is to be able to answer questions about a system without having to manipulate the system itself. If the model is as complex or more complex than the system, there is no point in having a model at all. Consequently, a more complete definition should include the fact that a model is a *simplified* representation of a system. Bézivin and Gerbé give a definition that summarizes these three characteristics [Bézivin and Gerbé 2001]:

> *"A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system."*

A corollary of this definition is that a given system is never fully described by a single model: since a model is a simplification, an abstraction, it implies a loss of knowledge about the represented system. The only way to obtain all the information associated with the system would be to manipulate the system itself, or a representation that is either as complex or more complex, which would consequently not be a model in the sense we defined it. As a consequence, a study of a single system is often performed with several models, each one being designed to answer a specific set of questions. For instance, a plane can be modeled by a scale model to be tested in wind tunnels, by a 3D graphic computer model to visualize its aspect, by a collection of blueprints used for construction.

Models are prominent in software engineering, and more largely in computer science. For example, a source code is a model of a program; a UML class diagram is a model of the classes manipulated in an application; the same application can also be modeled by a UML collaboration diagram, which provides a different kind of information, namely how the classes interact; the entities related to a given application are usually modeled by data stored in databases or XML files; and the list goes on and on.

II.2. What is a metamodel

The notion of metamodel, despite its great importance, is more recent than that of model. A metamodel is a model that describes a family of models; in other words, it is a model that describes the structure, but also the semantics, of other models. In fact, a metamodel is what gives meaning to a model. Take for instance the plane blueprints example we gave before: to interpret this model, one needs to know what does the different elements depicted in the schematics mean. This meaning can be either implicit, relying on collective unconscious to intuitively convey information, or explicit, through legends or explanatory documents specifying and documenting the syntax used in the schematics. These legends and documents compose a metamodel, describing a family of schematics (including the plane schematics at hand).

The Object Management Group (OMG) proposes the following definition [MOF 2002]:

"A meta-model is a model that defines the language for expressing a model."

This definition relates to the notions of language and grammar in language theory: like a grammar is a collection of rules defining a set of sentences (a language), a metamodel is a collection of rules defining a set of models. A model that pertains to the language defined by a metamodel is said to be *conformant to* this metamodel. For instance, a valid XML document needs to conform to some XML Schema, which is its metamodel.

An important thing to note is that a metamodel is itself a model; as a consequence, it can also conforms to its own metamodel (which is therefore a metametamodel), and it can be manipulated like any other model. In fact, the number of layers of abstraction can be arbitrary big. However, most MDE frameworks limit themselves to three modeling layers,

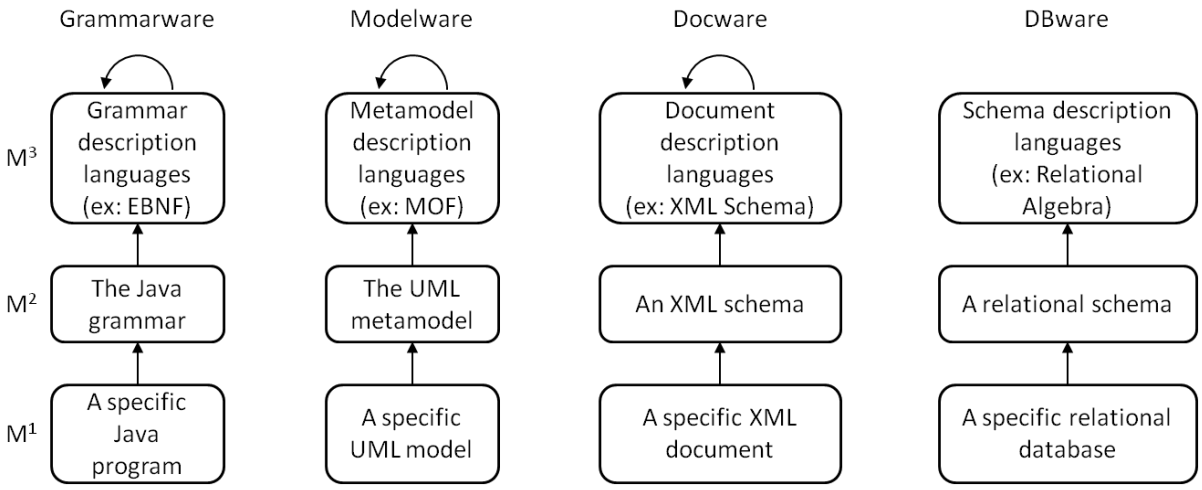


Figure 8. Extract from [Favre et al. 2006]: metamodeling hierarchy in various technical spaces.

and often rely on a self-descriptive top-level metamodel, which conforms to itself (metacircularity). This three-layer approach has been successfully used in many technical domains, long before the emergence of MDE: Figure 8, taken from [Favre et al. 2006], shows how this three-layer hierarchy has been independently used in different technical spaces.

In the domain of grammars and languages, a commonly used top metametamodel is the Extended Backus-Naur Form (EBNF), a grammar conforming to itself. Similarly, in software modeling, the MetaObject Facility (MOF) of the OMG, described later in this chapter, is often used as a common metametamodel and is itself defined in MOF. An XML document is validated against some XML schema, which is validated against XML Schema, the XML schema of XML schemas. Databases are described by relational schemas, written according to relational algebra[3].

The main idea in MDE is to make metamodels explicit and well-defined, in order to make them amenable to computer processing. By precisely defining metamodels, it becomes possible to develop generic tools for manipulating any model conforming to the metamodel. This tooling is what makes models productive instead of being just contemplative: in MDE, models are processed, analyzed, transformed, etc.

[3] Note that in this case, the top metametamodel is not self-descriptive: relational algebra is not defined in relational algebra.

II.3. What is a transformation

Operations performed on models are called transformations in the MDE terminology. A transformation is a process that generates one or several target models from one or several source models. This process can be either manual or automated; even though MDE consider both cases as transformations, the latter kind is obviously the most interesting one, and the automated nature of transformations is often implied. Transformations are defined for metamodels, making them applicable to any models conforming to these metamodels.

Ideally, both source and target models should conform to well-defined metamodels. However, there is often a need to generate artifacts without an explicit metamodel, such as text files with no particular format (e.g. documentation) or numeric results (e.g. metrics about a model). Therefore, some MDE implementations make a distinction between model-to-model and model-to-text transformations.

Transformations come in different flavors: they can be simple one-way mapping between a model and another, be reversible, involve several source models (possibly conforming to different metamodels), generate several target models, etc. A transformation is said to be endogenous (resp. exogenous) if the source metamodel and the target metamodel are the same (resp. different). In-place transformations are a special case of endogenous transformations where the source and target model are one and only model: instead of generating a new model, the transformation modify the model itself. A more complete taxonomy of transformations can be found in [Diaw et al. 2010].

Like models and metamodels, transformations are not an innovation of MDE. They are only a formalization of practices that are ubiquitous in software engineering. For instance, the conversion of a spreadsheet file to a comma-separated values (CSV) file is an exogenous one-to-one transformation; linking is a transformation from a set of objects files (models) written in an object file format (metamodel), such as the Common Object File Format (COFF) or the Executable and Linkable Format (ELF), to a single executable, usually in the same format; the generation of HTML documentation from annotated source code, by Doxygen or Javadoc for instance, is another example of transformation that was done way before MDE existed.

The goal of MDE is to provide a consistent methodology and a set of tools for facilitating the writing of transformations that used to be developed in an ad hoc manner. Assuming a common metametamodel, all metamodels can be manipulated in the same way. As a consequence, all transformations can be written with the same paradigms and technologies, with tools specialized for the task.

II.4. What is Model-Driven Engineering

MDE is an attempt to provide a unified conceptual framework to reason and communicate about models, metamodels and transformations. The goal is to bring together the experience gathered in different technical domains (languages, databases, software modeling, etc.) to construct a foundational theory.

This new paradigm is in line with the previous evolutions of software engineering, which always boiled down to increasing the level of abstraction at which the developer works: we went from non-structured programming to procedural programming, then to object-oriented analysis and design, and more recently to component-based development. Each of these paradigm shifts opened the door to new software developments of ever-increasing complexity, by protecting developers from this complexity and making software more comprehensible and maintainable.

MDE aims at increasing once again the level of abstraction by having the developer manipulate very high-level artifacts (models), hiding the underlying complexity thanks to frameworks and automated transformations. To make software more manageable, MDE proposes a strict separation of concerns: different aspects of an application should be described by distinct models, which would later be woven in a result artifact (usually an executable). This approach, similar to the one applied in Aspect Oriented Programming (AOP) [Kiczales et al. 1997], allows each aspect to be centralized in a single place instead of being scattered all over the project, facilitating its development but most of all its maintenance. Moreover, thanks to the extensive use of metamodels and transformations, the developer can specify the different models representing an application with specific languages instead of using a generic modeling language.

MDE have been made a reality in several projects. In fact, many projects even preceded MDE, which tries to unify existing approaches *a posteriori*. The first implementations of MDE date back to the late 1980s, when Computer-Aided Software Engineering (CASE) tools were supposed to revolutionize software development. These tools allowed developers to model applications through one or several graphical models, and generated the corresponding source code. A precursor in this domain was Fabrik [Ingalls et al. 1988], an integrated development environment written in Smalltalk that provided a kit of components that could be wired together graphically to build new components and applications. However, these tools did not meet the expected success, for two main reasons. Firstly, there was no true standard modeling language at that time. Consequently, every tool had its own modeling language, which implied a great learning effort from the developers. Secondly, these tools used to generate a massive amount of code, usually in a General-Purpose Programming Language (referred to as GPPL in this thesis). In addition to being hard to comprehend, this code was hardly modifiable, at the risk of losing the synchronization with the source model(s), since the complexity of the generated code made it hard to reverse-engineer. Because of this, the development process was limited to a top-down approach that quickly showed its limits compared to more agile methods.

After its creation in 1997 by the UML Partners, and its subsequent adoption by the OMG, the Unified Modeling Language (UML) quickly became a *de facto* standard in industry. As a consequence, many CASE tools started using UML as a modeling language, and still do today (e.g. Rational Software Architect, Bouml, Poseidon, etc.). However, the use of a general-purpose modeling language such as UML has some drawbacks. First of all, even though the support for round-trip engineering improved over the years, it is still far from being perfect (mainly because there is no one-to-one mapping between UML and code). More importantly, UML is probably not the best modeling language for a majority of projects. Indeed, UML is very generic; it was designed to support the modeling of any object-oriented software. Because it is general-purpose, the benefit in terms of abstraction, when compared to GPPL such as Java or C#, is not that important. In fact, it is often faster and easier to write a piece code rather than creating the corresponding UML models. Of course, UML diagrams provide a better visualization of an application, and do a rather good job at separating

concerns, but this gain is not always worth the effort, weighted against the burden of maintaining two separate representations of the application.

This is the reason why more and more companies turn to domain-specific modeling. The actual trend is to transition from developing software with general-purpose languages (programming languages or modeling languages) to development with Domain-Specific Languages (DSL). A DSL is a language, either textual or graphical, that is dedicated to a particular domain; it uses a vocabulary and constructs that are very close to the domain, so that a practitioner can very easily write and understand specifications in this language. Unlike general-purpose languages, a DSL targets a narrow set of problems, but allows the resolution of these problems much more efficiently. In the domain of M&S, many environments have provided graphical DSLs for modeling various systems, along with code generation features to ease their simulation [Hill 1993] [Hill and Gourgand 1993] [Hill 1996]. There are numerous examples in computing: the Structured Query Language (SQL), for manipulating data in relational databases; Cascading Style Sheets (CSS), for describing the look and formatting of documents; regular expressions, to specify lexers; etc. DSLs can also be used to allow non-computer experts to interact with or customize their applications.

However, for a long time, developing a DSL and the surrounding tools (editors, parsers, generators, etc.) was a very costly process that did not always provide a satisfactory return on investments. This situation has changed with the emergence of MDE environments focusing on metamodeling rather than just modeling. Indeed, more and more tools that facilitate the development of metamodels and tooling are available. With these environments, designing a DSL along with textual and graphical editors, model transformations and code generators becomes a much easier task. Therefore, we observe the multiplication of environments (often in the form of Integrated Development Environment (IDE) plug-ins) specialized for specific domains. Considering this, it becomes necessary to make a distinction between the environment developer (the “toolsmith”) and the user of this environment (the “practitioner”). The toolsmith is responsible for creating metamodels for the domain at hand, as well as different transformations to allow the generation of useful assets from domain models. He can also create a full-fledged environment for domain-specific modeling, usually with the help of the metamodeling environment. The practitioner then leverages this environment to easily model his domain

entities, and uses the various transformations to automatically generate artifacts such as executables, documentation, diagrams, specifications conforming to some other domain-specific metamodel, etc.

This ability to easily create tool suites targeted to a particular domain is probably the most important contribution of MDE. MDE frameworks focusing on metamodeling can be thought of as meta-frameworks, meaning frameworks designed to create frameworks. By lowering the cost of creating domain-specific tools, MDE allows the multiplication of specialized environments for various domains, thereby drastically improving the process of developing applications in these domains.

In the following section, we will present some of the major MDE implementations.

III. Survey of Model-Driven Engineering implementations

The MDE approach has a concrete expression in several projects, coming mainly from industry. Indeed, even though academic researchers work on theorizing MDE and on developing innovative technologies for it, MDE is very much set in industry, and is sustained by important groups such as IBM, Microsoft or the Object Management Group (OMG). Hereafter, we present the most tried and tested MDE initiatives.

III.1. Model-Driven Architecture

The Model-Driven Architecture (MDA) [Kleppe et al. 2003], adopted by the OMG in 2001, gave a great boost to the domain of Model-Driven Engineering. MDA is a methodology based on a collection of standards that focuses on separating the business concerns from the platforms/technologies on which applications will be deployed.

The MDA usually relies on a four layered metamodeling stack[4], as depicted in Figure 9 [MOF 2002]. The top layer (M3) is occupied by the universal metametamodel of MDA, the Meta Object Facility (MOF). Below, the M2 layer contains the metamodels conforming to MOF, notably the UML metamodel, which is ubiquitous in MDA. These metamodels are

[4] In fact, MDA does not really restrict the number of layers; to avoid confusion, later versions of the specification do not include this figure.

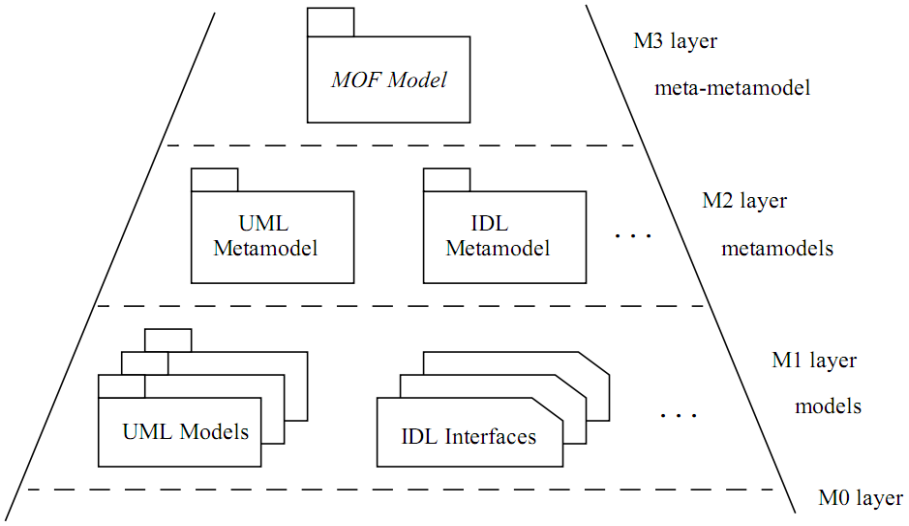


Figure 9. Extract from [MOF specification 1.4 2002]: Four layer modeling in MDA.

instantiated in user-defined (or generated) models, which represents entities of the M0 layer, usually software artifacts.

The major concept in MDA is the strict separation between platform-independent and platform-specific models. A software system is represented by several models that represent various viewpoints [Truyen 2006]:

- The computation independent viewpoint is at the more abstract level. It represents the application viewed by the domain experts, who can express their requirements with a vocabulary very close to them, without worrying about the implementation choices.
- The platform-independent viewpoint is more detailed and begins to describe how the application is or will be implemented. However, the representation of the system is still abstract in the sense that it leaves aside all the lower-level details, which depend on the platform(s) where the application is deployed.
- The platform-specific viewpoint fully describes the application for a given platform. A platform is a set of subsystems and technologies that serve as foundations for developing, deploying and running applications. Examples of platforms are CORBA, J2EE, databases, web services, etc. The platform-specific viewpoint is a refinement of the platform-independent one that provides enough details to generate the application for the specific platform.

All these viewpoints are captured in models: Computation Independent Models (CIM), Platform Independent Models (PIM) and Platform Specific Models (PSM). These models conform to some metamodel, usually UML, and can be manipulated through transformations.

The transformations involved in MDA are mainly refinement transformations, meaning transformations that increase the level of details contained in a model (i.e. lower its abstraction level). The goal is of course to automate most of these transformations, by using models of the mapping between PIMs and PSMs. A common practice in MDA is to generate a PSM from a PIM simply by adding annotation (marks) to the original UML model. These annotations, defined in UML profiles, define how the abstract entities specified by the PIM must be mapped to the target platform. Afterwards, the annotated model can be automatically processed to generate the target artifacts (executables, configuration files, deployment descriptors, etc.) in the target platform.

To specify the transformations, the OMG adopted a standard set of languages called Query/View/Transformation (QVT), which contains three different transformation languages:

1. Relations, a declarative language supporting complex object pattern matching and implicitly generating traces of the transformation;
2. Core, a pruned version of Relations, equally powerful but less user-friendly;
3. Operational Mappings (QVTO), an imperative language allowing a procedural style more familiar to most developers.

III.2. Eclipse Modeling Project

The Eclipse Modeling Project (EMP) is an Eclipse Top Level Project that serves as an umbrella project for a collection of more focused sub-projects targeting different aspects of MDE [Gronback 2009]. It aims at providing an exhaustive set of industry-ready MDE tools for the Eclipse platform, as well as hosting innovative and exploratory projects.

The Eclipse platform is a software development environment, first created by IBM, which is now a free and open-source project leaded by the Eclipse Foundation. Initially dedicated to

Java development, the Eclipse platform evolved into a multi-language, highly-extensible framework, thanks to a powerful plug-in system. This plug-in mechanism, along with the integration of MDE concepts in the platform, transformed the Eclipse platform into a full-fledged language workbench [Fowler 2005], meaning an environment for defining languages and tools for these languages.

The EMP is divided in three main topics: abstract syntax development, concrete syntax development, and model transformation. Abstract syntax development projects deal with the creation, edition, querying and validation of metamodels. The most important project in this regard is the Eclipse Modeling Framework (EMF), which provides a core for all other modeling projects. Concrete syntax development projects focus on providing tools for defining graphical and textual syntaxes for metamodels, as well as generating editors corresponding to these syntaxes. Finally, model transformation projects concern code generation, or more generally text generation (Model-to-Text (M2T) transformations), and generation of models from models (Model-to-Model (M2M) transformations).

The scope of EMP also includes implementations of industry standards, notably from the OMG. For instance, EMP contains implementations of MOF, UML, QVT and XML Schema Definition. However, the support of these standards only represents a part of EMP, and most projects do not comply with any industrial norm. For that matter, it is interesting to note that many projects represent the cutting edge of MDE research, and explore less studied aspects such as megamodel management, model weaving, reverse-engineering, etc.

To sum up, EMP is a set of Eclipse-based projects covering most of the MDE aspects. These projects provide many powerful tools to create metamodels, with both abstract and concrete syntaxes, query them, validate them, transform them, etc. Some tools also allow generating Eclipse plug-ins in order to automatically create custom environments for domain-specific modeling. The project is free and open-source, and is supported by an important community.

III.3. Software Factories

The notion of Software Factories, envisioned by Microsoft, stems from an analogy between the software industry and more classical goods-producing industries [Greenfield and Short

2004]. The idea underlying software factories is the industrialization of software engineering: when faced with an ever-increasing demand, other industries had to transition from relying on people craftsmanship to relying on normalized and partially automated processes. To increase in scale while reducing costs, the production of goods stopped being a prerogative of craftsmen to become an industrialized process, with products becoming a simple assembling of normalized components flowing through supply chains.

The production of merchandise became a sequence of specialized steps, performed by operators and machines specialized for an operation, where some raw materials and/or components coming from the upstream supply chain are transformed and/or assembled into a final or intermediary product, ready for distribution or further transformation. This specialization made possible the automation of parts of the process, but also allowed the apparition of product lines, meaning families of similar products differing in just a few respects (size, color, nature of some components, etc.). These product lines are a good way to answer the ever-increasing demand of customized products, without making production costs and times increase exorbitantly: all products of a family can be produced in a similar way, with a few variation points in the process allowing the customization.

The idea behind software factories is to apply the same evolution to software development. As of now, most software products are created using generic processes and tools (notably generic programming languages). Object-oriented and component-based programming improved the situation by allowing the developers to think about applications at higher levels of abstraction, to reuse existing software, and to facilitate comprehension and evolution of applications. However, even though the reuse of provided components eases development, the tools available to transform or assemble these components is still quite limited.

In this regard, much work has been done in the context of Service-Oriented Architecture (SOA), especially for web services, to automate or at least facilitate the orchestration and choreography of services, notably through standard DSLs (Business Process Execution Language (BPEL), Web Service Choreography Interface (WSCI)) and tools supporting these languages (workflow editors, BPEL engines, etc.). This approach of defining an environment

specialized for a family of products, in this case applications composed of interacting web services, is exactly what software factories are about.

In substance, a software factory is an environment allowing mass production of software. More concretely, such factories are usually composed of several elements: libraries or frameworks, which provide a common platform for a family of software products; DSLs, to simplify the specification of the variation points, possibly accompanied by specialized editors; generators, to handle all the boilerplate needed to map the developer's need to the underlying framework/library; and more generally, the factory can contain any tool that could facilitate the development of an application, such as skeletons, wizards, etc. Greenfield and Short summarize this with the following definition [Greenfield and Short 2004]:

> *"A software factory is a software product line that configures extensible tools, processes, and content using a software factory template based on a software factory schema to automate the development and maintenance of variants of an archetypical product by adapting, assembling, and configuring framework-based components."*

The Microsoft team working on software factories came up with a more pragmatic definition:

> *"A software factory is a structured collection of related software assets. When a software factory is installed in a development environment, it helps architects and developers predictably and efficiently create high-quality instances of specific types of applications."*

This approach is implemented in Microsoft's IDE Visual Studio: when creating a new project, the developer can choose between a wide variety of project types. Depending on the template selected, he is provided with several assets such as configuration files, code skeletons, deployment descriptors, etc. In addition, the environment is configured with a collection of libraries corresponding to the type of application targeted, as well as specific DSLs and editors along with the corresponding code generators. For instance, when creating a Windows Presentation Foundation (WPF) application, the environment creates a template

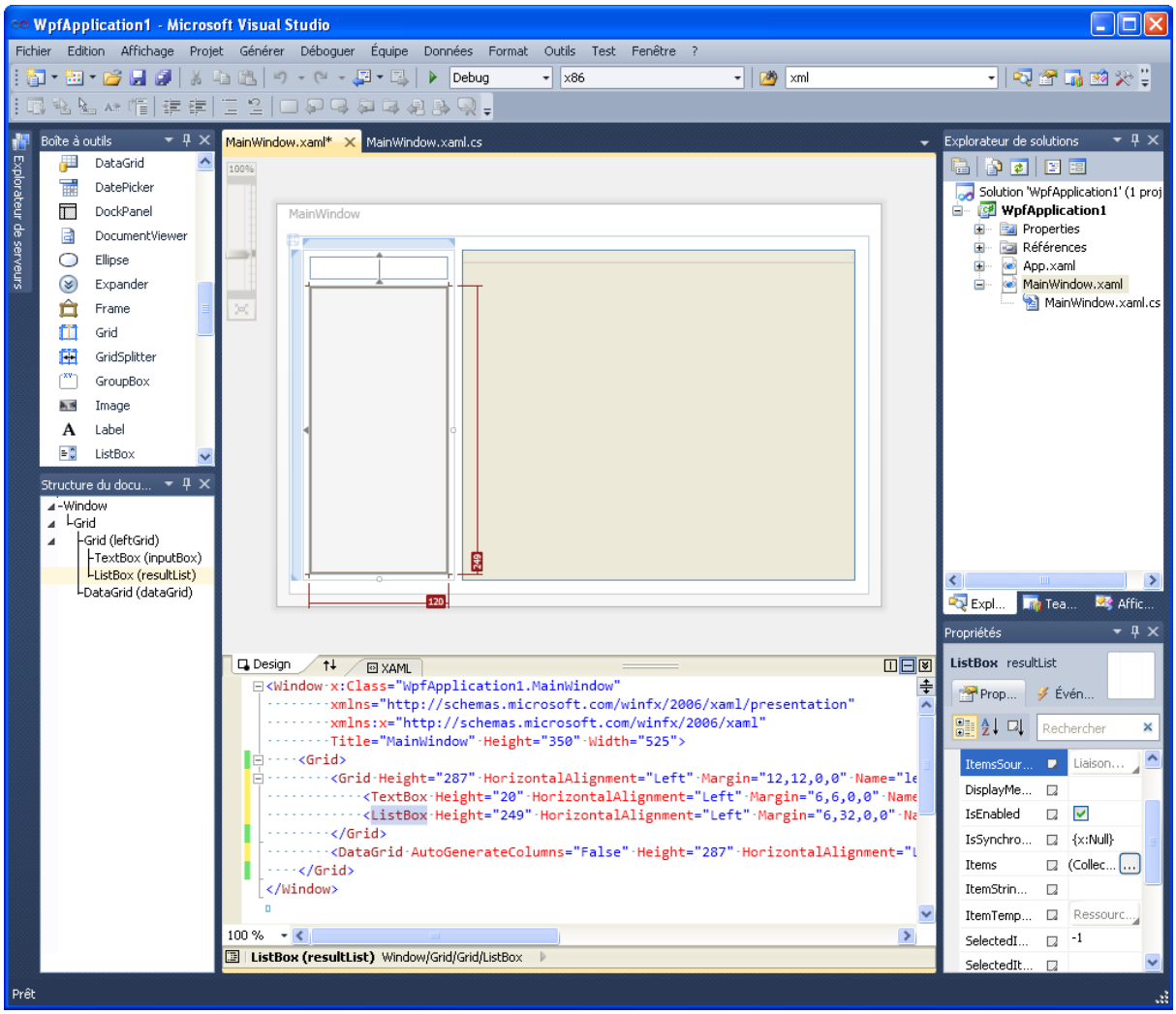


Figure 10. Development of rich desktop applications with Visual Studio 2010 WPF software factory.

application with a skeleton of a main window, which can be customized using Microsoft's DSL for user interface definition, the eXtensible Application Markup Language (XAML). This language uses a vocabulary specific to the UI design domain, and can be used either textually or through a graphical editor with drag-and-drop features. The XAML specification is automatically transformed into C# (or VB.Net) code on top of the WPF libraries. Visual Studio becomes a software factory for creating rich desktop applications, as can be seen in Figure 10.

Out-of-the-box, Visual Studio provides several software factories for creating console applications, class libraries, Windows services, web sites, web services, Microsoft Office add-ins, games, etc. Of course, these factories do not encompass all the possible software

families. Consequently, the environment gives the possibility of creating custom software factories, in the form of a "meta-software-factory": when faced with the need to create many similar applications, a developer can create his own factory to speed up and facilitate the development of these applications.

In a nutshell, a software factory is a specialized environment dedicated to the development of a particular type of applications. By capturing the common aspects of a software family in an application template, a developer can customize an environment in order to limit the development of new applications to the specification of a few points of variation, possibly with a collection of tools facilitating this specification. This approach is quite similar to the language workbench notion we presented previously, but is more focused on predefined or generated assets, and less on DSLs.

III.4. Other Modeling Driven Engineering initiatives

The world of Model-Driven Engineering is not limited to the three approaches we just described. There are many projects and tools that implicitly or explicitly adopt MDE principles, both in industry and in academy.

The greatest offer concerns UML tools: dozens of applications are available for creating UML diagrams and generating code in various programming languages, or reverse-engineer diagrams from code. Some of them are free and open-source (Bouml[5] (before version 5.0), ArgoUML[6], Umbrello[7], …), others are commercial tools (IBM Rational Software Architect[8], Poseidon for UML[9], MagicDraw UML[10], Objecteering[11], ...). All these tools provide a common set of features, such as diagrams creation/edition, code generation, serialization in a standard format (the XML Metadata Interchange (XMI)), etc. However, most of them are limited to these features, making them very limited implementations of MDE concepts. The lack of metamodeling features makes the creation of domain-specific modeling languages impossible with these tools, going against the MDE approach of using several specialized

[5] http://www.bouml.fr/ (last accessed 26/06/2012)
[6] http://argouml.tigris.org/ (last accessed 26/06/2012)
[7] http://uml.sourceforge.net/ (last accessed 26/06/2012)
[8] www.ibm.com/software/awdtools/swarchitect/ (last accessed 26/06/2012)
[9] http://www.gentleware.com/ (last accessed 26/06/2012)
[10] http://www.nomagic.com/products/magicdraw.html (last accessed 26/06/2012)
[11] http://www.objecteering.com/ (last accessed 26/06/2012)

models for representing applications. There are however some exceptions, notably tools that implement the MDA, such as Objecteering, or tools that provide advanced customization capabilities, such as Papyrus[12].

To obtain more advanced MDE features, one can turn to language workbenches. We already presented Eclipse and Visual Studio, two IDEs that can both be used as a platform for specialized editors, but other alternatives are available. For instance, JetBrains proposes a full-fledged environment for language oriented programming, named MetaProgramming System (MPS)[13]. MPS is an open-source product that facilitates the creation of textual DSLs, and more importantly the creation of specialized editors for this DSL. The strength of MPS lies in its use of projection editors: instead of working with textual files like most editors, a projection editor uses an internal abstract representation of the "source" (akin to a model in other MDE environments), allowing several views to manipulate it without going out of sync. In addition to the synchronization aspect, using an internal representation allows the editing views to provide much more information to the user, greatly enhancing the development experience.

This approach is also used in another language workbench developed by Intentional Software, where an internal model is mapped to several views through small model transformations, allowing the domain knowledge to be more easily expressed. However, their tool, the Intentional Domain Workbench[14], is not readily available at the moment.

Apart from these commercial products, metamodeling environments are also actively studied by the scientific community. One of the pioneers in this regard is MetaEdit+, created in the early 90s as part of the MetaPHOR project [Smolander et al. 1991]. This tool, which later became commercial, focuses on the creation of graphical domains specific languages and includes features for developing code generators.

Along the same lines, AToM^3, developed at the Modelling, Simulation and Design Lab (MSDL), also provides a graphical environment for defining domain-specific metamodels along with a graphical concrete syntax [De Lara and Vangheluwe 2002]. Unlike MetaEdit+,

[12] http://www.papyrusuml.org/ (last accessed 26/06/2012)
[13] http://www.jetbrains.com/mps/ (last accessed 26/06/2012)
[14] http://intentsoft.com/ (last accessed 26/06/2012)

AToM^3 has a strong focus on model transformations, relying on graph rewriting and graph grammars to allow transformations between heterogeneous metamodels. Another interesting aspect of AToM^3 is that it is developed to bring MDE techniques to the M&S domain; even though these two domains are closely related (after all, they both deal with models, metamodels/formalisms, transformations, analysis, etc.), few works have been done to integrate them. The stated goal of AToM^3 is to facilitate multi-formalism modeling and simulation by relying on MDE techniques such as metamodeling and model transformations.

In addition to these full-fledged environments, the scientific community also works on more specific MDE items. Several metamodeling languages are available, notably the Kernel Meta Meta Model (KM3) [Jouault and Bézivin 2006] and Kermeta [Muller et al. 2005]. KM3 is a language with a textual concrete syntax and an abstract syntax which is similar to other existing metametamodels, such as the MOF, but much simpler. Its strength lies both in its simplicity and its compatibility with widely-used metametamodels like Ecore or the MOF, supported through bidirectional transformations defined in the ATLAS Transformation Language (ATL) [Bézivin et al. 2003]. KM3 is used as the default language in a rich repository of metamodels called Atlantic Zoo[15], which hosts numerous metamodels developed by the community. Automatically-generated mirrors are available to obtain the metamodels expressed in different metametamodels.

Kermeta also provides a textual concrete syntax for defining metamodels, but uses Essential MOF as the underlying metametamodel. However, Kermeta proposes to go beyond the usual data-only metamodeling and to also include operations in the metamodel. Concretely, this means that a Kermeta metamodel can define constraints, transformations, in fact a whole behavior associated with the metamodel. Basically, Kermeta aims at bringing encapsulation to the metamodeling domain, by making it possible to bundle meta-data and meta-operations in a single entity.

This approach is not common, and operations on metamodels are usually well-separated from the metamodel definition. Most notably, model transformations are usually specified using a specific language, different from the metamodeling language. We already presented QVT, the transformation language standardized by the OMG, but other alternatives are

[15] http://www.emn.fr/z-info/atlanmod/index.php/Zoos (last accessed 26/06/2012)

available. The most famous one is probably the ATLAS Transformation Language (ATL) [Bézivin et al. 2003], hosted by the Eclipse Modeling Project, which operates on KM3 metamodels. Even though ATL differs from QVT on a number of aspects, the two languages can be approximately aligned, allowing transformations in one language to be automatically transformed into transformations in the other. A great number of ATL transformations defined by the community are hosted on the ATL Transformations Zoo[16].

Finally, the scientific community also works on more prospective aspects of MDE. For example, the global management of MDE artifacts, i.e. handling the numerous models, metamodels, transformations and generated assets involved in an MDE project, is tackled by *megamodeling* [Bézivin et al. 2004]. Another example is *metamodel comparison*, which focuses on automatically determine the "diff" of two metamodels, for instance to automate the migration of models from one version of a metamodel to another. Many interesting works can be found on the ATLAS Model Weaver (AMW) project page[17].

After this quick overview of different implementations of the Model-Driven Engineering approaches, we will proceed to a more detailed description of the MDE framework we chose for our work, namely the Eclipse Modeling Project.

IV. Eclipse Modeling Project

We just saw that the offer in terms of MDE environments is quite rich, albeit not profuse. Among these propositions, we retained the Eclipse Modeling Project for the following reasons. First of all, it is a free and open-source project, supported by a huge community. It supports only parts of MDA, but MDA is a very restricted approach to MDE anyway: it has a (too) strong focus on UML and the notion of platform-independence/platform-specificity. On the other hand, the EMP adopts a more open approach and emphasizes metamodeling and transformations in a broader sense than MDA.

Regarding language workbenches, the alternatives to Eclipse such as Visual Studio or Intentional Domain Workbench have the major drawback of being commercial, often with expensive licences. MPS is free and open-source, but targets only a very specific aspect of

[16] http://www.eclipse.org/m2m/atl/atlTransformations/ (last accessed 26/06/2012)
[17] http://www.eclipse.org/gmt/amw/ (last accessed 26/06/2012)

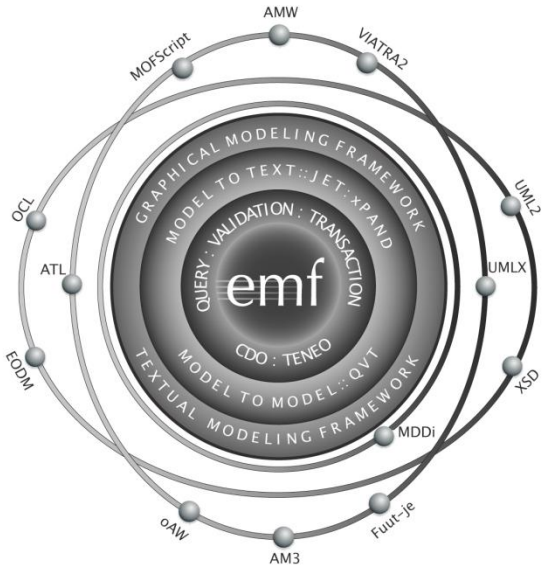


Figure 11. EMP logo representing the various subprojects [EMP logo 2006].

MDE, namely language oriented programming. Thus, even though it excels in the domain of textual DSLs, it lacks many features when compared to more generic environments like the one provided by EMP.

Finally, environments provided by the scientific community could have been an interesting alternative, but often lack support and are rarely widely used, making their adoption more risky. Moreover, many scientific teams rely on Eclipse as the underlying platform for their project, making it possible to use their tools in conjunction with those provided by EMP.

As a consequence, we chose the Eclipse platform and the Eclipse Modeling Project as the MDE environment to support our works on a model-driven DEVS environment. We will therefore present in more detail the various EMP components, depicted in Figure 11, a preliminary logo of the project.

These components are useful in different phases of an MDE project. The usual workflow for such a project is as follows:

1. Define the appropriate metamodels. The core of all MDE project is composed of metamodels that describe the abstract syntax of the elements of the domain at hand. To define these metamodels, EMP relies on the Eclipse Modeling Framework (EMF), in particular on a universal metametamodel called Ecore.

2. (Optionally) define concrete syntaxes for these metamodels. To facilitate the creation and edition of models, it is often a good idea to develop textual or graphical syntaxes for the metamodels. This way, a practitioner can develop models using high-level languages instead of manipulating them directly in their serialized form or through generic editors. EMP contains two independent projects for defining concrete syntaxes: the Graphical Modeling Framework (GMF) for graphical syntaxes, and the Textual Modeling Framework (TMF) for textual syntaxes.
3. (Optionally) define model transformations over the metamodels. An important aspect of MDE is the ability to easily manipulate model that conforms to a well-specified metamodel. An MDE project often contains several model transformations that produce models from others (Model-to-Model (M2M) transformation) or that generate text from models, often code for a specific platform (Model-to-Text (M2T) transformation).

The following sections describe each of these phases in more detail, with an emphasis on the tools provided by EMP to conduct each of them.

IV.1. Abstract syntax development

The core of EMP is EMF [Steinberg et al. 2008], one of the oldest Eclipse projects. EMF is a modeling framework that provides several facilities to create and manipulate models and metamodels. From a metamodel specification, it generates the corresponding Java classes along with tools for viewing and editing models conforming to it.

The pivot of EMF is its metametamodel Ecore, which is approximately an implementation of Essential MOF, a subset of MOF containing only the fundamental elements. Ecore is pretty simple, as Figure 12 shows. Basically, an Ecore metamodel is a collection of classifiers, organized in packages, which contains attributes and references to each other, as well as parameterized operations. Elements can also be annotated, for instance to constrain them or specify their body.

The creation and edition of an Ecore metamodel can be performed in several ways. Out-of-the-box, EMF provides a tree-like editor to add, remove or move model elements and modify their properties. It is also possible to import a metamodel from a set of annotated

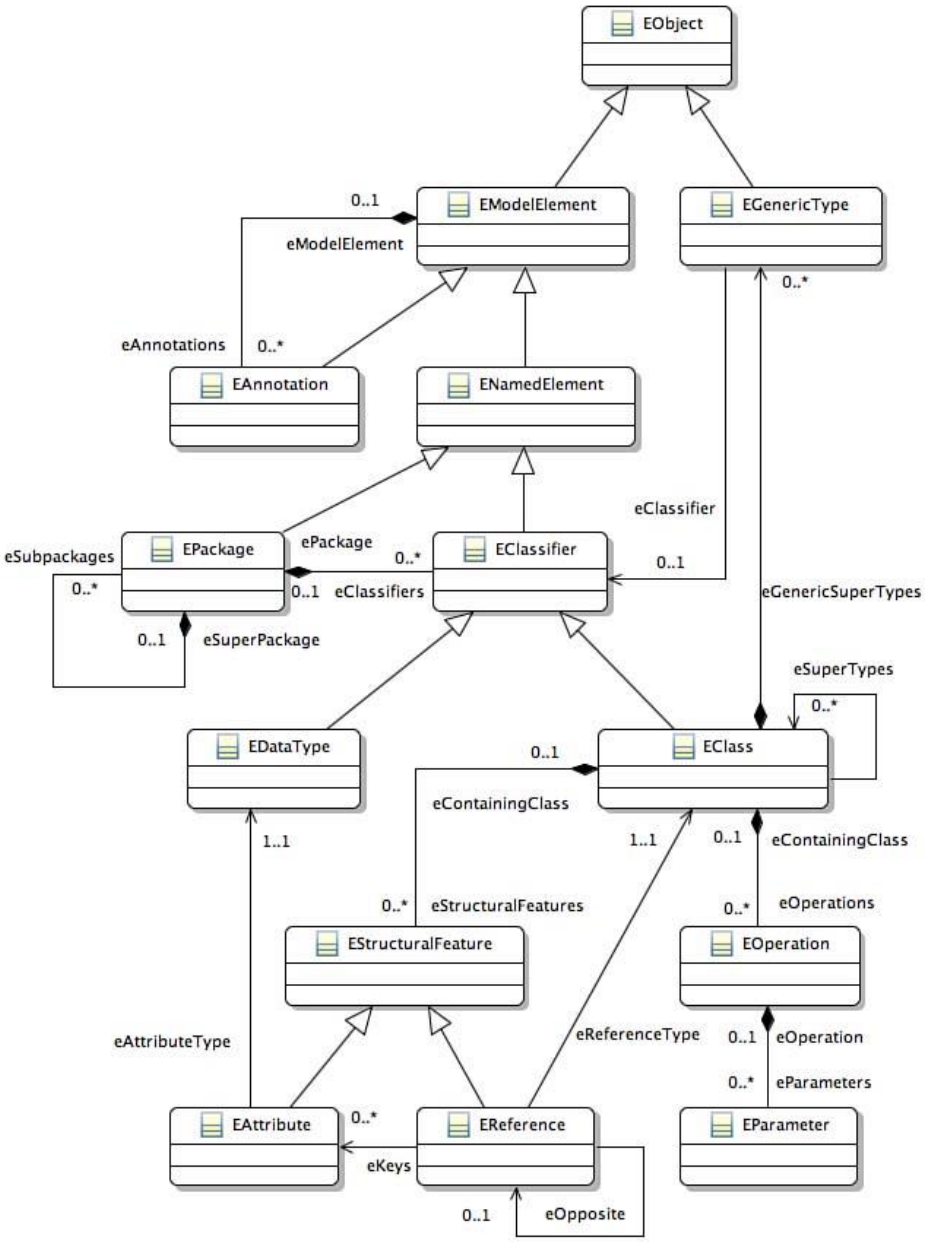


Figure 12. Extract from [Gronback 2009]: Ecore model.

Java class. Finally, it is also possible to edit directly the serialized version of the model, which uses the XMI format. In addition to these possibilities, components can be installed to allow the importation of a metamodel from an XML Schema Definition or from a UML2 model. Other projects also provide a graphical and a textual concrete syntax for Ecore.

Given an Ecore model, EMF generates a set of Java classes allowing instance models (models conforming to the metamodel) to be represented and manipulated in memory. In addition, EMF also provides a generic model editor, either using a reflective API or through generation

of an Eclipse plug-in. This generic editor is similar to the one used by default to create Ecore models (tree structure and properties view), for a good reason: since the Ecore metamodel is itself an Ecore model, the supporting tooling is defined using the same tools as other Ecore models (bootstrapping). The code generated by EMF can be customized with a generator model, a lower-level model complementing the Ecore model with implementation details.

In addition to these core components, the EMF project includes many subprojects providing various features:

- Model Query 2: Java API and SQL-like language for performing queries over EMF models, in order to retrieve specific elements;
- Model Transaction: transactions for model manipulation, allowing multiple threads to edit the model, undo/redo, checking of model integrity, etc.;
- Validation Framework: definition of constraints over models, allowing some semantics to be added to them; the constraints can be checked in batch or live mode, and can be defined either in Java or in Object Constraint Language (OCL) [Warmer and Kleppe 2003];
- Connected Data Objects: model repository;
- Compare: model comparison and merging
- Teneo: model persistence in databases, based on persistence frameworks such as Hibernate or EclipseLink)

Most importantly, EMF is the foundation for all other EMP projects, which notably rely on Ecore as a central pivot to enable interoperation between the different tools.

IV.2. Concrete syntax development

Like we said before, creating a model conforming to some metamodel can be done with the EMF-generated editor. However, this editor is quite basic, and is not very user-friendly: the edition of models is tedious, the overall structure is hard to grasp, and few contextual help is provided. As a consequence, two projects were initiated to tackle the issue of concrete syntax development, both graphical (Graphical Modeling Framework (GMF)) and textual (Textual Modeling Framework (TMF)).

IV.2.1. Graphical syntax development

GMF is built upon EMF and the Graphical Editing Framework (GEF), a framework for building graphical editors and views for the Eclipse platform. Concretely, GMF provides a simple way to define a graphical syntax for a corresponding metamodel, and automatically generate a full-fledged editor in the form of an Eclipse pug-in. To do so, it embraces an approach very much in line with MDE and more specifically with EMP: indeed, the definition of an editor is specified through various models, which are then merged and transformed, eventually resulting in a generated artifact ready to deployment. All this is of course performed using the tools provided by EMP, the same that are available to the users, hence exhibiting another example of the bootstrapping capabilities of MDE environments.

In practice, the definition of a graphical editor with GMF is done through three models. A graphical definition model specifies the various graphical elements to be used in the editor. It is composed of a figure gallery (a collection of shapes, lines, etc.), which can be reused in multiple projects; these figures are then linked to diagram elements such as nodes, connections, labels, etc. A sample graphical definition model for a mindmap diagram is provided in Figure 13, taken from the official GMF tutorial[18].

A tooling definition model specifies the tooling to include in the editor. This model allows the toolsmith to very easily describe the tools that he wants to provide in his editor, such as palettes, toolbars, menus, etc. Figure 14 shows a sample possible tooling definition model for the mindmap diagram.

A mapping model represents the links between a domain model defined in Ecore and the graphical and tooling definitions. This model maps the graphical diagramming elements and those of the abstract syntax, as can be seen in the example provided in Figure 15. Thanks to this mapping, a diagram has a direct correspondence with an instance model conforming to the domain (meta)model.

[18] http://wiki.eclipse.org/Graphical_Modeling_Framework/Tutorial (last accessed 27/06/2012). A collection of GMF tutorials and samples can also be found at http://gmfsamples.tuxfamily.org/ (last accessed 27/06/2012).

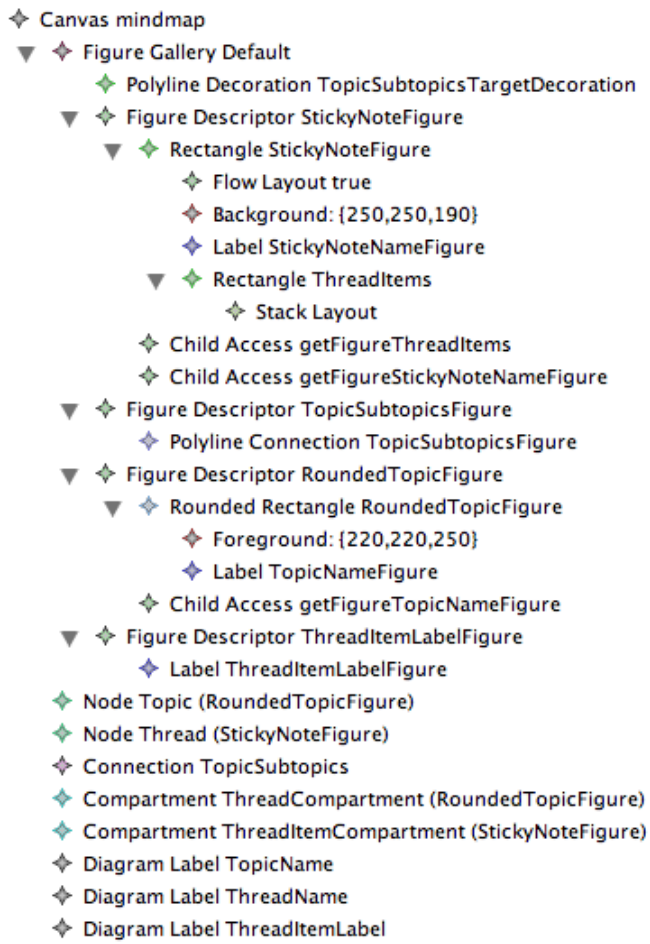


Figure 13. GMF – sample graphical definition model[18].

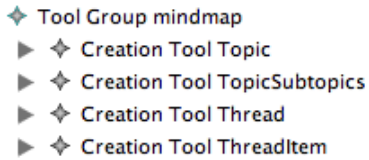


Figure 14. GMF – Sample tooling definition model[18].

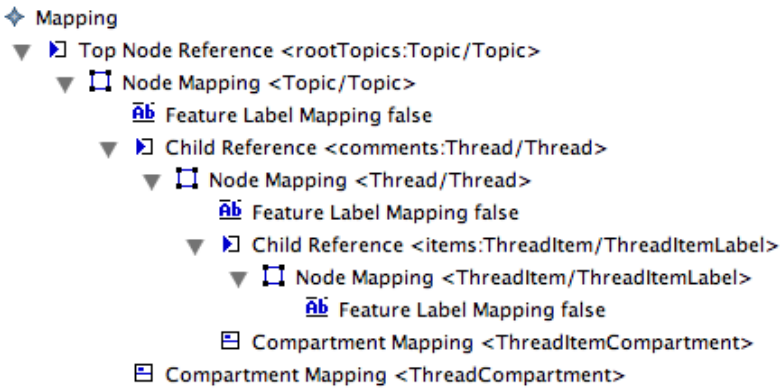


Figure 15. GMF – Sample mapping model[18].

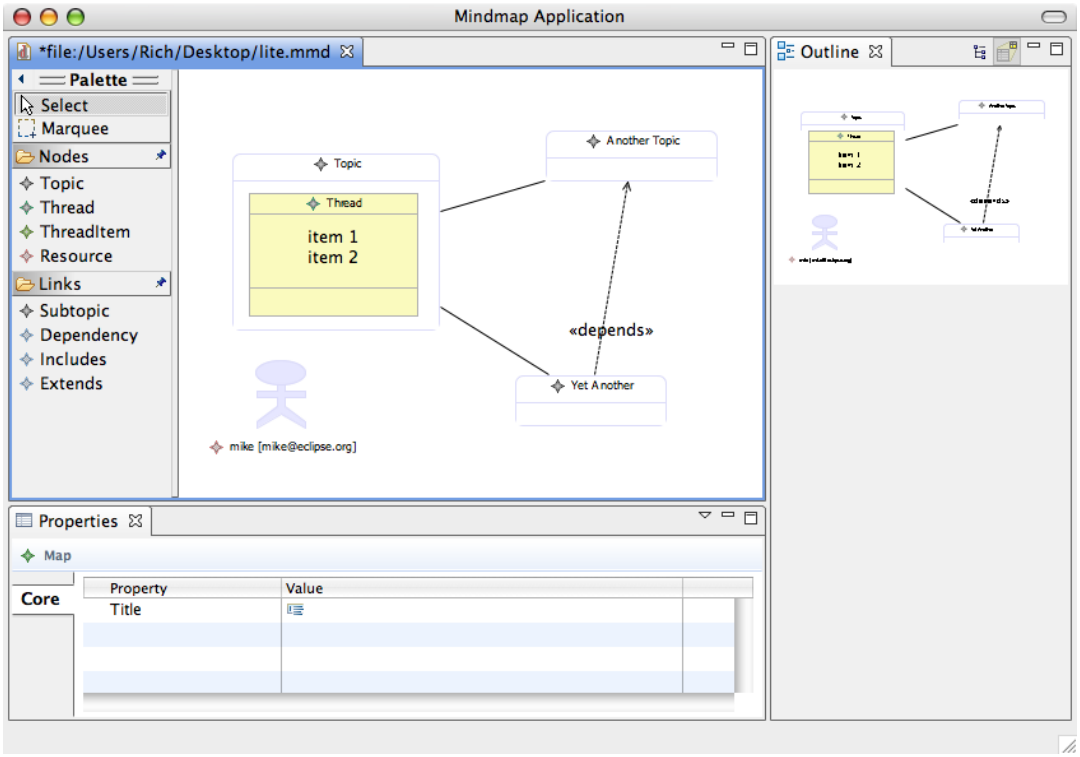


Figure 16. GMF – Sample generated editor[18].

The mapping model is automatically transformed into a generator model, from which a diagramming plug-in is generated by GMF. The generated editor provides a diagramming canvas with automatic layout capabilities, the tooling to create, edit, remove diagram elements, and can be customized by the toolsmith in several ways. Figure 16 is a screenshot of a mindmap editor automatically generated by GMF.

In a nutshell, to create a diagramming editor with GMF, a toolsmith needs to define or select a domain model, from which he can derive a graphical definition model and a tooling definition model. These two models can be customized before being associated with the domain model in a mapping model. Finally, the mapping model is “wrapped” in a generator model, which is used to generate the editor plug-in. This workflow is summarized in Figure 17.

IV.2.2. Textual syntax development

A graphical syntax is not always practical. For some applications, working with plain text files can be much more intuitive and effective. When this is the case, the toolsmith can leverage

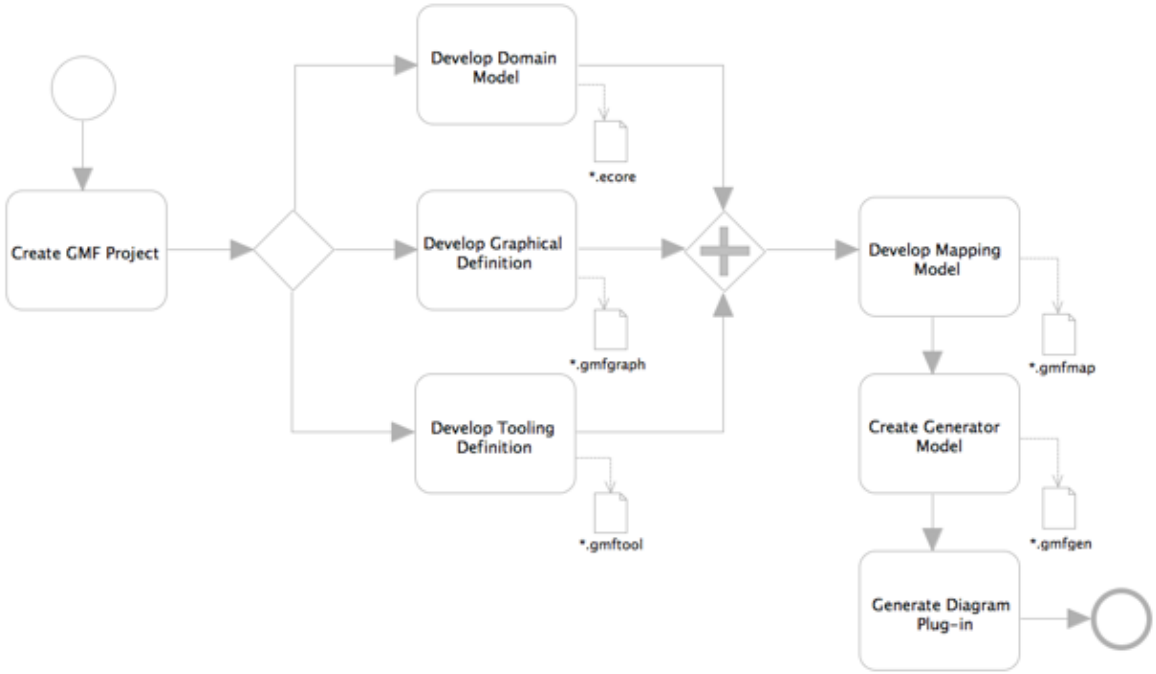


Figure 17. GMF-tooling workflow[18].

the Textual Modeling Framework (TMF) to define a textual concrete syntax for his metamodel.

At the time of this writing, TMF includes two subprojects that are still at an early stage. The first one, named Textual Concrete Syntax (TCS) [Jouault et al. 2006], does not seem to be supported anymore, and is very similar to its still active counterpart, Xtext.

Xtext provides a domain-specific language for defining grammars. This language is based on the Extended Backus Naur Form, but adds some additional features to facilitate the integration of the grammar into a larger MDE project. Concretely, an Xtext file defines a grammar in a classical way, as a collection of terminal symbols and production rules. However, it also contains so-called assignments, which are responsible for creating the abstract syntax tree (AST) as the parser processes the input. This way, the parsing rules and the structure of the resulting AST are contained in a single place, making it easy to map a concrete syntax to an existing domain model.

Given such a file, Xtext automatically generates a parser, a serializer and a full-blown editor for the specified language, in the form of an Eclipse plug-in. The parser allows a source specification in the defined language to be transformed into an instance model for further

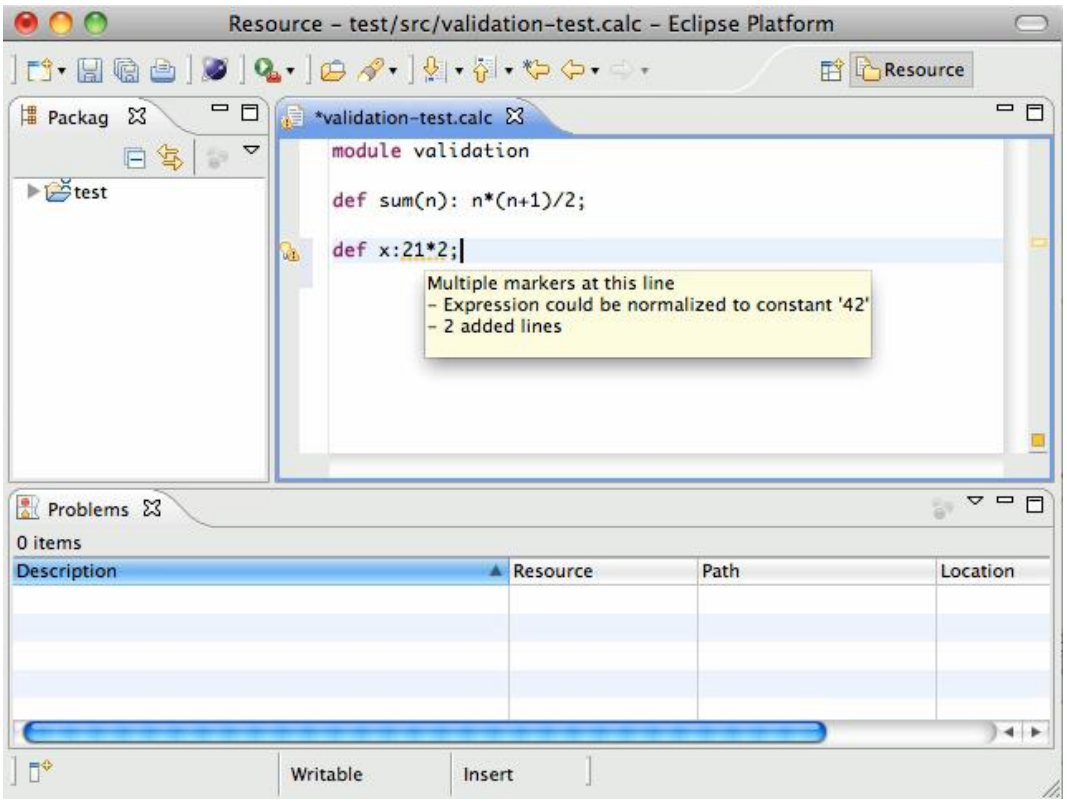


Figure 18. Sample editor generated by Xtext.

use, while the serializer takes a model and generates its representation in the textual language. As Figure 18 shows, the editor provides many user-friendly features such as syntax highlighting, autocomplete, error detection, outline view, and so on.

IV.3. Model transformations

We just saw the various possibilities offered by EMP to enable the creation and edition of models by a practitioner. However, remember that the goal of MDE is to make models productive and not only contemplative. As a consequence, the toolsmith role goes beyond the development of metamodels and editors: he must also provide the practitioner with a collection of transformations to automate as much tedious work as possible. These transformations can be divided in two main groups: model-to-model transformations, and model-to-text transformations.

IV.3.1. Model-to-model transformations

A model-to-model (M2M) transformation is a transformation from one or several input model(s) to one or several output model(s). EMP includes three M2M transformation

engines: INRIA's ATL, an implementation of OMG's QVT Operational, and an implementation of OMG's QVT Core and Relational. The latter is still experimental, and ATL and QVTO are quite similar [Jouault and Kurtev 2006], so we will only present the one we actually used for this thesis, namely QVTO.

QVTO is an imperative language: a transformation is defined as a sequence of mappings from source model elements to target model elements. A QVTO file begins with statements that import metamodel definitions. These metamodels are used to specify the type of models manipulated by the transformation, which can be input, output or input/output (in the case of in-place transformation) models. Then, an entry point for the transformation must be defined, which typically obtains the root element of the model and invokes a main mapping.

The transformation is defined through several mapping, each one specifying how a specific model element should be transformed. A mapping typically queries various properties or sub-elements of the source element, and creates diverse elements, properly initialized, in the target element. When dealing with composite elements, a mapping will often invoke other mappings on the components, until eventually the entire source model gets transformed.

Of course, the language provides usual features such as variables, conditional branching, loops etc. It is also possible to define queries over a model, so that they can be reused in several mappings. These queries are specified with OCL, the usual language for model manipulation used in many other places. Examples of QVTO transformations and OCL expressions are provided in Chapter 5.

IV.3.2. Model-to-text transformation

M2M transformations are very useful, but the end-result artifacts expected by a practitioner are often textual: the ultimate goal of many MDE projects is often to generate some source code in a GPPL, destined to be compiled into a final executable or library. Another example might be documentation, which must usually be generated in some textual format in order to be distributed.

Therefore, in addition to M2M transformations, there is a need for model-to-text (M2T) transformations. EMP provides three different components providing this feature. The first one is Java Emitter Templates (JET), a technology inspired by Java Server Pages; the second one is Acceleo, an implementation of yet another OMG MOF standard named Model to Text Language (MTL); the last one is Xpand, a comprehensive framework including a well-featured language along with the corresponding editor.

All these frameworks use a similar approach based on templating: the expected output is specified as a template file containing placeholders for embedding content specific to the input model. Such templates make it very easy to define any kind of output, in a concise way. The code of the placeholders is written in some querying language: Java for JET, OCL in MTL, and a custom language named Xtend for Xpand.

V. Conclusion

In this chapter, we introduced a software engineering approach called Model-Driven Engineering (MDE). MDE is a relatively recent domain, but draws inspiration from practices that have always existed in computer engineering. The goal of MDE is to change the way we develop, maintain and manage software applications. The main idea is to use multiple models to describe the software, using a vocabulary closer to the domain than what is usually proposed by general-purpose programming languages, and to automate as much as possible the generation of artifacts from these models, notably executable files.

To do so, MDE put forth the notion of metamodel, i.e. model defining precisely the structure and semantics of models, specified in well-defined (or even standard) languages (metametamodels). Based on these metamodels, a toolsmith can easily develop sets of tools to facilitate the development of applications by practitioners, such as specialized editors and automatic transformations between models, as well as code generators.

We presented the main MDE approaches. The Model-Driven Architecture is a collection of standards proposed by the OMG that rely heavily on UML and focus on the notion of platform-independent and platform-specific models. The concept of “software factories” is supported by Microsoft and aims at lowering the development costs of similar software (software product line) by creating a tooling specific to each software family. Finally, the

Eclipse Modeling Project is a pragmatic approach to MDE that contains a set of MDE tools for the Eclipse platform. We described this project in more details, explaining how a toolsmith can leverage EMP to develop metamodels, concrete syntaxes (either graphical or textual) along with the corresponding editors, and transformations (either model-to-model or model-to-text).

In Chapter 5, we will show how this approach can be applied to the domain of DEVS Modeling & Simulation, and how it can solve the issue of interoperability between heterogeneous tools, among other benefits. Before that, the next chapter will present another programming paradigm, namely metaprogramming. Like MDE, metaprogramming aims at automating the generation of software and can provide several improvements to the domain of DEVS M&S, especially to simulator implementations. We will see that metaprogramming operates at a lower level of abstraction than MDE, making implementations more complex but closer to the result aimed at, and arguably more powerful.

Chapter 4. Metaprogramming

I. Introduction

In Chapter 2.II.3, we saw that a great number of implementations are available to simulate DEVS models. All these simulators are based on a similar concept, which follows from the DEVS framework itself. Indeed, the major benefit of DEVS is to be able to model any system in a uniform way, and to simulate all these models in the same manner. As the DEVS formalism makes a clear distinction between models and simulators, it is possible to develop generic simulators capable of handling any DEVS models. This approach, which is used in all actual DEVS environments, comes at a cost: since simulators are not tailored for specific models, their performances are far from optimal and more importantly they cannot enforce all the constraints required by the formalism on the handled models.

An intuitive and (most of the time) easily verifiable notion is that the more generic a program is, the less efficient it is. A solution that is especially tailored to answer a specific problem or small set of problems has good chances of being more effective than a solution solving a bigger set of problems. For example, a piece of code computing the result of an equation will perform better than a full-fledged equation solver. A generic solution can also be more error-prone due to the wider range of inputs it must accept: verifying the validity of

a given input before computing the solution can sometimes become a problem on its own, and incur additional computations and iterative modifications of the input by the user.

However, this classic trade-off between abstraction/genericity and performance/ease of use/correctness can be overcome to attain the best of both worlds by using metaprogramming and particularly dedicated program generation. In this chapter, we introduce this programming paradigm that consists roughly in writing programs that generate programs. Metaprogramming can be used to develop generic solutions to some problems while still keeping the benefits of custom-tailored solutions: the trade-off between genericity and efficiency/safety is bridged over by using meta-solutions, which generate specialized solutions for large sets of problems instead of solving them directly.

We will start this chapter by explaining the main concept of metaprogramming. Then, we will provide a comprehensive survey of the current metaprogramming techniques, along with a comparison of their pros and cons. These two sections are largely based on a previous work we did in the context of real-time simulation [Touraille et al. 2012]. Finally, we will thoroughly describe a particular technique called C++ Template MetaProgramming (C++ TMP), which we put in practice in our meta-simulator.

II. Concepts and definitions

The simplest definition of a metaprogram is “a program that creates or manipulates a program”. This definition encompasses many concepts: source code generation, compilation, reflection, string evaluation, and so on. There is currently no broadly accepted taxonomy of metaprogramming systems. Two interesting propositions can be found in [Sheard 2001] and [Damaševičius and Štuikys 2008], where the authors identify concepts and relationships to characterize metaprograms. When dealing with performance, the most useful aspect of metaprogramming is program generation. Sheard provides the following definition for a program generator:

> *“A program generator (a meta-program) solves a particular problem by constructing another program (an object program) that solves the problem at hand. Usually, the generated (object) program is “specialized” for the particular problem and uses fewer resources than a general purpose, non-generator solution.”* [Sheard 2001]

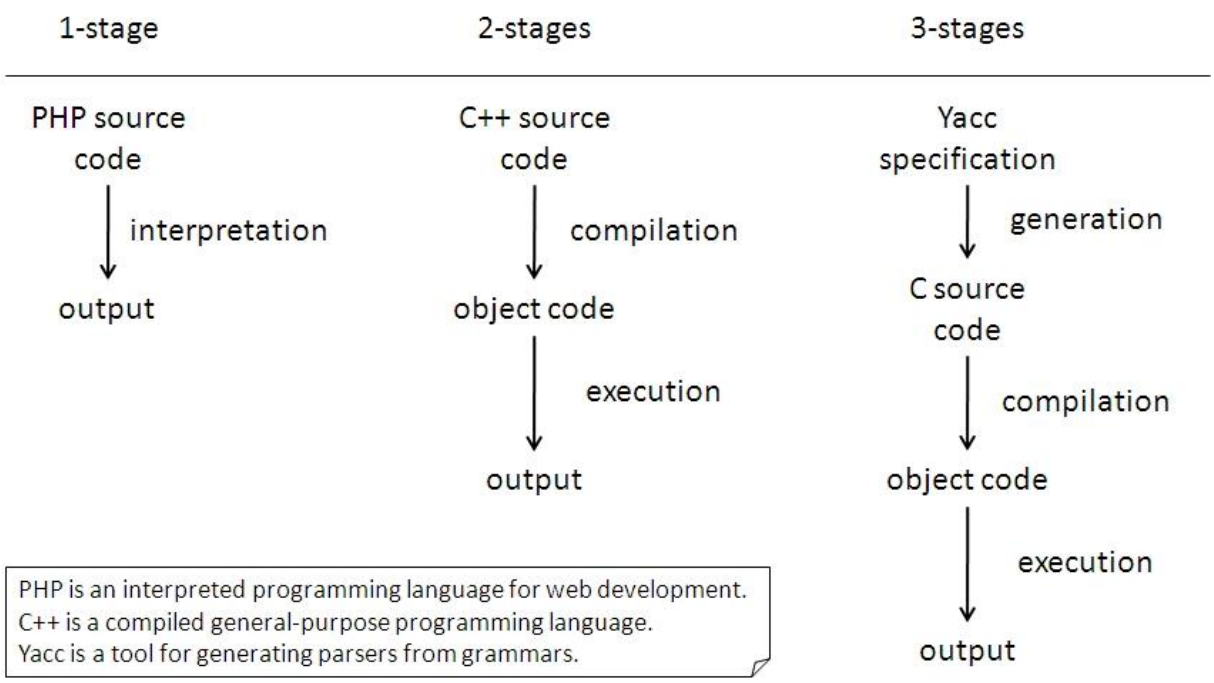


Figure 19. Samples of n-stages executions.

The definition by Damaševičius and Štuikys is more general and simply states that:

> *"Software generation is an automated process of creation of a target system from a high-level specification."* [Damaševičius and Štuikys 2008]

These definitions are nicely encompassed by the concept of *multi-stage programming* [Taha 1999], that is decomposing the execution of a program into several steps. As Figure 19 shows through some examples, any kind of program execution can be seen as a sequence of steps. In particular, execution can be performed in a single stage (interpretation as in PHP program execution), two stages (e.g., compilation + execution), or more (e.g., code generation + compilation + execution).

Adding a new stage to the interpretation approach means reducing the abstraction level, to finally produce a code optimized for performing the operations described in the source program. Optionally, each stage can be parameterized by other input than the source specification. For example, a compiler produces machine code corresponding to the input source code, and specific and optimized for a specific target machine.

In the following subsections, we will present several metaprogramming techniques that allow the generation of specialized and optimized artifacts without sacrificing genericity and abstraction. Adopting a pragmatic point of view, we will emphasize practical examples rather than theoretical aspects.

III. Survey of metaprogramming techniques

III.1. Text generation

Metaprogramming can be as simple as writing strings into a file. For example, the following C code is a metaprogram that generates a C file containing the code for printing the string passed to the metaprogram:

```
#include <stdio.h>

int main( int argc, char * argv[] )
{
    FILE * output;
    output = fopen( argv[ 1 ], "w" );

    fprintf( output, "#include <stdio.h>\n\n" );
    fprintf( output, "int main(void)\n" );
    fprintf( output, "{\n" );
    fprintf( output, "    printf(\"%s\");\n", argv[ 2 ] );
    fprintf( output, "}" );

    return 0;
}
```

Code 4. C metaprogram generating a C source code printing the string given as argument.

Of course, this contrived example adds little value compared to writing the object program directly. However, this technique becomes quite valuable when the object code is too tedious to write by hand, or when the metaprogram performs computations on its arguments before generation as all the computations that are performed beforehand will not need to be executed in the object program. Compared to other metaprogramming techniques, a major advantage of source code generation is that the output is readable by a human, thereby helping debugging. However, the string approach presented here has the drawback that the validity of the object program cannot be enforced at the meta level.

Two types of generation can be distinguished: data generation and instructions generation.

III.1.1. Data generation

The aim of data generation is to compute before execution a set of values so that they are already loaded in memory when the program starts. For example, some programs use a look-up table containing the sine of many values instead of computing the sine each time it is needed. Most of the time, this look-up table is populated at startup; however, a more efficient approach is to generate the code for initializing the look-up table directly with the correct values. This technique can also be used to accelerate stochastic simulation by having a metaprogram generate the pseudo-random numbers instead of generating the stream during execution. The metaprogram creates a file containing the initialization of an array with the generated numbers. This file can then be compiled and each simulation that must use the random stream only has to link to the object file obtained. In these programs, the time that it takes to obtain a pseudo-random number is significantly decreased (in [Hill and Roche 2002], the authors observed a factor of 5). This approach fits with the large memories we have even on personal computers since it favors execution time at the expense of additional object file space. When the amount of pseudo-random numbers is too large to be stored in memory, memory mapping techniques can be used [Hill 2002a] [Hill 2002b].

III.1.2. Generation of instructions

Data generation mostly consists in generating values. It is nevertheless metaprogramming, because the code needed to store these values in a data structure accessible at runtime must also be generated. However, this is not the nominal use of metaprogramming. A metaprogram occasionally generates data, but mainly generates instructions. We can consider the metaprogram as a metasolution to a given problem. Instead of solving the problem, the program generates a solution to the problem. A metaprogram is often more generic than a regular program, and it can create solutions specialized for the inputs it receives. For example, in [Missaoui et al. 2008], a metaprogram was used to generate efficient and carefully crafted programs for computing all the reverse translations of oligopeptides containing a given set of amino acids. The metaprogram took as input the set of amino acids to be considered and generated a non-recursive pile handling program with a great number of nested loops necessary to compute the backtranslations of any oligopeptide (composed of the amino acids specified at generation). The resulting programs

would have been difficult, if not impossible, to write manually without sacrificing some efficiency.

III.2. Domain-Specific Languages

Domain Specific Languages (DSLs) are programming or modeling languages dedicated to a particular domain. They contrast with General-Purpose Programming Languages (GPPLs) such as C, Java, and Lisp and general-purpose modeling languages such as UML, which can be used in any domain. A DSL usually defines a textual or graphical syntax for manipulating the domain entities. Specifications written in the DSL are compiled either directly to machine code or more often to source code in a GPPL.

The advantages of using DSLs are numerous. The most obvious is that they provide abstractions that are at the appropriate level for the user, hence expediting development of models and applications. However, the same effect can be more or less achieved in a GPPL through libraries. More interesting are the opportunities that DSLs bring regarding verification and optimization. Indeed, since the language is domain-dependent, the tools for manipulating specifications can themselves be domain-dependent. This means that verification about the program can be performed at design-time using domain knowledge. Similarly, the DSL compiler can perform several domain-specific optimizations [Lengauer 2003]. For the sake of illustration, let's consider electronic circuit simulation. Using a traditional library,[19] the user would manipulate classes and functions; for example, there could be classes for representing each type of logical gate. When compiling this program, all the GPPL compiler sees is classes and functions and so it is only able to perform generic optimizations such as loop unrolling, inlining, and so on. Now, if the same program were written in a DSL for circuit simulation (such as VHDL [Ashenden 2008] for example), the DSL compiler could perform more specific optimizations. For example, the DSL compiler could be aware that the combination of two NOT gates is equivalent to no gates at all, and consequently avoid generating the code for these two gates in the object program.

DSLs have been used in simulation for a long time. In [Hill 1993] [Hill 1996], Hill surveyed some of the most prominent simulation DSLs, which are shortly introduced here. Back in the

[19] "Traditional library" as opposed to "active library", a notion that is presented later in this section.

sixties, IBM introduced GPSS (Global Purpose Simulation System [Gordon 1962]), a language for discrete event simulation with both a textual and graphical syntax, which could be compiled to machine code or to interpretable pseudo-code. In the late seventies, the SLAM simulation language [O'Reilly and Nordlund 1989] was introduced. It made possible the usage of several modeling approaches (process, event, continuous [Crosbie 2012]), possibly combined together. A few years later, SIMAN (SIMulation ANalysis [Pedgen et al. 1995]) was released. It allowed modeling discrete, continuous, or hybrid systems, and was used in several modeling and simulation software such as ARENA, which is still evolving and widely used. We can also cite the QNAP2 (Queuing Network Analysis Package II [Potier 1983]) language, based on the queuing networks theory.

The trend in the past few years has been to make languages even more specialized. Indeed, the notion of specificity is rather subjective. After all, even though the languages presented above can be seen as more specific than a GPPL such as FORTRAN or C, they can also be seen as much more generic than a DSL for, for example, flight simulation. The more specific a DSL is, the easier it is to use by specialists who are not simulationists; in addition, it becomes more amenable to verification and optimization. However, it also greatly limits its scope; as a consequence, the cost of developing the DSL should not be higher than the outcome obtained by using it.

Fortunately, there are more and more tools available to ease their development. One of the most active areas in this regard is Model Driven Engineering (MDE) [Schmidt 2006] [Gronback 2009]. Indeed, the MDE approach and the tools developed to support it enable, among other things, the creation of new DSLs in a very short time. DSLs are supported by providing metamodels and model transformations to develop abstract syntaxes, concrete syntaxes, code generators, graphical editors, and so on.

III.2.1. Embedded Domain Specific Languages

The development of a DSL from scratch can be quite cumbersome. Tools are available to facilitate the process, such as Lex and Yacc [Levine et al. 1992] or MDE-related tools, but it is still not a seamless experience. Moreover, the DSL puts some burden on the user who must learn an entirely new syntax and use some specific tools to generate or execute its code. Finally, it sometimes happens that a DSL must contain a GPPL as a sublanguage to provide

the user with complete flexibility on some part of a program (these are sometimes called *hybrid DSLs*).

An interesting approach to overcome these issues is the concept of Embedded Domain Specific Languages (EDSLs), sometimes called *internal DSLs* [Fowler 2010]. These DSLs are embedded in a host language (usually a GPPL), meaning that they are defined using the host constructs and syntax. This implies that every expression in the DSL must be a valid expression in the GPPL. Even though this can seem an unacceptable restriction, the gains obtained by using an EDSL are quite attractive:

- the syntax is the same as the host GPPL, so the user can focus on learning the DSL semantics;
- the GPPL can be used in synergy with the DSL to have both the practical expressivity of the DSL and the theoretical expressivity of the GPPL;
- it is not necessary to write a custom lexer, parser and code generator because all of these are provided for the GPPL and can be exploited at no additional charges for the DSL, including the generic optimizations performed by the GPPL optimizer.

Fowler identified several techniques for writing DSLs in object-oriented languages and more broadly in any procedural language [Fowler 2010]. These techniques aim at providing a *fluent interface* to a library, meaning a language-like API. The code using the library should be readable as if it were written in a language of its own. Method chaining is one of the techniques that can be used to provide a fluent interface. Code 5 hereafter shows a sample code for operating a robot using a fluent interface. The interface of the “Robot” class is designed so that the code using it can be read as if using a language specific to robot manipulation.

```
Robot wall_e;

wall_e.
    right( 60 ).
    forward().
        length( 100 ).
        speed( 20 ).
    left( 20 ).
    deployArm();
```

Code 5. Sample use of a fluent interface for robot manipulation.

The syntax of the DSL can be even richer in languages that support operator overloading, such as C++ and C#, and/or operator definition, such as Scala and F#. In the first case, the DSL designer can reuse existing operators in his own language to provide a more usable syntax while in the second case, he can even create his own keywords to extend the original GPPL syntax.

However, as enjoyable as providing a language-like interface to a library can be, it is not acceptable in the case at hand if it unfavorably affect performance. In general, providing a fluent interface implies additional runtime computations, notably to keep track of the context. Does it mean that EDSLs should not be employed when execution time is a concern? No, not if we couple the notion of EDSL with the one of *active library*. Veldhuizen and Gannon, who coined the term in 1998, provide the following definition:

> "*Unlike traditional libraries which are passive collections of functions and objects, Active Libraries may generate components, specialize algorithms, optimize code [and] configure and tune themselves for a target machine [...].*" [Veldhuizen and Gannon 1998]

In other words, an active library can provide the same generic abstractions as a traditional one while at the same time providing maximum efficiency thanks to code generation and optimization, achievable through metaprogramming. We can see an active library as a library generator that creates libraries tailored for the user specific needs, depending on its parameterization.

Therefore, an EDSL implemented as an active library will be able to perform the same operations the compiler executes when dealing with external DSL. Ahead of execution, the library can perform abstract syntax tree rewriting, evaluation of constant expressions, domain-specific optimizations, and so on, effectively resulting in a code specialized for the user's program. Languages providing the most powerful facilities for implementing "active EDSLs" are functional languages such as dialects of the LISP family, which provides macros for extending the syntax and quasiquoting/unquoting[20] for generating code, and languages with very flexible syntax such as Ruby or Scala. However, these languages are not as widely

[20] These terms will be defined in the next subsection "Multi-stage programming languages"

used as other languages such as Java or C, and their strength lies in aspects other than performance. On the contrary, the C++ language is one of the most used languages and exhibits some of the best performances. And the good news for this now 'old' language is that it supports active libraries, EDSLs, and metaprogramming, through the use of the template mechanism, as will be exposed in the next subsection.

A pioneer in the area of C++ active EDSLs is Blitz++ [Veldhuizen 1998], a scientific computing library that uses metaprogramming to optimize numerical computations at compile-time, making it possible to equal and sometimes exceed the speed of FORTRAN. Since then, frameworks have been developed to facilitate the writing of C++ EDSL. The most accomplished one is probably Boost.Proto [Niebler 2007], a C++ EDSL for defining C++ EDSL. This active library makes it very easy to define grammars, evaluate or transform an expression tree, and so on, all in C++ and with most of the computation happening at compile-time. It has been successfully exploited in several domains such as parsing and output generation (Boost.Spirit), functional programming (Boost.Phoenix) or parallel programming (Skell BE [Saidani et al. 2009]).

III.3. Multi-stage programming languages

Multi-stage programming languages (MSPLs) are languages including constructs for concisely building object programs that are guaranteed to be syntactically correct [Taha 1999].

In traditional programming languages, program generation is performed either through strings or datatypes manipulation. The first approach, presented at the beginning of this section, has the major drawback that there is no way of proving that the resulting program will be correct. After all, "`obj.meth(arg)`" and "`î#%2$µ`" are both strings, but the first one may have a meaning in the object program while the second will probably be an erroneous piece of code. This issue can be solved by using datatypes to represent the object program (e.g., manipulating instances of classes such as Class, Method, Expression, Variable, and so on), but doing so makes the metaprogram much less readable and concise.

To overcome these problems, MSPLs contain annotations to indicate which pieces of code belong to the object program. These pieces of code will not be evaluated during execution and will simply be "forwarded" to the object program. They must nevertheless be valid code,

hence providing syntactic verification of the object program at the meta level. This is the core of the LISP macros system, where the quote operator is used to annotate object code. More interestingly, dialects of the LISP family usually include a backquote (quasiquote in Scheme) operator. A backquoted expression, as a quoted expression, will be included in the object program, but it can also contain unquoted expressions that will be evaluated before being included.[21] Backquoted expressions can be nested to provide an arbitrary number of stages. The LISP code below provides an example of a power function that generates the code for computing x^n for a fixed `n`. The `pow` function takes two parameters, the base `x` and the exponent `n`, and is recursively defined ($x^0 = 1$, $x^n = x * x^{n-1}$). However, since the multiplication is backquoted, it will not be performed immediately. Instead, it will be included in the generated code. The two operands being unquoted (through the comma operator), they will be replaced by the result of their evaluation. Eventually, the `pow` function will generate unrolled code containing neither recursive calls nor test for base case.

```
(define (pow x n)
    (if (= n 0)
        1
        `(* ,x ,(pow x (- n 1)))
    )
)

(pow 'var 4)

;Output: (* var (* var (* var (* var 1))))
```

Code 6. LISP Code generating the code computing x^n.

This backquote feature, even though powerful, can create some issues regarding symbol binding. In particular, instead of being bound at the meta-program level, they are bound during evaluation (in the object program), which is often not the desired behavior. The MetaML language [Taha and Sheard 1997] has been developed, among other reasons, to solve these problems.

Functional languages are well-suited for metaprogramming thanks to their homoiconicity ("data is code") and have been broadly used to develop metaprograms. However, functional languages are far from being mainstream and are not very well suited to computation-

[21] A similar feature, command substitution, can be found in Unix shells. Incidentally, the character used to denote commands that must be substituted is also the backquote.

intensive simulations. More classic choices in this domain are C, C++, and Java. In these languages, an arbitrary number of stages would be difficult to achieve, although one of them still has some multi-staging capability.

III.3.1. C++ Template MetaProgramming

C++, the well-known multi-paradigm language, includes a feature for performing generic programming: templates. As a reminder, a class (resp. function) template is a parameterized model from which classes (resp. functions) will be generated during compilation, more precisely during a phase called template instantiation. This feature is very useful for writing generic classes (resp. functions) that can operate on any type or a set of types meeting some criteria. Moreover, templates proved to be more powerful than what was originally thought when they were introduced in the language. In 1994, Unruh found out that they could be used to perform numerical computations such as computing prime numbers and Veldhuizen later established that templates were Turing complete [Veldhuizen 2003]. These discoveries gave birth to a metaprogramming technique called C++ Template MetaProgramming (TMP) [Abrahams and Gurtovoy 2004].

Veldhuizen described C++ TMP as a kind of partial evaluation (cf. Chapter 4.III.4), however, it does not really correspond to the accepted definition. A more appropriate way to look at it would be to consider C++ with templates as a “two-and-a-half-stage” programming language: one stage and a half for template instantiation and compilation (which are in this case two indivisible steps), and one stage for execution.

C++ TMP exhibits several characteristics very close to functional programming, namely, lack of mutable variables, extensive use of recursion, and pattern matching through (partial) specialization. As an example, Code 7 hereafter presents the equivalent in C++ TMP of the LISP code showed above. The class template `pow` is parameterized by an exponent `N`. It contains a member function `apply` that takes as parameter a base `x` and multiplies it by x^{N-1}. To do so, the template is instantiated with the decremented exponent and the `apply` member function of this instantiation is invoked. The base case is handled through template specialization for `N=0`. When the compiler encounters `pow<4>::apply(x)`, it successively instantiates `pow<3>::apply(x)`, `pow<2>::apply(x)`, `pow<1>::apply(x)`, and finally `pow<0>::apply(x)`, which matches the specialization (multiplication by 1), hence stopping

```
template < unsigned int N >
struct pow
{
    static double apply( double x )
    {
        return x * pow< N - 1 >::apply( x );
    }
};

template <>
struct pow< 0 >
{
    static double apply( double x )
    {
        return 1;
    }
};

[...]

double res = pow< 4 >::apply( var );
//the generated assembly code will be equivalent to var * var * var * var;
```

Code 7. C++ template metaprogram for computing x^n.

the recursion. Since the functions are small and simple, the compiler can inline them and remove the function calls, effectively resulting in generated code that only performs successive multiplications.

Writing metaprograms with C++ TMP is not as seamless an experience as it is in LISP or MetaML. The syntax is somewhat awkward since this usage of templates was not anticipated by the C++ Standards Committee. Fortunately, several libraries have been developed to smooth the process. The most significant ones are the MetaProgramming Library (MPL) and Fusion, two libraries belonging to the Boost repository. Boost.MPL is the compile-time equivalent of the C++ Standard Library. It provides several sequences, algorithms and metafunctions for manipulating types during compilation. Boost.Fusion makes the link between compile-time metaprogramming and runtime programming, by providing a set of heterogeneous containers (tuples) that can hold elements with arbitrary types and several functions and algorithms operating on these containers either at compile-time (type manipulation) or at runtime (value manipulation). Since C++ TMP is the technique we used to develop the DEVS-MetaSimulator, we provide a more thorough presentation in the next subsection, including introduction to these libraries.

III.4. Partial evaluation

The last metaprogramming technique we will present in this section is known as *partial evaluation* [Jones 1996]. Partial evaluation can be seen as automatic multi-stage programming where the programmer is not required to invest any additional effort to obtain the benefit of program specialization.

In mathematics and computer science, *partial application* is the technique of transforming a function with several parameters to another function of smaller arity where some of the parameters have been fixed to a given value. Partial evaluation is quite similar except that it applies to programs instead of mathematical functions. A partial evaluator is an algorithm that takes as input a source program and some of this program's inputs, and generates a *residual* or *specialized* program. When run with the rest of the inputs, the residual program generates the same output as the original program. However, in the meantime, the partial evaluator had the opportunity to evaluate every part of the original program that depended on the provided inputs. Consequently, the residual program performs fewer operations than the source program, hence exhibiting better performance. Formally, we consider a program as a function of static and dynamic data given as input, which produces some output:

$$P : \left(I_{static}, I_{dynamic} \right) \xrightarrow{yields} O$$

Given this definition, a partial evaluator can be defined as

$$PE : \left(P, I_{static} \right) \xrightarrow{yields} P^*$$

such that

$$P^* : \left(I_{dynamic} \right) \xrightarrow{yields} O$$

As an example, the C code of the binary search algorithm is shown in Code 8. The algorithm recursively searches a given value in an ordered array and returns the index of the value (or `-1` if it is not found).

```
int search( int * array, int value, int size )
{
    return binary_search( array, value, 0, size-1 );
}

int binary_search( int * array, int value, int firstIndex, int lastIndex )
{
    if ( lastIndex < firstIndex ) return -1;      // value not found

    int middleIndex = firstIndex + ( lastIndex - firstIndex ) / 2;

    if ( array[ middleIndex ] == value )          // value found
    {
        return middleIndex;
    }
    else if ( array[ middleIndex ] > value )      // value is before
    {
        return binary_search( array, value, firstIndex, middleIndex - 1 );
    }
    else                                          // value is after
    {
        return binary_search( array, value, middleIndex + 1, lastIndex );
    }
}
```

Code 8. Binary search in C.

The search function takes three parameters: the ordered array, the searched value, and the size of the array. Assuming we have a means to partially evaluate this function, Code 9 shows the residual function we would obtain with size = 3.

```
int search3( int * array, int value )
{
    if ( array[ 1 ] == value )
    {
        return 1;
    }
    else if ( array[ 1 ] > value )
    {
        if ( array[ 0 ] == value )
            return 0;
        else
            return -1;
    }
    else
    {
        if ( array[ 2 ] == value )
            return 2;
        else
            return -1;
    }
}
```

Code 9. Binary search residual function for a given size (3).

In this residual function, partial evaluation eliminated all recursive calls, the computation of the middle index, and the test for the "value not found" base case. Most of the time, specializing a function or a program will yield code that is both smaller and faster since many instructions will be eliminated. However, since partial evaluation performs operations such as loop unrolling and function inlining, it can sometimes lead to code bloat, a phenomenon known as over-specialization. In addition to improving performance, the specialization of functions or programs can also be used to verify assertions about inputs before execution. The assertions will be checked by the partial evaluator which will abort the residual program production if one of them is not verified.

At this point, it should be obvious that partial evaluation shares the same goal as other multi-stage programming techniques, namely, to produce an optimized version of a program, specialized for some data. The main difference is that the process of specialization is assigned to a partial evaluator, that is, a program that will automatically perform the generation. The partial evaluator is in charge of determining which pieces of data are *static* and hence can be exploited to perform ahead-of-time computations, and which ones are *dynamic* (not known before runtime).

A partial evaluator operates in two main phases. First of all, it must annotate the input program to discriminate between *eliminable* and *residual* instructions. This step, called *binding-time analysis*, must ensure that the annotated program will be correct with respect to two requirements: congruence (every element marked as eliminable must be eliminable for any possible static input) and termination (for any static input, the specializer processing the annotated program must terminate). The second step actually performs the specialization by using several techniques, the most prominent ones being symbolic computation, function calls unfolding (inlining), and program point specialization (duplication of program parts with different specializations).

There are not many partial evaluators available yet. Most of the existing ones target declarative programming (notably functional and logic programming), where programs can be easily manipulated. However, there also exist partial evaluators for some imperative programming languages such as Pascal and C. Regarding more recent languages, it is interesting to mention [Chepovsky et al. 2003], a work that aims at providing a partial

evaluator for the Common Intermediate Language. This Common Intermediate Language is the pseudo-code of the Microsoft .NET framework to which are compiled many high-level programming languages (C#, VB.NET, C++/CLI, F#...).

Partial evaluation can sometimes be combined with other metaprogramming techniques. In other work [Herrmann and Langhammer 2006], the authors apply both multi-stage programming (cf. previous subsubsection) and partial evaluation to the interpretation of an image-processing DSL. Their interpreter first simplifies the original source code by partially evaluating it with some static input (the size of the image), then generate either bytecode or native code using staging annotations, before eventually executing the resulting program with the dynamic input (the image to be processed). Even though all these steps are performed at runtime, the improvement in execution time as compared with classical interpretation is extremely good, up to 100x.

III.5. Comparison of the different approaches

In order to compare the metaprogramming techniques described previously, we retained the following criteria:

- **Generation speed**: how fast the metaprogram generates the object program.
- **Syntactic correctness of the object program**: whether the syntactic validity of the object program is enforced at the metaprogram level (✔) or not (✖).
- **Ease of development/Ease of use**: how easy it is to create and use metaprograms. It is important to draw the distinction between these two activities, especially in the case of DSLs and partial evaluation. Indeed, most of the time, the people developing the tools and the ones using it will not be the same, and the work they have to provide will be very different. For instance, developing a partial evaluator is a distinctly difficult task, while developing an application and partially evaluating it is almost the same as writing a “classical” application, since partial evaluation is mostly automatic. Regarding text generation and multi-stage programming, we considered the case of ad-hoc programs developed with a particular aim in mind, not libraries. Consequently, it is irrelevant to differentiate between development and use.

- **Use of common tools**: whether the user can use widely available tools such as a compiler or an interpreter for a common language (C/C++, Lisp…) (✔) or whether tools particular to the approach must be considered (✖).
- **Domain-specific optimizations**: potential of the metaprogram for performing optimizations specific to the domain at hand (e.g., flattening a model before runtime).
- **Non domain-specific optimizations**: potential of the metaprogram for performing generic optimizations (e.g., unrolling a loop with a constant number of iterations). Only the optimizations performed by the metaprogram are considered, not the optimizations applied on the object program at a later stage (such as optimization during the compilation of generated source code).

In Table 1 hereafter, scores range from one star (★) to five stars (★★★★★). Each criterion has been expressed so that a high score denotes an asset.

	Text generation	Domain Specific Langages		Multi-stage programming	Partial Evaluation
		External DSLs	Embedded DSLs		
Generation speed	★★★★★	★★★☆☆	★☆☆☆☆ to ★★★☆☆ (1)	★☆☆☆☆ to ★★★☆☆ (1)	★★☆☆☆
Syntactic correctness of the object program	✖	✖	✔	✔	✔
Ease of development	★★★☆☆	★★★★☆	★★☆☆☆	★★★☆☆	★☆☆☆☆
Ease of use		★★★★★	★★★★☆ to ★★★★★ (2)		★★★★☆
Use of common tools	✔	✖	✔	✔	✖
Domain specific optimizations	★★★★★	★★★★☆	★★★☆☆	★★★★☆	★☆☆☆☆
Non domain specific optimizations	★★★★☆	★★★☆☆	★★☆☆☆	★★★☆☆	★★★★★

(1) Depending on the language and the compiler used.

(2) Depending on the host language, the potential syntaxes can be more or less constrained.

Table 1. Comparison of metaprogramming approaches.

As can be seen from this table, there is no one-size-fits-all solution. Instead, each approach has its strengths and weaknesses and selecting one must be done on a case by case basis, depending on the properties required for the application at hand.

To write the DEVS MetaSimulator presented in this thesis, we chose C++ TMP, for several reasons:

- "Legacy" reasons. This work originated from the realization that most existing DEVS simulator were not really type-safe: they all relied on discarding type information, either through a "universal" base class (e.g. C#'s `object` class) or through a base class provided by the library (e.g. the `entity` class in DEVSJava). Consequently, we wanted to develop a prototype of a type-safe simulator, using genericity. we could have done it in C# or Java, since they both have generic programming facilities, but being more fluent in C++, we started working in this language. This is during the development of this prototype that we realized the potential of template metaprogramming with regard to DEVS simulation.
- C++ ubiquity. The C++ language is widely used: most developers can be expected to have a C++ compiler and IDE at hand, and at least basic notions allowing them to use a simple API.
- Performances. Simulation of complex models can quickly become costly in terms of computation. Therefore, a generic simulator should make provision for these cases by being sufficiently efficient (ideally without hindering its usability). C++ being at the moment one of the languages with the greatest potential regarding performance, it makes a sensible choice for writing a simulator.

IV. Metaprogramming in C++

To fully understand the simulator proposed in Chapter 6, one needs to grasp the concepts behind C++ metaprogramming, particularly preprocessor metaprogramming and template metaprogramming. Consequently, we provide hereafter a short tutorial that explains it in more details.

First, we will see the C++ features that can be leveraged to perform metaprogramming, before explaining the fundamental concepts being C++ template metaprogramming. Finally,

we will provide an overview of the multiple libraries at hand to smooth the development of metaprograms.

IV.1. C++ features used for metaprogramming

IV.1.1. Preprocessor

The C++ preprocessor, inherited from C, is a program invoked by the compiler to perform pre-compilation operations such as file inclusion, macro expansions and so on. The programmer uses preprocessor directives (recognizable by the leading '#') to include files into another, produce compiler errors, define constants and macros and perform conditional compilation. A macro is akin to a function in that it accepts arguments, but this is where the comparison ends. Indeed, a macro does nothing more than simple text substitution: everywhere the macro is invoked, the preprocessor replaces the invocation by the body of the macro, with each formal parameter being replaced by the actual arguments.

Macros are useful to generate code that would be tedious to write by hand, notably when a huge amount of repetitive code is needed, but should not be used as a replacement for usual factorization in functions. Indeed, because they only perform simple text substitution, macros can lead to unexpected behavior when invoking them with arguments containing side-effects, or when the replacement leads to code that will be parsed in a non-expected way.

The preprocessor can also conditionally include or excludes code from compilation. This is often used to remove some code from release build, or to accommodate compilers.

IV.1.2. Templates

Templates are a feature of C++ originally aimed at writing generic code, i.e. code working with a wide set of types. Indeed, C++ being statically and fairly strongly typed, every function parameters, variables, constants and so on must be assigned a type at compile-time. This greatly hinders reusability since many data structures and algorithms are independent of the type of the data manipulated and only need the type to fulfill a small set of requirements, such as allowing copy or comparison of values.

To overcome this issue, the C++ language provides the template mechanism. A class template (resp. function) is parameterized by one or more types, and defines a mold from which the compiler will generate classes (resp. functions) during a phase called template instantiation. Each time the template is used with a template argument that has not been encountered previously, the compiler instantiates the template with this argument, meaning that it creates a new class (resp. function) where all uses of the template formal parameter have been replaced by the type used as argument. For example, the C++ Standard Library provides several containers as class templates. These containers can be instantiated with any type fulfilling the requirements imposed by the container, so it is possible, for instance, to create lists of integers and lists of strings with a single `list` class template, as shown in Code 10.

```
std::list<int> integerList; // during template instantiation, the compiler
                            // will generate a class from the list
                            // template, using int as argument.
std::list<std::string> stringList; // another class will be generated, with
                                   // std::string as argument
```

Code 10. Two different instantiations of the same class template.

Templates are basically a metaprogramming facility embedded into the language. Class (resp. function) templates are metaclasses (resp. metafunctions) that are instantiated by the compiler with several arguments, effectively generating code without the developer being aware of that.

In addition to types, templates can be parameterized by integers, enumerations, pointers and pointers to member. Of course, since the template is instantiated at compile-time, the arguments must be known at compile-time (e.g. integral literals).

An important feature provided with the template mechanism is template specialization. We said before that templates let us write generic code, meaning having a single implementation usable with a wide range of types. However, for some types, the generic implementation can be flawed or simply inefficient. Therefore, we need a way to provide an alternate implementation for these types. This can be achieved through template specialization, by writing a new version of the template with a fixed set of arguments. For example, the standard class `vector` is specialized for booleans in order to optimize the

space used to store the values. Interestingly, for class templates, it is also possible to fix only some of the template parameters. This technique, called partial template specialization, allows complex pattern matching to occur when selecting the implementation to instantiate for a given set of arguments. For example, one can specialize a class template to handle pointer types in a particular way, whatever the pointed type is.

The use of C++ templates does not come without a cost. The first negative outcome they lead to is an increased compilation time: on most compilers, template instantiation is a costly process that sometimes drastically increase the time needed to compile a program. This behavior is mainly due to the way template instantiation is implemented; by changing this implementation, some recent compilers [Clang] manage to greatly reduce this negative impact. The second drawback of using templates is code bloat: since the compiler generates a new class/function for each set of arguments encountered, the size of the resulting executable can become significant. Some techniques can be used to limit this effect, like using a type-unsafe base implementation with a thin type-safe wrapper.

IV.1.3. Substitution Failure Is Not An Error

The last feature quite useful for metaprogramming in C++ is a relatively unknown rule often referred to as Substitution Failure Is Not An Error (SFINAE). This rule states that when substitution of template parameters leads to invalid code, the compiler must not generate an error and simply ignore the template. Thanks to that, it is possible to provide several overloads/specializations for different scenarios without worrying about breaking the compilation. For the sake of illustration, consider Code 11 hereafter:

```
template < typename T >
T * begin( T * array )
{ return array; }

template < typename Container >
typename Container::iterator begin( Container & container )
{ return container.begin(); }
```

Code 11. Generic `begin` function template.

The aim of these functions is to provide a homogeneous way to obtain an iterator at the beginning of a range of values, be it an array or a container. Assuming we did the same to

obtain the end iterator, we can now use standard algorithms identically with arrays and containers, as shown in Code 12:

```
1  int arr[] = { 0, 1, 2 };
2  std::vector<int> vec( arr, arr + 3 );
3
4  int sum;
5
6  sum = std::accumulate( begin(arr), end(arr), 0 ); // sum == 3
7  sum = std::accumulate( begin(vec), end(vec), 0 ); // sum == 3
```

Code 12. Sample use of the generic `begin` function.

When the compiler encounters a call to begin, it must instantiate all function templates in order to choose the correct overload to invoke. However, this can lead to invalid code, as is the case on line 6 where we call begin with an array of integers: the instantiation of the second overload should result in a compiler error since `int[]` does not define a nested type named `iterator`. Fortunately, thanks to SFINAE, the compiler will not emit any errors and simply remove this template from the set of potential overloads. The same rule holds when dealing with class template specializations.

SFINAE tricks can be used to achieve some kind of compile-time introspection, for example determining if a type defines a member with a given name or provides a default constructor. This is also very useful to enable or disable function overloads and template specializations depending on a compile-time condition. The Code 13 shows how we can write a sort function that uses different algorithms depending on the size of the input array:

This code might seem a little esoteric, but it is not all that complicated. Each function first declares a parameter of type “reference to an array of N elements of type T”. The type of the elements T and the size of the array N will both be deduced by the compiler when calling the function. The second parameter is an unnamed parameter whose type is “pointer to an array of X chars”. The trick is that X is dependent on the template parameters. In the first version, if the size of the input is inferior to 1024, the array has a size of 1, which is perfectly fine. However, if the size is superior or equal to 1024, the type of the parameter will be “pointer to an array of 0 chars”. Since zero-sized arrays are illegal in C++, the compiler will not consider this function for overload resolution, and the other one will be chosen. In addition, the parameter is defaulted so that the user needs not be aware of its presence.

```
template < typename T, size_t N >
void sort( T (&array)[ N ], char (*)[ (N < 1024) ] = 0 )
{
    // uses algorithm suited to small inputs
}

template < typename T, size_t N >
void sort( T (&array)[ N ], char (*)[ (N >= 1024 ) ] = 0 )
{
    // uses algorithm suited to big inputs
}

int small[ 50 ];
int big[ 5000 ];

sort( small ); // invokes the first overload
sort( big );   // invokes the second overload
```

Code 13. Sample use of Substitution Failure Is Not An Error (SFINAE).

The Boost libraries provide some utility classes, namely `enable_if` and `disable_if`, to make the use of SFINAE to enable/disable overloads and specializations more readable, as can be seen in Code 14:

```
template < typename T, size_t N >
typename boost::enable_if_c< (N < 1024) >::type sort(T (&array)[N]) {…}
template < typename T, size_t N >
typename boost::disable_if_c< (N < 1024) >::type sort(T (&array)[N]) {…}
```

Code 14. SFINAE with `enable_if` and `disable_if`.

IV.2. Data structures and control flow

As other programming paradigms, C++ template metaprogramming uses data structures and some ways of controlling the flow of execution. However, the data that can be manipulated is quite restricted: it can only be compile-time entities, meaning types and constants. Similarly, the flow of execution concerns operations performed by the compiler, not runtime operations.

As we will see, template metaprogramming is quite similar to functional programming in several aspects, most notably the lack of mutable state, extensive use of recursion to define data types and functions, and pattern matching for conditional execution.

IV.2.1. Typelists

Typelists are the primary data structures used in C++ TMP. Invented by Alexandrescu [Alexandrescu 2001], typelists provide a mean to store types in a compile-time structure, which can then be passed to compile-time algorithms and iterated over for processing.

Typelists are recursively defined: a typelist is composed of a head, which can be any type, and a tail, which must be itself a typelist. The following code gives one possible definition of a typelist[22]:

```
template < typename Head, typename Tail >
struct typelist
{
    typedef Head head; // car equivalent
    typedef Tail tail; // cdr equivalent
};
```

Code 15. Basic definition of a typelist.

You can note the similarity between this definition and the `cons` function familiar to LISP programmers. Replace `typelist` by `cons`, `head` by `car` and tail by `cdr`, and the similarity should be even more obvious.

Using this definition, we can create an arbitrary long list of types. Here is how we would declare the list of all signed integral types:

```
typedef typelist< char,
        typelist< short,
        typelist< int, long > > > signedIntegerTypes;
```

Code 16. Typelist of signed integer types.

However, you will notice that the most inner list (`typelist< int, long >`) is a bit odd: the first element is indeed a type, but the second one is not a list, it is a single value. This inconsistency would complicate the writing of algorithms, but can be easily solved by using a special type to denote the empty list:

[22] Note that this definition does not impose the use of a typelist for the second argument (the tail). A better implementation would use partial template specialization to make this requirement explicit and trigger a compiler error when it is not fulfilled.

```
struct null_type; // does not even need a definition,
                  // declaration is enough.

typedef typelist< char,
        typelist< short,
        typelist< int
        typelist< long, null_type> > > > signedIntegerTypes;
```

Code 17. Null-terminated typelist of signed integer types.

Given a list, we can access its element using the nested typedefs or, as we will see in a few moments, using pattern matching through template specialization:

```
signedIntegerTypes::head someChar = 'c';
signedIntegerTypes::tail::head someShort = 12;
```

Code 18. Simple access to elements of a typelist.

The approach exposed here can be leveraged to create more complex data structures such as maps or trees. The metaprogramming libraries described later in this subsection provide many such type containers, essential for TMP.

IV.2.2. Metafunctions

Now that we have seen how to store types in compile-time structures, we need some way of manipulating these types. To do so, we will use so-called *metafunctions*, meaning functions that operate on types or constants, at compile-time. Concretely, a metafunction is a class template, parameterized by some types and/or constants, which "returns" some type or constant through a nested typedef or a static constant member variable.

The simplest metafunction is the identity: it returns the type given as argument.

```
template < typename T >
struct identity
{
    typedef T type;
};
typedef identity< int >::type integer; // integer == int
```

Code 19. `identity` metafunction.

The convention for metafunctions is to return a single entity, named `type` if it is a type or `value` if it is a constant.

Several small metafunctions are included in the C++ standard library, under the name "trait classes". For example, the `add_pointer` trait class can be used to transform a type into a pointer-to-type:

```
typedef std::add_pointer< T >::type pointer_to_T;
```

Code 20. Sample trait class provided by the C++ Standard Library.

The advantage of using this over `T*` is that if `T` is a reference type, `add_pointer` removes the reference before adding the pointer. This kind of manipulation would not be possible without resorting to template metaprogramming.

Of course, metafunctions can be much more powerful (and, inevitably, more complex). As an illustration, we will write a metafunction that computes the size of a typelist. Since C++ TMP precludes mutable state, we need to use a recursive algorithm. It is quite simple: if the list is empty (base case), return 0; else (recursive case), return the size of the tail plus one.

Let us first examine the recursive case. Remember that a metafunction is implemented by a class template, where each parameter of the function corresponds to a template parameter. Thus, we need a class `size` parameterized by a typelist. To compute and return the size, we define a constant and initialize it with the correct value, which is computed by adding one to the result of recursively invoking (instantiating) `size` with the tail of the list:

```
// recursive case
template < typename List >
struct size
{
    static const size_t value =
        1 + size< typename List::tail >::value;
};
```

Code 21. `size` metafunction – recursive case.

Now that we have defined the recursive case, we need to implement the base case in order to prevent the compiler from infinitely instantiating `size`. The only way to achieve this is to provide a template specialization that will eventually be chosen by the compiler and stop the chain of instantiations. The base case occurs when the given list is empty, so we need to specialize `size` for `null_type`, the type we use to represent an empty list:

```
// base case
template <>
struct size< null_type >
{
    static const size_t value = 0;
};

//usage
typedef typelist< int, typelist< char, null_type > > someList;

size_t someListSize = size< someList >::value; // someListSize == 2
```

Code 22. `size` metafunction – base class and usage.

This sample metafunction shows how C++ TMP compares to functional programming. Due to the lack of mutable state, the logic is implemented using a combination of recursion (through recursive instantiation) and pattern matching (through specialization).

For the sake of completeness, here is a more evolved example that demonstrates how more complex pattern matching can be used, as well as the use of intermediate results to clarify the code. The aim of this metafunction is to append a type at the end of the typelist. Once again, we need to rely on recursion to iterate over the list until the end. Appending a type to an empty list boils down to creating a list containing the type as sole element. In the case of a non-empty list, we start by appending the type to the tail, before concatenating the head to the result of the previous computation.

```
// recursive case
template < typename List, typename T >
class append
{
    // intermediate computation
    typedef typename
        append< typename
            List::tail,
            T
        >::type tailPlusT;

  public:

    // result
    typedef
        typelist< typename
            List::head,
            tailPlusT
        > type;
};
```

Code 23. `append` metafunction – recursive case.

In order to make the code more readable, the recursive case uses a typedef to store the result of the recursive call. Only the final result of the metafunction, namely `type`, is made accessible to the outside of the class by placing it in a public section. To write the base case, we cannot just use an explicit value as pattern, like we did in the previous example. We need to match any pair of arguments where the list is empty, without enforcing any constraints on the type to append. This can be achieved through partial template specialization, which allows the programmer to fix only some of the arguments.

```
// base case
template < typename T >
struct append< null_type, T >
{
    typedef typelist< T, null_type > type;
};
```

Code 24. `append` metafunction – base case.

IV.2.3. Static polymorphism

Subtype polymorphism is the object-oriented principle that let programmers manipulate several objects whose types belong to the same hierarchy in a uniform way, without caring about the actual type of the objects. One can write functions or classes manipulating

references to a base class or an interface, and use these with any object whose type derives from the base class or implement the interface. In practice, this feature is implemented through virtual methods, which can be overridden by subtypes to provide customized behavior. Any call to a virtual method performed on a base class reference will be resolved at runtime and end up invoking the most specialized override corresponding to the actual type of the instance. For example, consider the following hierarchy, where the `Serializable` interface is implemented by two otherwise unrelated classes, `Settings` and `Results`:

```
struct Serializable
{
    virtual void serialize( Stream & outputStream ) const = 0;
};

struct Settings : Serializable
{
    // implement serialize
    void serialize( Stream & outputStream ) const { … }
    … // other members
};

struct Results : Serializable
{
    // implement serialize
    void serialize( Stream & outputStream ) const { … }
    … // other members
};
```

Code 25. Representation of serializable objects through inheritance.

Given that, one can write a polymorphic function capable of handling any objects implementing the `Serializable` interface:

```
void serializeToSomeFile( Serializable const & serializable )
{
    FileStream someFile( "path/to/someFile" );
    serializable.serialize( someFile ); // dynamically resolved
}
```

Code 26. Serialization with inheritance.

The call to the serialize method will be resolved at runtime, invoking the method appropriate to the actual type of the serializable object.

```
Settings settings;
Results results;

serializeToSomeFile( settings ); // invokes Settings::serialize
serializeToSomeFile( results );  // invokes Results::serialize
```

Code 27. Invocation with dynamic dispatch.

In generic programming with templates, the way to achieve polymorphism is a bit different. Instead of relying on base classes and interfaces to define the kind of messages that an object can receive, generic programming adopt an approach close to duck typing: if an object "quacks like a duck" and "walks like a duck", then it can be considered a duck, without explicitly stating that it **is** a duck. Practically, this means that any object can be used with a generic function/data type, as long as it fulfills a given set of requirements (providing a method with certain parameters, overloading some operators, etc.). The previous example can be rewritten with generic programming:

```
// no more need for the Serializable interface

struct Settings
{
    // no more need for virtual methods
    void serialize( Stream & outputStream ) const { … }
    … // other members
};

struct Results
{
    void serialize( Stream & outputStream ) const { … }
    … // other members
};

// Generic function, templated on the type of Serializable
template < typename Serializable >
void serializeToSomeFile( Serializable const & serializable )
{
    FileStream someFile( "path/to/someFile" );
    serializable.serialize( someFile ); // statically resolved
}
```

Code 28. Representation of serializable objects through genericity.

The generic function `serializeToSomeFile` can be called with any object providing a `serialize` method taking a `FileStream` as argument:

```
Settings settings;
Results results;

serializeToSomeFile( settings ); // calls serializeToSomeFile< Settings >,
                                 // which invokes Settings::serialize
serializeToSomeFile( results );  // calls serializeToSomeFile< Results >,
                                 // which invokes Results::serialize
```

Code 29. Invocation with static dispatch.

The main difference with the previous version lies in the time when the call to the method is resolved. With dynamic polymorphism (often referred to as dynamic dispatch), the correct method will be chosen at runtime, using some indirection such as a pointer to a virtual method table. Here, the choice is performed at compile-time, hence the name “static polymorphism”.

Bertrand Meyer provided a comprehensive comparison of these two approaches, genericity versus inheritance [Meyer 1988]. We will not delve into the topic; instead, we will focus solely on the technique at hand, i.e. static polymorphism. Here are some of its advantages compared to dynamic dispatch:

- It incurs a slight improvement in performance. Indeed, since no indirection is needed, the compiler can perform more optimizations, for instance inlining and branch prediction.
- Pass-by-value is possible, while dynamic dispatch imposes pass-by-reference or pass-by-address.
- All type information regarding the argument is retained: the actual type of the argument is known to the compiler in the called function. This is particularly important in template metaprogramming since types are the primary data that is manipulated.

Of course, these advantages do not come for free. In addition to the costs inherent to programming with templates (potential code bloat, increased compilation time), static polymorphism is hardly compatible with return values meant to be used polymorphically. Indeed, sometimes a function can return several instances with different types, depending on some conditions. Dynamic polymorphism handles that by making the return type a reference/pointer to a common base class, but this is no longer possible with static

polymorphism since we want to retain the type of the instance. Two solutions are possible to overcome this issue. The first one concerns the cases where the type of the instance returned depends only on compile-time values such as template parameters. In these cases, it is possible to write a metafunction, akin to the function, that provides the return type corresponding to the arguments given. The function must then be specialized so that each specialization returns only one type of object. The second solution must be used when the type of the return value depends on runtime conditions. The trick is to use a callback to forward the value instead of returning it. This callback must be a template function, and will be invoked in place of each return statement. Consequently, the “return” value will be given to the callback along with all its type information.

IV.3. Metaprogramming libraries

Since its discovery, C++ metaprogramming has drawn a lot of interest in the developer community. As a consequence, many libraries have been created to facilitate its use. One of the first library providing and using template metaprogramming facilities, Loki, was written in 2001 by Alexandrescu and described in his seminal book “Modern C++ design” [Alexandrescu 2001]. Loki provided – and still does, despite the cessation of its development – smart pointers, typelists, implementations for several renowned design patterns, and so on. Many of the ideas and concepts appearing in Loki have since then inspired many other more fine-grained libraries, most of which are included in the Boost repository.

To develop the work exposed in Chapter 6, we made extensive use of some of these libraries, which we will now briefly introduce.

IV.3.1. Boost.Preprocessor

Boost.Preprocessor [Boost.Preprocessor] [Abrahams and Gurtovoy 2004] is a metaprogramming library providing facilities to generate code at preprocessing-time. Contrary to the other libraries presented in the following pages, Boost.Preprocessor is not related to the template mechanism. Instead, it allows the developer to leverage the C++ preprocessor to generate code that would otherwise need to be manually written. This is particularly useful to avoid writing repetitive code that would be incredibly tedious to write by hand, but also to write generic libraries whose code depends on user’s parameters.

By using macros and preprocessor directives such as file inclusion, Boost.Preprocessor allows one to write quite evolved code-generating code. The major features provided are described hereafter.

- **Repetition**. The library proposes several ways to repeatedly generate some code, possibly with a varying argument. This can be used to generate simple things such as a long list of parameters, but also more complicated code, like classes and functions for instance. Such repetition is achieved either through higher-order macros or through recursive file inclusions.
- **Arithmetic, logical and comparison operations**. Preprocessor metaprograms often need to manipulate numerical values. However, performing computation on them during preprocessing can be a bit tricky due to the textual nature of the preprocessing process. To ease the manipulation of values, Boost.Preprocessor implements the most common operations as macros. For example, to test if some token `x` is even, one would write `BOOST_PP_EQUAL( BOOST_PP_MOD( x, 2 ), 0 )`.
- **Conditional construct**. Macros can be invoked conditionally thanks to an `if-then-else` construct, implemented as a higher-order macro.
- **Data structures**. As in TMP, it is possible to define data structures in preprocessor metaprogramming. When TMP structures are meant to manipulate types, preprocessor structures allows one to store and pass around macro arguments, usually text to be used in the generated code, but also numerical arguments or even macros to be invoked. Boost.Preprocessor supports four types of structures: *sequences*, which are strings composed of successive parenthesized macro arguments; *tuples*, which are parenthesized comma-separated lists of macro arguments; *arrays*, which embed both a tuple and its length, to ease manipulation; and *lists*, which are recursively defined with a head being a macro argument and a tail being itself a list or nil.

  Here are examples of each kind of structure:

```
#define SOME_SEQ ( int i ) ( 4 ) ( func(i) )
#define SOME_TUPLE ( int i, 4, func(i) )
#define SOME_ARRAY ( 3, SOME_TUPLE )
#define SOME_LIST ( int i, ( 4, ( func(i), BOOST_PP_NIL ) ) )
```

Code 30. Boost.Preprocessor data structures.

- **Algorithms**. To manipulate the data structures just presented, Boost.Preprocessor provides several algorithms. They allow accessing, inserting and removing elements, filtering a sequence based on a predicate, applying a macro to each elements of the structure, converting between structures and so on.

In this work, we used the preprocessor for two major things: overcoming the lack of variadic templates (i.e. templates with a variable number of arguments) in the current C++ standard[23], and providing a nicer syntax for users by hiding some complexity and implementation details behind macros.

IV.3.2. Boost.Type Traits

Traits classes are class templates used to associate some information with a type, usually some other types or some properties. To facilitate their use with metaprogramming, each class template in the Boost.Type Traits library focuses on a single trait; they are unary (sometimes binary) metafunctions, returning a single type or value. Type traits can be grouped in two main categories: those that describe a property of a type, and those that transform a type into another.

Many properties of a type can be obtained at compile-time. For instance, one can test if a type is a pointer, if it is an integral type, if it has a trivial constructor or if one type is a subtype of another. Boost.MPL even provides a macro for generating a type trait indicating if a type has a member with a given name. This compile-time introspection is essential in TMP where type properties often determine the flow of execution. It is also very useful in generic programming to provide different implementations of an algorithm or data structure depending on some properties of the type used to instantiate it.

Regarding transformations, Boost.Type Traits contains metafunctions to add and remove const/volatile qualifiers, reference, pointer, as well as facilities to transform integral types into signed/unsigned types, decay arrays into pointers, etc. These transformations are much simpler than the one provided by Boost.MPL, but are basic constructs essential to the development of generic code and more complex metafunctions.

[23] The C++11 standard thankfully fills this lack.

Generic programming being a paradigm widely used in C++, the C++ Standards Committee decided that type traits should be included in the new standard C++11 to make developers' work easier. Consequently, many of the class templates proposed in Boost.Type Traits are now part of the standard library.

IV.3.3. enable_if

enable_if is a set of utilities, included in the Boost repository, which aim at facilitating the use of SFINAE. Several class templates are provided to enable (`enable_if`) or disable (`disable_if`) function overloads or template specializations depending on some condition. The condition can either be a constant boolean or a metafunction returning a boolean (through a static member named `value`).

To enable or disable a function overload, one simply needs to use the corresponding class template in the function signature, either in the return type or as the type of a parameter (possibly unused). To make things easier for the user, these class templates are metafunctions that return `void` by default but can be parameterized to return some other type. This way, they can more easily be used in the return type of functions for example.

The following code gives an example (a bit contrived) where a multiplication by 8 is performed with bit shifting if the parameter is an integer, or with the multiplication operator otherwise.

```
template < typename T >
typename boost::enable_if< is_integral< T >, T >::type
multiplyBy8( T value )
{
    return value << 3;
}

template < typename T >
typename boost::disable_if< is_integral< T >, T >::type
multiplyBy8( T const & value )
{
    return value * 8;
}
```

Code 31. Function specialization with `enable_if` and `disable_if`.

IV.3.4. Boost.MPL

The libraries presented until now are useful to do Template MetaProgramming, but are not solely focused on this usage. On the contrary, the Boost.MetaProgramming Library (MPL) is specifically aimed at making C++ TMP an enjoyable experience, by providing a great number of facilities hiding most of the complexity of this type of programming (within the limits of the language, of course).

Metafunctions and lambda expressions

Two of the core concepts of the library are metafunctions and metafunction classes. We saw previously that a metafunction is a compile-time entity parameterized by types and returning a type, usually through a nested typedef named `type`. To facilitate functional programming, the MPL also defines the concept of metafunction class: a metafunction class is a class containing a nested metafunction named `apply`. This notion is very useful since it allows metafunctions to become first-class values that can be passed around and manipulated like regular types. In particular, they can be passed to other metafunctions, enabling higher-order programming. These invocable entities are called *lambda expressions*.

Writing the metafunction class corresponding to each metafunction would quickly become tedious, and would greatly clutter up the code. To avoid that, the MPL provides a mean to create lambda expressions “on the fly”, at the point where they are needed. It achieves this by providing *placeholders*, special types that can be used in a metafunction argument list but are replaced with actual arguments when the lambda expression is invoked. A metafunction whose argument list contains placeholders is a *placeholder expression*, and becomes a lambda expression, meaning a type that can be invoked, possibly with some parameters.

As an example, consider the `is_same` binary metafunction, which returns true if its arguments are the same type and false otherwise. This metafunction is provided by the standard library, but a possible implementation would be

```
template < typename T, typename U >
struct is_same : false_type {}; // false_type = type "encapsulating" false

template < typename T >
struct is_same< T, T > : true_type {}; // true_type = true
```

Code 32. Possible implementation of the `is_same` metafunction.

Since it is a metafunction, it is not a type but a class template, so it cannot be used like other types (especially, it cannot be passed as argument to other metafunctions). One solution would be to define a metafunction class encapsulating `is_same`:

```
struct is_same_f
{
    template < typename T, typename U >
    struct apply : public is_same< T, U > {};
    // note the use of inheritance to appropriate the result of the
    // metafunction, a trick called metafunction forwarding
};
```

Code 33. Encapsulation of a metafunction into a metafunction class.

However, having to write such a wrapper for every single metafunction we want to transform into a lambda expression would be a real burden. A simpler solution is to use placeholders:

```
namespace mpl = boost::mpl[24];

typedef is_same< mpl::_, mpl::_ > is_same_f;
```

Code 34. Using placeholders to turn a metafunction into a lambda expression.

This placeholder expression (which is now a type and not a class template) can be used as an argument and can be invoked with some parameters at some point. The placeholders will be replaced with the arguments provided during the invocation.

```
typedef mpl::apply< is_same_f, int, int >::type result; // true
```

Code 35. Invocation of a lambda expression with actual arguments.

When defining a placeholder expression, not all parameters need to be bound to a placeholder. Partial application is possible, meaning that some parameters can be given a

[24] For the sake of brevity, the rest of this document will assume such a namespace alias has been defined.

value while others remain unbound, effectively producing a new metafunction (or rather lambda expression) with a smaller arity:

```
typedef mpl::apply< is_same< int, mpl::_ >, float >::type result; // false
```

Code 36. Partial application of a metafunction.

Note that the lambda expression is invoked with only one argument (`float`): indeed, the first argument of `is_same` was fixed in the placeholder expression, hence resulting in a unary lambda expression.

The MPL provides several general purpose metafunctions for higher-order programming (invocation, composition, argument binding) as well as several `if` metafunctions for selecting a type or executing a metafunction depending on a condition.

Sequences and algorithms

The major part of the library consists of sequences and algorithms. In a way very similar to the C++ Standard Library, the MPL provides many containers corresponding to different data structures, as well as several algorithms to operate on these containers. However, there is a huge difference between the standard library and the MPL: where the former is aimed at storing and manipulating data at runtime, the latter's purpose is to store and manipulate types at compile-time.

The sequences proposed by the library are all based on the concept of typelist, but are implemented with several different techniques, optimized for different uses. Like runtime containers, each MPL sequence supports a given set of operations and provides some complexity guarantees. For example, only Associative Sequences (`boost::mpl::set` and `boost::mpl::map`) allows accessing an element from a key. Getting the nth element is possible for any Forward Sequences (i.e. sequences that can be iterated upon from begin to end), but it is more efficient if the sequence is a Random Access Sequence (amortized constant time instead of linear). The MPL sequences provided are vectors, lists, sets and maps; they support more or less the same operations than the corresponding runtime container, with similar complexities (with some variations due to the mandatory use of recursion in TMP).

Here is a sample code that creates an MPL sequence containing three types, add a type at the end, and define a variable whose type is the last element in the sequence.

```
typedef mpl::vector< bool, char > small_signed_types;
typedef mpl::push_back< small_types, short >::type medium_signed_types;
mpl::at_c< medium_signed_types, 2 >::type shortVariable = -42;
```

Code 37. Manipulation of an MPL vector.

Apart from the simple manipulation metafunctions (adding/removing elements, accessing an element by index or by key, getting the size of a sequence), the MPL provides many algorithms to operate on sequences. Most of them are compile-time counterparts of standard algorithms, and can be partitioned in three categories:

- Iteration algorithms, which allows traversing sequences while applying some operations on the elements. The most prominent one is `accumulate` (also known as `fold`), which computes a new type by combining all the elements in a sequence using a given binary operation.
- Querying algorithms. The library provides the ability to search in a sequence a given element or an element satisfying a given predicate, to count occurrences of some type or of types satisfying a predicate, to test if two sequences contain the same elements, etc.
- Transformation algorithms. These algorithms create a new sequence from existing ones through some transformation. One can copy a sequence, replace all occurrences of an element with another one, remove duplicate elements, sort a sequence, and so on. All these algorithms have a reverse counterpart that operates on the sequence in reverse order, which can be useful mainly for performance reasons.

Internally, all these algorithms use iterators, making them independent of the actual sequence provided. Thanks to that, it is possible to extend the library with custom sequences or custom algorithms with seamless integration.

As an example, the following code creates a sequence of types, removes all types that are not primitive, and checks the content of the resulting sequence:

```
typedef mpl::vector< int, std::string, std::complex, double > types;
typedef mpl::remove_if< types, boost::is_fundamental< mpl::_ > > result;
BOOST_MPL_ASSERT(( mpl::equal< result, mpl::vector< int, double > ));
```

Code 38. Sample use of MPL algorithms.

The removal is performed with the `remove_if` algorithm, which is given a predicate in the form of a placeholder expression. Then, we use `equal` to check that the resulting sequence contains the expected types. This allows us to introduce static assertions: which are a very useful feature, provided by several Boost libraries (including the MPL) and added in C++11. A static assertion, like its runtime counterpart, checks a condition and generates an error if it is not verified. However, instead of generating a runtime error, a static assertion triggers a compiler error, allowing early verification of some aspects of a program. In fact, the code above does not even need to be run: the simple fact that it compiles means that it was successful. Static assertions are essential in TMP because they allow enforcing invariants on classes, and preconditions/postconditions on metafunctions. Without them, the program would probably still fail to compile, but not at the best time and with a cryptic error message, pretty much like a runtime application that would not check any error case.

Numerical computation

In addition to types, C++ template metaprograms can also manipulate integral constants. This can be used to index compile-time data structures, to optimize certain computations, or to easily create types with arithmetic computations. For example, the MPL tutorial [Abrahams and Gurtovoy 2004] describes the development of a dimensional analysis library, effectively creating a type system for numbers to prevent meaningless operations such as adding a mass to an electric current[25]. This library represents dimensions as combinations of the fundamental dimensions standardized in the International System of Units (SI) (meter, kilogram, second, etc.). Thanks to template metaprogramming, invalid computations are caught at compile-time, preventing many mistakes to be made. The interesting thing is that new dimensions are created as needed: when an operation results in a quantity whose dimension is not defined yet, it is created by the library by combining the type of the operands. For instance, dividing a length by a time provokes the creation of a dimension corresponding to velocity.

[25] A complete library for static dimensional analysis is provided by Boost.Units [Boost.Units].

The MPL homogenizes number manipulation with type manipulation by providing number wrappers, which are simply types encapsulating a number. For example, the integer 42 would be represented by the type `mpl::int_< 42 >`. The actual value can be obtained through a static member variable named `value`. Wrappers are provided for the most common integral types, and a generic wrapper can be used to represent any integral value with any integral type: `mpl::integral_c< short, 42 >`.

The rationale behind these wrappers is to make it possible to reuse the data structures and algorithms developed for type manipulation. Thanks to them, an MPL sequence can be used to store integral constants without any modification. To further facilitate their integration, number wrappers are also nullary metafunctions returning themselves.

Numeric computations on number wrappers are performed through metafunctions, making it possible to pass them to higher-order metafunctions. All the common operations (addition, comparison, logical and, bit shifting, etc.) are implemented in the library, and can be overloaded for user-defined wrappers. Indeed, it is interesting to note that integral constants can be combined into more complex compile-time data types, such as complex or rational numbers for instance.

IV.3.5. Boost.Fusion

The standard library works only on runtime (dynamic) data, the MPL only on compile-time (static) data. Having pure compile-time computation is nice, but it usually needs to be transposed into the runtime world at some point.

Boost.Fusion is the link between compile-time and runtime entities. It allows mixing seamlessly static and dynamic computation. In practice, it provides several mixed data structures and algorithms. The data structures can be seen as a generalization of tuples, meaning data structures that can contain heterogeneous values (with different type), without erasing their types. A common practice is to do the compile-time type manipulation with MPL, then map them to Fusion containers for runtime processing of corresponding values.

As an example, the following code uses a Fusion `map` to associate types with their name as string.

```
namespace fusion = boost::fusion;

typedef fusion::map<
    fusion::pair< int, std::string >,
    fusion::pair< std::string, std::string >,
    fusion::pair< Foo, std::string >
> name_map_type;

name_map_type name_map(
    fusion::make_pair< int >( "int" ),
    fusion::make_pair< std::string >( "std::string" ),
    fusion::make_pair< Foo >( "Foo" ) );

std::cout << fusion::at_key< int >( name_map ); // prints "int"
```

Code 39. Sample use of a Fusion map.

The first lines define the kind of map we need by stating what content will be stored: the *type* `int` will be associated to a `string` *value*, like the types `string` and `Foo`. The next lines declare a variable to hold the values, and initialize it with the appropriate key/value pair: `int` is associated with `"int"`, `Foo` with `"Foo"` and so on. Finally, the string value associated with the key `int` is looked up using the `at_key` function, and printed.

Fusion provides most of its functions and algorithms in two versions, metafunction and function. The former is used to obtain at compile-time the type of the result, while the latter computes at runtime the actual value. For example, `result_of::at_key< name_map_type, int >::type` would return the type `std::string`, at compile-time, while `at_key< int >( name_map )` returns the string value `"int"`, at runtime.

The processing of Fusion sequences usually involves both compile-time and runtime operations. Consequently, most processing is achieved through mixed metafunctions/functions operating on both static and dynamic parameters. In practice, these are implemented as function templates, parameterized at least by the sequence type and the sequence, and possibly other template and/or non-template parameters. The `at_key` function used previously exhibits such a duality: it has two static parameters, namely the sequence type (which is automatically deduced from the dynamic parameter) and the key, and one dynamic parameter, the sequence to investigate.

In addition to the usual query operations (accessing an element, getting the size of a sequence, testing the presence of a key, etc.), Fusion provides many algorithms to process

sequences. Like in MPL, they are mainly transpositions of those included in the Standard C++ Library, and internally use iterators to be independent of the manipulated sequence. However, an interesting difference with MPL and the Standard Library is that many Fusion algorithms are evaluated lazily: instead of computing the result when invoked, they return a view on the original(s) sequence(s) which will compute elements on the fly when needed. The rationale behind this design choice is that much like MPL, Fusion algorithms are functional in nature and does not mutate their parameters. Without lazy evaluation, that would imply that each time an algorithm is called on a sequence, a new sequence containing the result would be created, generating many useless copies and degrading performances. By returning views, which are lightweight objects, these costly copies are avoided, allowing the developers to chain as many algorithms as they need without having to worry about the execution time increasing drastically.

The following example creates two sequences (a list and a vector), and then apply two algorithms to concatenate them and filter the values to retain only those that are integers:

```
auto[26] zero = fusion::make_list( 0, "zero" );
auto numbers = fusion::make_vector( 1, "one", 2, "two" );

auto numberValues = fusion::filter< int >( fusion::join( zero, numbers ) );

assert( numberValues == fusion::make_vector( 0, 1, 2 ) );
```

Code 40. Sample use of Fusion lazy algorithms.

Until the comparison in the assertion, no actual computation is done. The call to `join` returns a `joint_view` that simply stores references to `zero` and `numbers`. This view is then passed to `filter`, which simply returns a `filter_view`. The actual concatenation and filtering happens while iterating over the view to perform the comparison with the expected result.

A last interesting feature provided by Fusion is its powerful extension mechanism, which permits the integration of third-party and user-defined types. The library natively supports std::pair, tuples, boost::array and MPL sequences as Fusion-compliant sequences, and allows developers to integrate their own data structures by specializing a few operations and

[26] `auto` is a C++11 keyword that allows one to omit the type of a variable: the type is inferred by the compiler based on the initialization value. We used it here to make the code shorter and more readable.

providing suitable iterators. Thanks to that, the scope of application of Fusion algorithms is not limited to the sequences provided by the library, they can be used with any types fulfilling the few requirements described by the extension mechanism. Moreover, all Fusion-compliant sequences can coexist and work in conjunction seamlessly, as all algorithms only rely on iterators.

V. Conclusion

In this chapter, we described the metaprogramming paradigm. The main idea of metaprogramming is to write programs that manipulate or create other programs. We explained how this approach could be used to develop applications that are both generic and highly efficient. Then, we surveyed several metaprogramming techniques: simple text generation, domain-specific languages, multi-stage programming and partial evaluation. We provided a comprehensive comparison of these various approaches, according to a collection of criteria.

Finally, we described in much detail C++ Template Metaprogramming (C++ TMP), as well as the various libraries available to smooth the application of this technique. C++ TMP leverages templates, a feature of C++ originally designed to allow generic programming, to perform computations at compile-time. These computations are carried out by the compiler during template instantiation, and usually deal with types, even though numeric values can also be manipulated. C++ TMP supports only a few features, making it very different from C++ programming; in fact, it is closer to functional programming, with which it shares many characteristics: lack of mutable data, intensive use of recursion, pattern matching, etc.

In Chapter 6, we apply C++ template metaprogramming to DEVS simulation to improve both performance and model verification. We achieve this by making the compiler perform many simulation operations at compile-time, effectively resulting in a simulator that is specialized for the DEVS model to be simulated. Before presenting this work, we return to Model-Driven Engineering in the next chapter, which is dedicated to the MDE environment for DEVS modeling that we developed.

Chapter 5. SimStudio, a Model-Driven DEVS Environment

I. Introduction

In Chapter 2.III, we pinpointed tool interoperability as one of the major challenges faced by the DEVS M&S community. Indeed, due to the ever-growing number of DEVS environments available, the collaboration between DEVS modelers has become harder and harder. One approach to solve this issue is co-simulation, a methodology where heterogeneous tools, possibly distributed, cooperate to simulate a model. Co-simulation has been the subject of many works, and is notably implemented in the High-Level Architecture (HLA) [Kuhl et al. 1999], the Department of Defense architecture for distributed simulation.

An alternative approach is to focus on models instead of focusing on simulators. A first possibility is to come up with a uniform interface for interrogating distributed models, allowing their integration in various simulation tools. To our knowledge, this idea has not been studied yet; it would probably be interesting to investigate it further.

However, the solution we decided to explore is different: we chose to work on unifying the representation of models, with a format independent of all simulators and easily

manipulable by computer software. This independent representation can be processed to automatically generate representations specific to each M&S tool, but this is not its sole purpose: using a well-defined and high-level format makes it easier to define all sorts of processing over models, such as static analysis, visualization, refactoring, etc.

To apply this MDE approach to DEVS M&S, we started developing XML Schemas and XSLT transformations [Touraille et al. 2009a] before adopting a framework more suited to this kind of development. We opted for the Eclipse Modeling Project, presented in the previous chapter, which is composed of multiple projects providing MDE languages and tools, based on the Eclipse platform.

Our ultimate aim is to integrate this approach in an M&S environment named SimStudio, which will serve either as an autonomous tool or as an overlay over existing software, by providing bridges between implementations through automated model transformations.

We will start this chapter by outlining SimStudio and it objectives. Then, we will describe the different elements we developed to implement this MDE approach: a DEVS metamodel, various model-to-model transformations, code generators, etc. Finally, we will present a simple application of our proposition, using the features actually implemented.

II. Application of Model-Driven Engineering to DEVS M&S

II.1. SimStudio, an extensible and integrating environment

Our goal in this chapter is to propose an approach for DEVS M&S relying on MDE to facilitate creation of new features, ease model implementation and help integrating heterogeneous tools.

At the core of this approach is a unified format for representing DEVS models. Indeed, DEVS models are currently developed using tool-specific representations, such as Java classes in DEVSJava, C++ classes and “.ma” files in CD++, etc. As a consequence, it is difficult to develop new tools for manipulating DEVS models that could be adopted by a wide audience, and the exchange of models between scientists, for instance for reproducing experiments or for leveraging existing research, is greatly hindered.

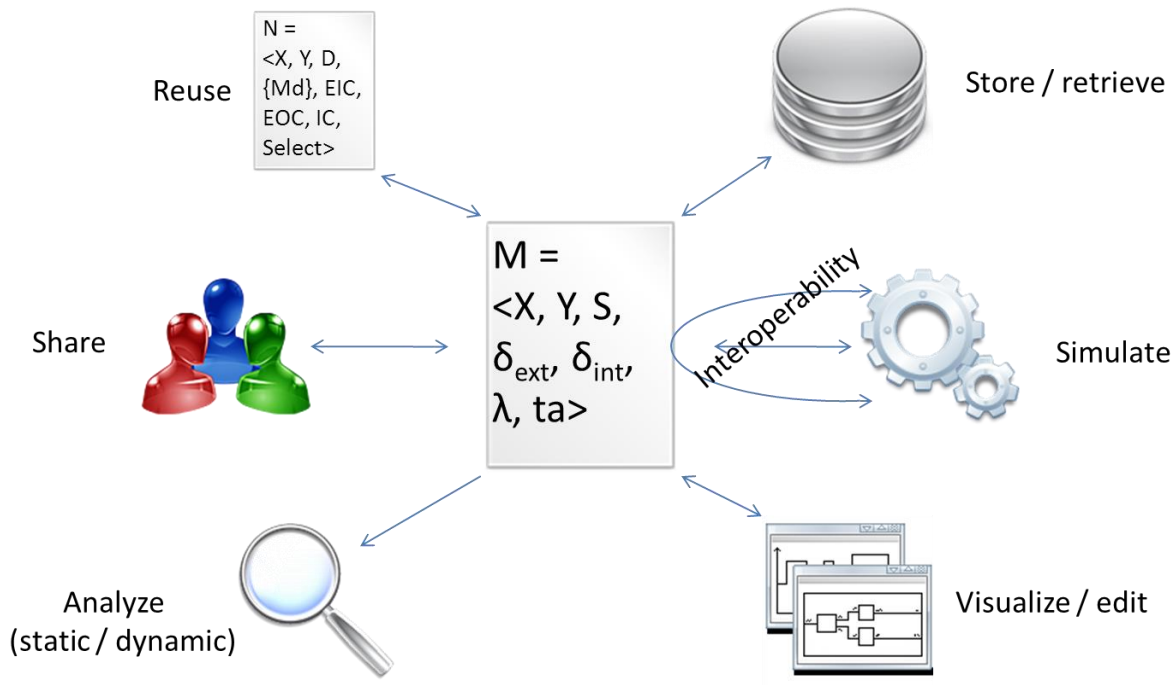


Figure 20. Various use cases for a DEVS model.

A solution to improve this situation is to devise a standard format for DEVS models. This format should allow the description of most model characteristics, in order to be usable for most use cases: design of the model, simulation, study of the results, analysis, etc. Figure 20 depicts various usages of a DEVS model that should be supported by a DEVS standard.

To experiment how such a standard would improve DEVS M&S, we developed our own platform-independent format, which could provide a basis for the DEVS Standardization Group. To do so, we leveraged the metamodeling capabilities provided by MDE environments, which allowed us to design a DEVS metamodel, but also to automate many operations over models, such as code generation or model verification.

Eventually, we aim at integrating this format and the various manipulations developed into a DEVS M&S framework named SimStudio [Traoré 2008] [Touraille et al. 2011]. The goal of SimStudio is to provide an integrating framework where theoretical advances in M&S could be concretized efficiently and brought to the community. To achieve this, SimStudio would rely on an extensible architecture, where new features could be plugged-in very easily and interoperate with others thanks to the pivot format. Figure 21 sketches this architecture, and shows how the various modules can be structured according to several axes:

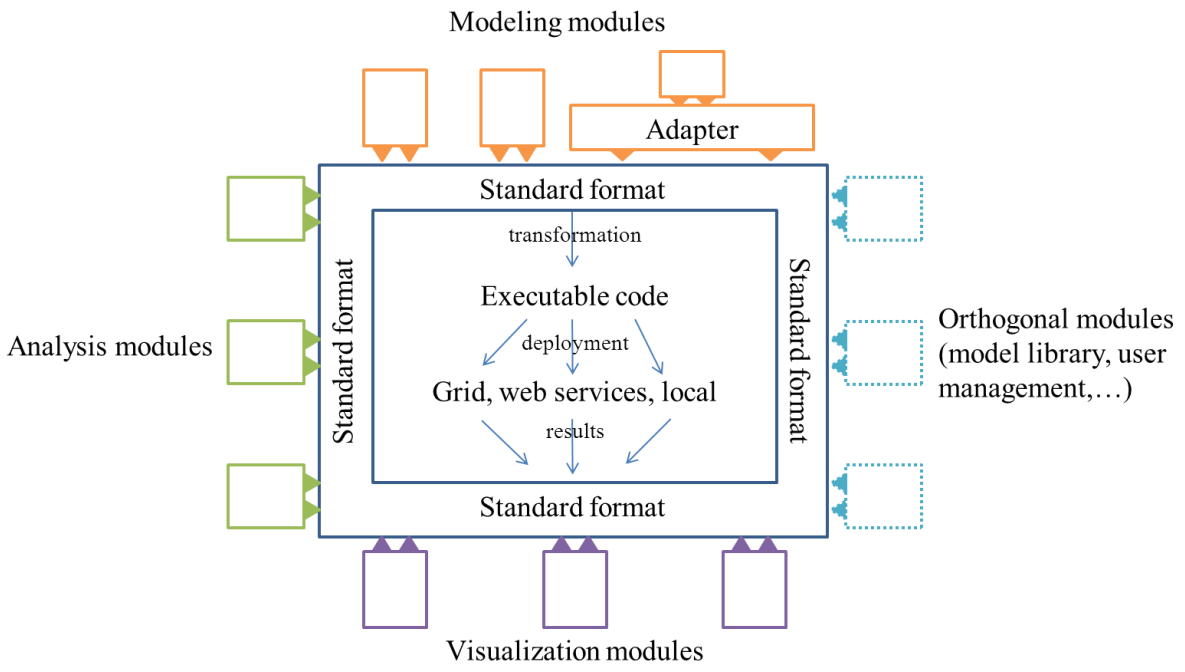


Figure 21. Overview of SimStudio.

- Modeling modules, which provide textual and graphical editors for creating and editing models. Adapter modules can also be developed to accommodate model specifications coming from external tools.
- Simulation modules, responsible for simulating models on various platforms either by interpreting their standard representation or by using code generation to transform the platform-independent representation into executable artifacts. Ideally, a wide range of target platforms should be supported in a transparent way, notably high-performance computing architectures such as clusters or grids.
- Visualization modules, providing features for displaying results in various forms: listings, diagrams, animations, etc.
- Analysis modules, implementing formal analysis methods for determining properties of the model based on its static representation, such as its complexity or its structural validity. The dynamic behavior of models can also be studied through modules analyzing simulation results, for instance by computing statistics.
- Managing modules, offering orthogonal services such as model repositories, collaborative tools, workspace customization, etc.

SimStudio is developed keeping in mind that many DEVS tools are already available, and should be capitalized on. Therefore, great care is taken to provide as much integration as possible with other tools, mainly through automatic transformations to and from the formats they use. Since the architecture of SimStudio relies on an extensible system, one could also imagine integrating existing tools into the environment by wrapping them into plug-ins. Thus, SimStudio positions itself as a complement to existing software, or as an overlay providing additional features. For instance, a model could be designed using SimStudio, but deployed to an external tool for simulation. The results could then be integrated back into SimStudio for visualization or analysis, for example.

To make this integration possible, it is essential to come up with a platform-independent format for representing DEVS-related data, which would accommodate most implementations and cover most use cases of an M&S development. We will now present our attempt at developing such a format, in the form of a DEVS metamodel.

II.2. DEVS metamodel

Basically, a DEVS model is composed of two parts: a structural one, dealing with state variables, ports, components, connections, etc., and a behavioral one, describing the temporal evolution of models thanks to various functions (transition functions, time advance function, etc.). These two aspects will have to be handled very differently in our metamodel. Indeed, metamodels are very well-suited to specify structural features, but representing behavior without depending on a particular platform or programming language is much more difficult. Before exposing how we tackled this issue, let us start by explaining the simplest aspect, which is metamodeling the structure of DEVS models.

II.2.1. Structure

Metamodeling the structure of DEVS models is quite straightforward for the most part, but some particular points deserve more attention, like the representation of user-defined types for instance. Another aspect worthy of interest is model verification. Indeed, metamodeling frameworks usually include validation[27] features, allowing a toolsmith to specify invariant

[27] The term “validation” has a different meaning in MDE and in M&S. In MDE, “model validation” refers to checking the consistency of a model with regard to its metamodel and the constraints it defines. The term was already used with this meaning in many technical domains such as formal language theory or structured

constraints over metamodels. In our metamodel, we define such invariants using OCL annotations over model elements [Damus 2007]. These annotations are processed by EMF to generate validation code that can be automatically invoked during the creation of DEVS models by a practitioner, generating errors if some constraints are not respected. This way, the modeler is warned early in the development cycle that his model has some inconsistencies, and can correct it right away.

In the following, we will expose the various elements of our metamodel that represent the structure of DEVS models, along with the constraints we specified for them. We will proceed from the ground up, presenting first the lower-level constructs that will serve as building blocks for the higher-level ones.

II.2.1.1. Types

A first thing to determine is how to represent types used in DEVS models. Indeed, in order to allow the generation of a fully functional code and verifications over models, it is important that some entities be associated with a type. In fact, types are needed wherever user-defined sets are used in the formalism, i.e. to specify the input and output sets of models and the state of atomic models.

Several solutions can be considered; the first one is to restrict the types usable by the modeler to a set of predefined/built-in types, such as string or integer. Of course, this solution is inconceivable as it tremendously limits the freedom of the modeler, making it impractical if not impossible to model “real-world” systems. To overcome this limitation, we could define a metamodel for types, allowing the modeler to define custom types. Such a metamodel should support at least compound data, but could also include aggregation, inheritance, enumeration, etc. In addition, it would be practical to provide concrete syntaxes to facilitate the specification of new types.

In the end, this metamodel would strongly resemble already existing metametamodels such as Ecore or XML schema. Consequently, instead of reinventing the wheel, it is much better to

document processing, but is used differently in the domain of Verification & Validation, where “validation” refers to determining if a model accurately represents the system under study according to the objectives set. In fact, “validation” in MDE is more related to V&V “verification”, i.e. the process of checking that an implementation satisfies initial requirements and specifications.

leverage one of these metametamodels, especially to benefit from their surrounding tooling like code generators and editors. In our case, the obvious choice is to rely on Ecore since we work in EMF. Therefore, our metamodel associates typed entities (ports and state variables) with an EClassifier, an element of the Ecore metametamodel representing types (see Figure 12, page 101). A classifier denotes a set of objects (instances) sharing the same characteristics. It can be a simple data type (EDataType), such as a number or an enumeration, or a full-fledged class (EClass), with attributes, references to other objects, and possibly some behavior (operations/methods) and constraints (invariants).

Using the EClassifier element from the Ecore metametamodel in our DEVS metamodel provides several advantages. First of all, it brings support to all the built-in types such as EInt, EBoolean, EString, EDate, etc. A practitioner can use these predefined types when designing his models without any additional work. Next, it provides a precisely defined metamodel for creating new types. If the built-in types are not sufficient to represent some aspects of the system under study, it is possible to create new types simply by creating a new EClassifier. Since we use an element of Ecore, we can leverage all the tools available for creating and editing Ecore models, such as graphical and textual editors. In addition, we can also take advantage of the code generation features provided by EMF: out of the box, user-defined types can be transformed into Java code by the EMF generator, and other target languages can easily be supported by using one of the numerous generators available (C++[28] [González et al. 2010], Python[29], C# (work in progress)[30], etc.). As a last resort, it is always possible to add support to a language by developing a simple Model-to-Text transformation for generating code from a classifier.

II.2.1.2. Ports

Now that we determined how user-defined types are represented in our DEVS metamodel, we can very simply specify the notion of *port*. A port has a name, denoted by a string attribute, and two references: one to its type (EClassifier), and one to the model it belongs to. We chose to use a bidirectional association between models and ports merely as a convenience, in order to simplify the definition of constraints.

[28] http://www.catedrasaes.org/trac/wiki/EMF4CPP (last accessed 21/06/2012)
[29] http://www.acceleo.org/pages/module-ecore-vers-python/en (last accessed 21/06/2012)
[30] http://wiki.eclipse.org/EMF4Net_Proposal (last accessed 21/06/2012)

We distinguish input and output ports by defining two subclasses. Those do not add any data or behavior, but will be useful when specifying couplings. Thanks to this differentiation, we will be able to assert structurally that EIC couplings must connect input ports to input ports, IC output ports to input ports, and EOC output ports to output ports.

The diagram in Figure 22 summarizes how we specify ports in our metamodel:

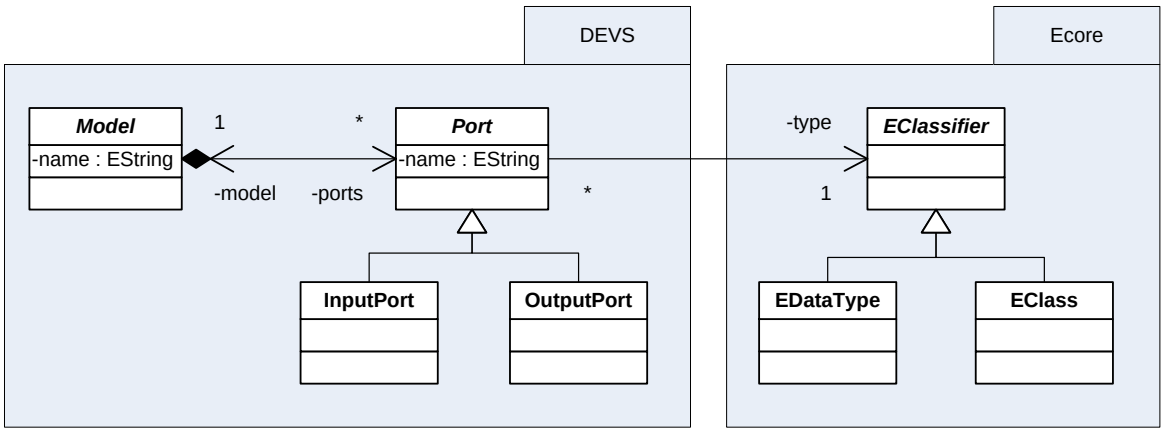


Figure 22. DEVS metamodel – Ports.

II.2.1.3. State variables

To specify the state of an atomic model, it is often easier to reason in terms of state variables, meaning that the state of the model, instead of being a single value, is split in a collection of variables (see multivariable sets in [Zeigler et al. 2000]). This makes the definition of the state set much clearer, as the various elements composing the state can be named meaningfully. Similarly, querying and modifying the state is easier when we can refer to a particular constituent by name.

Consequently, we define in our metamodel the notion of state variable, which is a simple element with a name and a type (EClassifier), as can be seen in Figure 23.

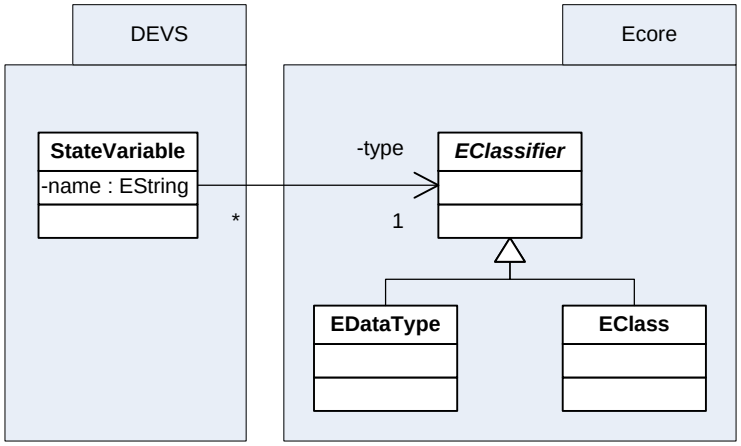


Figure 23. DEVS metamodel – State variables.

II.2.1.4. Atomic models

With ports and state variables, we have enough elements to define atomic models. Indeed, these are the only structural features needed to specify atomic models. The dynamic behavior will be handled specifically in Chapter 5.II.2.2.

Both coupled and atomic models are named elements associated with multiple ports. To factorize these commonalities, we use an abstract base class Model extended by both AtomicModel and CoupledModel. In addition to these features, atomic models also include a collection of state variables, as Figure 24 shows.

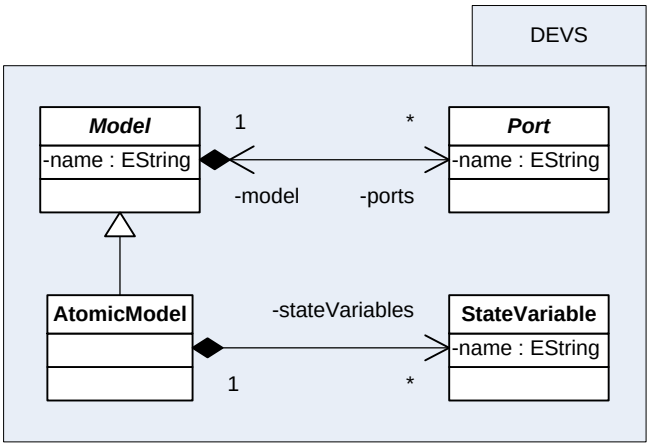


Figure 24. DEVS metamodel – Atomic models.

These relations between the various metamodel elements are not sufficient to ensure the correctness of models. Indeed, we would like to assert some additional constraints, such as

the uniqueness of port names in a given model. As we said before, we can use metadata to specify these invariants, by annotating elements of the metamodel with OCL expressions. As an example, Code 41 shows the specification in OCLinEcore[31] of the port names uniqueness invariant, defined over the Model class.

```
invariant PortNamesUniqueness:
    ports->forAll(p1, p2 : Port | p1 <> p2 implies p1.name <> p2.name);
```

Code 41. Port names uniqueness invariant in OCLinEcore.

In a similar way, we define an invariant over AtomicModel to assert that all state variables must have a unique name.

II.2.1.5. Components

A coupled model embeds a set of components, which are instances of other models. A component is instantiated either from an atomic or from a coupled model, allowing arbitrary levels of nesting. To be referred to, it must be identified by a name, allowing multiple instances of the same model to coexist in a given coupled model.

An interesting issue regarding components is that of parameterization. Indeed, it is often useful to parameterize models in order to make them more easily reusable. Consequently, a model instance (a component) should specify what values those parameters should take. The DEVS formalism does not address this issue, but we think this aspect should be precisely specified. When working on the XML version of our DEVS metamodel [Touraille et al. 2009], we initiated a reflection on this subject; however, we did not include the outcome of this reflection in this section, but we provide in Chapter 5.II.2.3.1 an overview of the approach we consider using.

Leaving aside the issue of parameterization, a component is thus a simple named element associated with a model, as depicted in Figure 25. We also include in the component an opposite reference to the coupled model that includes it, once again to facilitate some processing.

[31] http://wiki.eclipse.org/MDT/OCLinEcore (last accessed 21/06/2012)

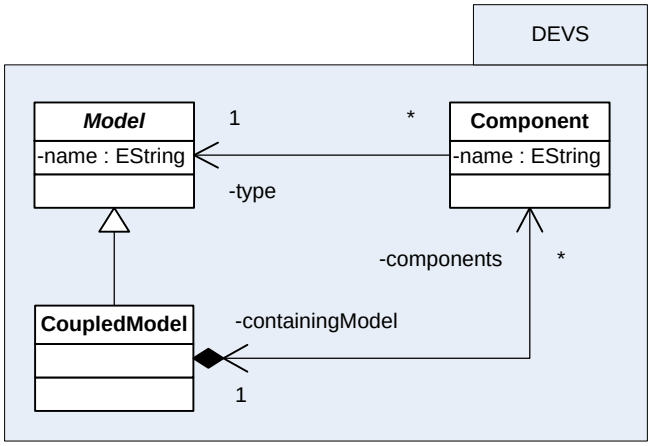


Figure 25. DEVS metamodel – Components.

II.2.1.6. Couplings

To specify the flow of events, coupled models define couplings between components. Three different types of couplings can be distinguished: external input coupling (EIC), internal coupling (IC) and external output coupling (EOC).

EIC connect input ports of the coupled model with input ports of the components; IC connect output ports of the components with input ports of other components; EOC connect output ports of components with output ports of the coupled model. As we see, each coupling is composed of a source and a destination. However, their specification differs with the type of coupling. For EIC, the source is simply an input port, and the destination is a pair composed of a component and an input port. For IC, the source is a pair component/output port and the destination a pair component/input port. Finally, the source of an EOC is a pair component/output port and the destination a single output port.

Since our metamodel distinguishes input and output ports, these various definitions can be specified structurally, as shown in Figure 26. We make all types of couplings inherit from a common base class merely as a convenience, so that a coupled model can simply store a set of couplings without caring about their actual kind.

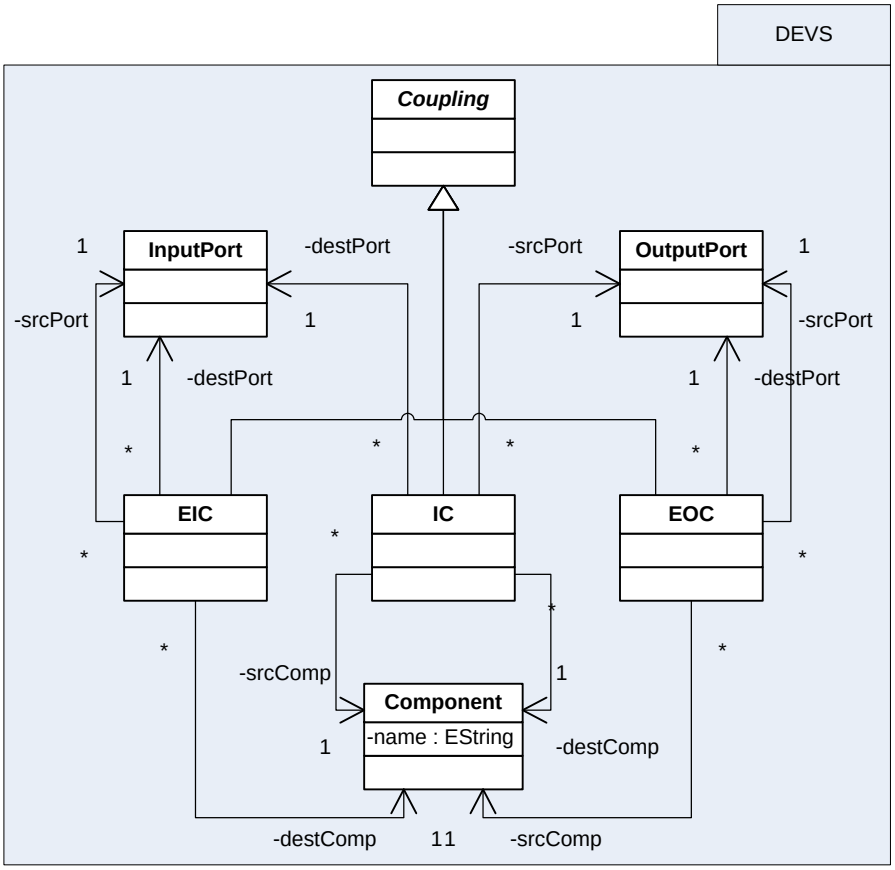


Figure 26. DEVS metamodel – Couplings.

Once again, the metamodel structure does not enforce all the requirements that couplings must fulfill to be valid. Therefore, we rely on invariants to specify these requirements, thus ensuring the integrity of instance DEVS models.

We defined the following constraints:

- in each coupling, the source port must belong to the model of the source component (or to the coupled model in the case of EIC), and the destination port must belong to the model of the destination component (or to the coupled model in the case of EOC);
- internal couplings must not define direct feedback loop, i.e. the destination component must not be the same as the source component;
- source and destination ports must be compatible, meaning that they must have the same type.

The latter invariant deserves more attention. Forcing connected ports to have the same type allow detecting inconsistencies such as connecting a port sending Missile values to a port accepting Puppy values, but it is also quite restrictive. Most notably, it does not take into account the notion of inheritance/interface implementation, nor the one of many implicit type conversion. In order to make reusable models, we should be allowed to send instances of derived classes to ports awaiting base class instances, or to send integers when double are expected.

To relax this constraint, we use an EClass operation that tests whether the class is a subtype of some other class. This way, we can assert that the type of the destination port must be a super type of the source port (or the same type). We found no way to account for type conversions in our metamodel, but we will see in Chapter 6 a way to leverage existing type systems to perform this kind of verification for us, in this case the C++ type system.

II.2.1.7. Select

Regarding the tie-breaking function, we made the choice to represent it in a static way rather than relying on some dynamic behavior of the model. Indeed, since the set of components is specified in the coupled model structure, it is possible to totally define the select function without needing any dynamic computation. As a reminder, the tie-breaking function is used when several components have an internal transition scheduled at the same time: the select function is invoked to determine which of these components should be activated in priority.

A basic approach would be to make the modeler enumerate all possible combinations of components, along with the component to activate in each case. However, that would be very tedious, especially as the number of components grows. Consequently, we tried to find a more practical way to specify the select function. The solution we retained consists in using a sequence of so-called “selection rules”. A selection rule simply associates a set of components (a subset of the component set) with one prior component, which must be in the set:

$$rule = (C, p), C \subseteq D, p \in C$$

The meaning of a rule (C, p) is as follows: when the set of imminent components contains all the elements of *C*, then *p* must be activated. However, it is possible to have several rules conflicting with each other. To solve this, we order the rules and decree that the first rule that matches the imminent set has priority over the subsequent rules. Therefore, the select function is defined through a sequence of rules:

$$selectRules = \big((C_0, p_0), \dots, (C_n, p_n)\big)$$

To know which component must be activated given a set of imminent components, we find the first rule whose set is included in the set of imminent, i.e. the smallest *i* such that $C_i \subseteq imminentSet$. The component to activate is the one specified in the rule, i.e. p_i.

Using such rules, it is possible to describe every case in a simple way. For instance, the basic case where components are prioritized according to a strict total order can be represented through simple rules where the sets have a single element. For instance, consider a coupled model with three components ordered as follows: c1 < c2 < c3 (c1 has priority over c2 and c3, c2 has priority over c3); the select function of this coupled model can be represented easily by the following sequence of rules:

$$selectRules = \big((\{c_1\}, c_1), (\{c_2\}, c_2)\big)$$

With these rules, if c1 and c2 are imminent, c1 will be selected as {c1} is the first set in the sequence to be a subset of {c1, c2}; if c2 and c3 are imminent, c2 will be selected, as {c1} is not a subset of {c2, c3} but {c2} is.

Now, assume that we need a special case: when all components are imminent, we want c3 to be selected. To specify this, we just need to add a rule at the beginning of the select sequence, as follows:

$$selectRules = \big((\{c_1, c_2, c_3\}, c_3), (\{c_1\}, c_1), (\{c_2\}, c_2)\big)$$

With this additional rule, c3 will be selected whenever all three components are imminent. The other cases do not change: for instance, if c1 and c3 are imminent, the second rule will be considered and c1 selected.

It is important to note that the order of the rules is essential: if we change the position of the first and second rules, the last rule we introduced will never be applied, and c1 will be selected instead of c3 when the three components are imminent. Therefore, it is important to specify in our metamodel that the select rules must be stored as a sequence and not as a set. Figure 27 summarizes the structure retained for the representation of the tie-breaking function.

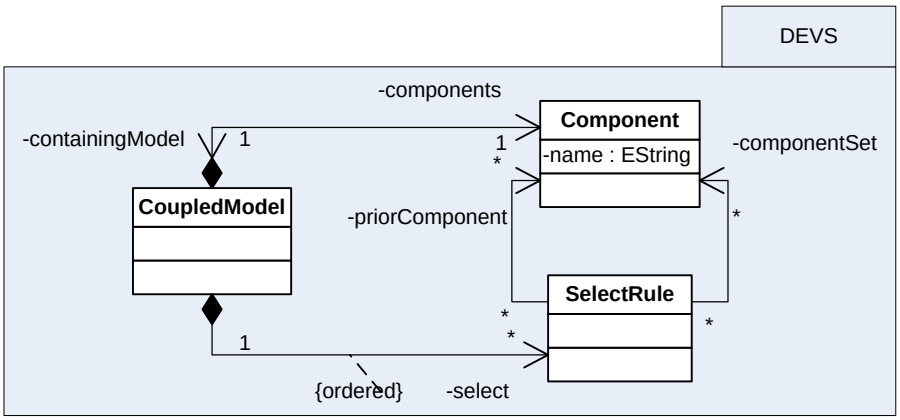


Figure 27. DEVS metamodel – Tie-breaking function (select).

To check for inconsistencies, we defined two invariants: the prior component in a rule must be present in the component set of the rule, and all the components used in the select sequence of a coupled model must belong to this model.

II.2.1.8. CoupledModel

With all these building blocks defined, specifying the structure of coupled model is straightforward. It boils down to aggregating all the needed elements in a single classifier: as show in Figure 28, a coupled model is simply a type of model composed of components, couplings and a sequence of select rules.

In addition to the constraint on select rules we already evoked, the coupled model element defines a constraint on component names, to assert their uniqueness.

The full definition of our DEVS metamodel is provided in textual format in Appendix A, and as a diagram in Appendix B.

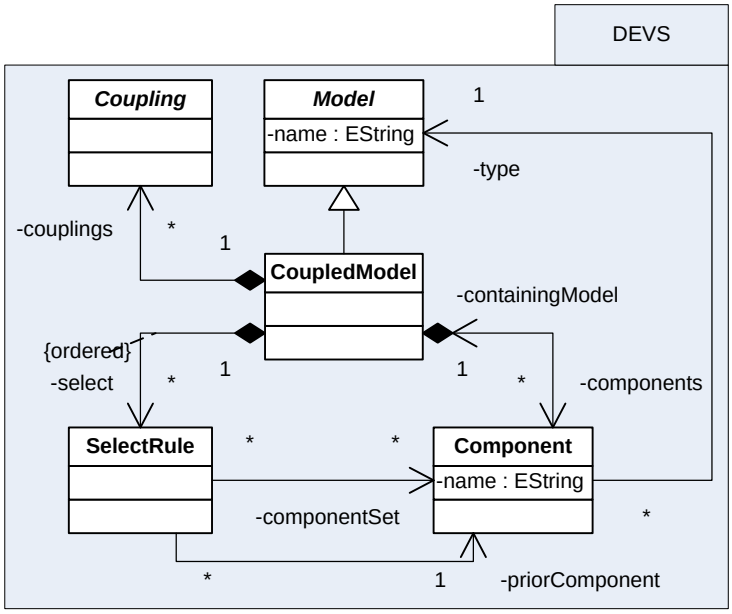


Figure 28. DEVS metamodel – Coupled models.

II.2.2. Behavior

We just saw how to represent the structure of atomic and coupled DEVS models. However, if a structural specification is sufficient for coupled models, it has to be completed to describe atomic models. Indeed, atomic models have behavior, specified through various functions (time advance, transitions, output). Therefore, our metamodel must provide a way to represent this behavior, without sacrificing its platform-independence.

II.2.2.1. Previous propositions

Previous works on DEVS standardization and interoperability have proposed solutions to the issue of behavior representation. In [Risco-Martín et al. 2007], where the authors present an XML DEVS metamodel, the various functions of atomic models are specified with simple constructs such as assignments and conditionals. The authors themselves acknowledge the fact that such constructs are too simple to adequately express a wide range of models; as a consequence, in [Mittal et al. 2007], they proceed to use JavaML, an XML Schema for describing Java code as XML. Doing so, they take advantage of the full power of the Java language, notably its wide standard library, but they also become tied to a particular programming language, which goes against our initial goal.

The approach adopted in [Janoušek et al. 2006] is more promising: in this work, the authors propose to use a simple Lisp-like language for specifying the functions, which is then mapped to an XML representation inspired by JavaML. Even though we agree that this is the direction to take, we think that such a language lacks a crucial feature to make it truly usable in actual M&S projects: libraries. Indeed, "real-world" models usually need to perform operations more complex than simple assignments, loops or conditionals; for instance, they might need to use some data structures, read files, draw random numbers or even interrogate databases and connect to servers. If these kinds of operations cannot be described in our metamodel, there is little chance of it getting widely adopted except for the simplest cases.

The issue of describing operations arises in almost all model-driven development. As early as in the late 1980s, the Shlaer-Mellor Method, an ancestor of the current MDA, advocated the separation between platform-independent and platform-specific models, and proposed a method to represent behavior directly into models so that implementation code could be fully generated [Shlaer and Mellor 1991]. Their solution was to represent object lifecycles using State Models (as finite state machines), and processing with Action Specifications: each state of an object was associated with some processing, described with a flowchart or a proper Action Language.

The Shlaer-Mellor method has been adapted to UML in Executable UML (xUML) [Mellor and Balcer 2002], and several action languages have been developed – but not standardized – such as the Object Action Language (OAL) or the Action Specification Language (ASL). These languages are very high-level languages aimed at describing actions on the model elements, such as creating objects, defining associations between them, or computing attribute values. These actions can be interpreted to actually *execute* the model, or can be transformed into code in a target programming language. However, these languages suffer from the same problems as the LISP-like one evoked above. They operate on a very high level, very close to the model, making it impractical to perform common operations. As only model elements can be manipulated, the slightest action that does not directly relate to the domain entities implies extending the model with a bunch of classes totally unrelated to the domain, making it impossible to adopt the usual layering approach where high-level abstractions are built on top of lower ones.

A solution to this issue would be to do in modeling as we did in programming, namely developing model libraries that would provide well-defined abstractions for use in domain models. These libraries would provide the same kind of features as those usually found in general-purpose programming languages, but as platform-independent models along with mappings to various platforms. For instance, there could be mappings for data structures, I/O operations, networking, database access, etc. To our knowledge, such libraries do not exist yet, making action languages not entirely fitting our needs.

As a consequence, we decided to come up with our own language, a "semi-generic language" that aims at being as generic as possible while still leaving the possibility to leverage all the lower-level libraries available in most programming languages.

II.2.2.2. Proposal of a "semi-generic" language

Before exposing our approach, it is good to summarize the properties that we want for our language. The most important one is of course platform-independence: we need to be able to map the language to the most prominent programming-languages such as Java, C++, C# or Python. However, we also want the language to allow using common APIs available and similar across platforms. For example, we want the language to support I/O, threading or XML processing without needing to develop libraries that are readily available in all target GPPLs. Ideally, we would also like to support as much reverse-engineering as possible, in order to enable the integration of existing DEVS model into our model-driven environment. Finally, the language should stay simple and high-level enough to be directly used by practitioners.

To fulfill all these requirements, we devised a language composed of generic expressions, directly mappable into GPPLs, and platform-specific snippets, for which the mappings must be explicitly specified somewhere. The core of the language only supports a restricted set of features, meant to be the common denominator between most imperative languages: variable declaration and initialization, assignment, arithmetic operations, basic control structures (loops, conditional statements), etc. The idea is to be able to automatically transform code written with this core into any imperative language.

If the functions of an atomic model can be described simply with such generic code, all is well and good. However, we already made quite clear that such a language is much too limited to efficiently represent the full range of possible behaviors. To allow the modeler to use more advanced features, especially APIs, we give the possibility to embed into generic code some “un-generic” code, whose actual content will depend on the target platform. Such platform-dependent code snippets are identified by a name, and can accept some parameters that will be used in the generated code. Basically, the language includes a system of macros to abstract away lower-level code.

As an example, consider Code 42, written in our semi-generic language, which could belong to some transition function of an atomic model.

```
#UniformRealGenerator(gen, 0, 1, seed)#
foreach (Entity curEntity in entities)
{
    if (#UniformRealGenerator.next(gen)# < 0.5)
    {
        curEntity.state = ! curEntity.state;
    }
}
```

Code 42. Sample semi-generic code.

This code is composed of generic parts, like the foreach and the comparison, and of platform-dependent parts, related in this case to random number generation. Two snippets are used: one for declaring the random number generator, and one for getting the next random value. We specify the name of the generator variable and the target range by providing them as arguments. Code 43 and Code 44 show how this code would map into respectively C++ and C#; note how the correspondence is straightforward for generic parts, but much less for platform-dependent ones.

```
std::uniform_real_distribution<> gen(0, 1);
std::mt19937 __gen_engine__(seed);
for (Entity & curEntity : entities) // range-based for loop (C++11 feature)
{
    if (gen(__gen_engine__) < 0.5 )
    {
        curEntity.state = ! curEntity.state;
    }
}
```

Code 43. Sample semi-generic code mapped to C++.

```
NPack.MersenneTwister gen = new NPack.MersenneTwister(seed);
double __genMin__ = 0;
double __genMax__ = 1;
foreach (Entity entity in entities)
{
    if ((gen.NextDouble() * (__genMax__ - __genMin__) + __genMin__) < 0.5)
    {
        curEntity.state = ! curEntity.state;
    }
}
```

Code 44. Sample semi-generic code mapped to C#.

The advantage of this approach, in addition to its simplicity, is that it accounts for differences between target simulators as well. Indeed, two DEVS environments, even if using the same programming language, may use different ways of doing certain operations. For example, accessing a value on a port is done differently in DEVSJava and in James II, even though they are both written in Java. Simulator-specific snippets can be used to abstract these differences, allowing many simulation environments to be targeted for generation. As a consequence, the term “platform” represents in our case both programming languages and simulation tools. We can even add libraries to the notion of platform: in a given language, various APIs can often be used for the same purpose. For instance, the uniform random generator from the previous example can be mapped in Java to the standard `java.util.Random`, or to the more advanced `umontreal.iro.lecuyer.rng.MT19937` from the Stochastic Simulation in Java (SSJ) library [L’Ecuyer et al. 2002].

Another asset of this approach is that it facilitates reverse-engineering. When transforming a DEVS model written for a specific environment into a platform-independent model, we can stumble on code that cannot be directly mapped to our semi-generic language. In this case,

the transformation can include in the target code a platform-dependent snippet, indicating that it failed to transform this part of the code. It can also create the mapping corresponding to the source platform, since it can be deduced from the source model. The modeler can then review the generated model to adjust the semi-generic code if need be, or to create mappings for other target platforms.

The content of functions of atomic models is not the only place where platform-dependent types can be needed. Recall that our metamodel allows specifying the type of state variables and ports by defining the model of these types, as instances of some Ecore class. However, this solution is not very practical when the type is already available in most platforms: what is the point of redefining it when we could reuse those existing? To allow platform-dependent types to be used in the structure of DEVS models, we could extend EClassifier with our own classifier, which would merely contain an identifier and some parameters, like the code snippets we presented above.

Until now, we did not talk about how the mapping is effectively achieved. A first approach would be to use a code generation language such as Xpand to define a transformation for each abstract snippet and each target platform. This solution is powerful and very flexible, but is probably a bit overkill and makes the creation of new platform-dependent snippets too complex and cumbersome. We propose a simpler approach where models are used to store the mappings. These models associate some snippet identifiers with their corresponding implementation, using textual templates possibly containing placeholders for the arguments. These mapping models could be stored in model repositories/libraries, progressively increasing the number of APIs directly available to the practitioner.

In the end, the transformation of a DEVS model to a specific DEVS environment involves a succession of refinements, each step producing a more specific version of the model. The first stage consists in transforming the body of functions into a specific programming language. Only the platform-independent parts of the functions are transformed, using generic transformations defined for each target programming languages. Then, the platform-dependent snippets must be handled. To do so, a solution is to have a transformation that takes several parameters: the DEVS model, the mapping model for the target simulation environment, and the mapping models for the target APIs. Another

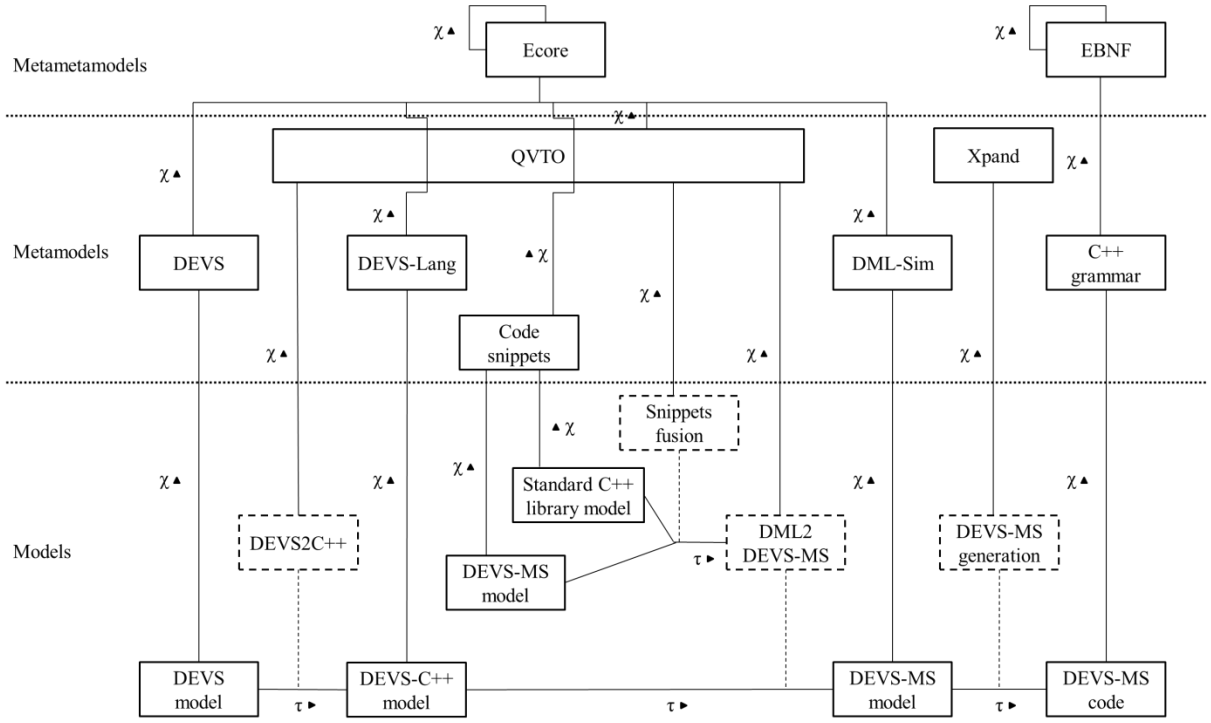


Figure 29. Megamodel of the generation to DEVS-MS.

solution is to use a higher-level transformation to automatically generate a model transformation from a set of mapping models. The generated transformation can then be applied to the DEVS model to obtain a DEVS model entirely specific to the target simulation environment and target libraries. Finally, actual code needs to be generated with a Model to Text (M2T) transformation, for integration into the target environment. Figure 29 gives an example of such workflow, showing the various elements needed to transform a DEVS model described with our metamodel to the same model in DEVS-MS[32]. In this megamodel, we use the notations proposed by Favre [Favre et al. 2006], where χ represents a relation "conforms to" and τ a relation "is transformed into". Besides, we denote transformation models by dashed rectangles. They are connected by a dotted line to the relation "is transformed into" where they are used.

To implement this semi-generic language, we relied on XText to define its grammar and the corresponding metamodel. The core of the language is based on C, to be as close as possible to the most mainstream languages. We extended this core with some grammar rules

[32] DEVS-MS is the C++ simulator we will present in Chapter 6.

```
grammar fr.isima.limos.devs.Code              // name of the grammar
    with org.eclipse.xtext.common.Terminals   // reuse existing grammar

// infer metamodel "code" from this grammar
generate code http://www.isima.fr/limos/devs/Code

// use existing metamodel
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

// From the C grammar (http://www.lysator.liu.se/c/ANSI-C-grammar-y.html)

Code:
    ( statements += Statement )*;

Statement:
    CompoundStatement |
    ExpressionStatement |
    SelectionStatement |
    IterationStatement |
    JumpStatement |
    LabeledStatement |
    DeclarationStatement // From C++
    ;

CompoundStatement:
    '{' Statement* '}';

ExpressionStatement:
    Expression? ';';

SelectionStatement:
    'if' '(' Expression ')' Statement ( 'else' Statement )? |
    'switch' '(' Expression ')' Statement
    ;

[...]

AssignmentExpression:
    '#' DeclaratorId ('('  ')' | '(' ArgumentExpressionList ')') '#' |
    ConditionalExpression |
    UnaryExpression ( '=' | '*=' | '/=' | '%=' | '+=' | '-=' | '<<=' |
'>>=' | '&=' | '^=' | '|=' ) AssignmentExpression
    ;
```

Code 45. Fragment of the grammar of our "semi-generic" language.

accounting for platform-dependent snippets. Code 45 shows a small fragment of the XText file defining the grammar of the language.

The grammar is defined with a notation similar to the Extended Backus-Naur Form, as a set of terminal symbols along with production rules for combining them. For instance, the

second production rule in the grammar above specifies that a statement can be a compound statement, or an expression statement, or a selection statement, etc. Each of these alternatives corresponds to some other rule defined elsewhere in the grammar. For example, the rule for compound statements is defined right below and specifies that a compound statement is a sequence of statements surrounded by an opening and a closing brace (terminals).

We included at the end of Code 45 the grammar extension that allows platform-dependent snippets to be embedded into generic code. To distinguish them from the rest of the code, we decided to denote them with surrounding sharp symbols, as shown in Code 42.

Even though the development of our semi-generic language is already quite advanced, we consider using or drawing inspiration from a recent work very similar to our approach [Hemel and Visser 2010]. In this work, the authors propose an intermediate platform-independent language to serve as a pivot language between DSLs and GPPLs: instead of having transformations from each DSL to each platform, their idea is to have a single transformation from each DSL to the intermediate language, and transformations from the intermediate language to each platform. The approach adopted is very close to ours: the intermediate language is designed to be the common denominator between most object-oriented languages, using Java as a basis but without the more “exotic” features. In addition to this, the language gives the possibility to declare so-called external classes, akin to abstract data types [Liskov and Zilles 1974]. These classes are in fact interfaces that must be implemented on each platform, usually by wrapping some existing API, thereby allowing easy integration of a great number of libraries.

II.2.3. Additional aspects to consider

In addition to the structural and behavioral features we already described, a standard DEVS metamodel should take into account some other aspects.

II.2.3.1. Parameterization and initialization

The notions of parameterization and initialization are a bit blurry in the DEVS formalism. Obviously, models often need to be parameterized, either to make them reusable in other models or to analyze the impact of some characteristics of the system under study. We think

that it is essential to make a clear distinction between parameters and inputs, as the latter are variable and trigger state changes while the former are just constants fixed prior to simulation.

Another aspect that we think should be formalized is initialization. Usual DEVS does not strictly specify how *initial states* should be determined, resulting in discrepancies in how DEVS environments handle the initialization of atomic models. Such differences can drastically modify the way models are designed from one platform to another. For instance, some implementations may consider that the atomic model is solely responsible for determining its initial state, in isolation, while others may defer initialization of components to coupled models. In the context of our metamodel, meant to be as universal as possible, it is important to define those concepts more strictly.

To illustrate, let us bring back the sample atomic model from Chapter 2.II.1.1, the RandomSwitch. As a reminder, this model was parameterized by the type of values handled and by a probability of success, determining for each input on which output port it should be sent:

$$RandomSwitch_{ValueType,successProbability} = \langle X, Y, S, ta, \delta_{int}, \delta_{ext}, \lambda \rangle$$

with the following state set (the other elements are not relevant here):

$$\begin{aligned} S = \text{currentValue} \in ValueType \times \\ \text{state} \in \{waiting, outputting\} \times \\ \text{isSuccess} \in \{true, false\} \end{aligned}$$

Such a specification is not sufficient to simulate a random switch. Indeed, before being simulated or used in a coupled model, this model must be instantiated, meaning that its parameters must be fixed and that its initial state must be determined. An analogy can be made with the object paradigm—which, as a side note, shares many concepts with DEVS—: like a class needs to be instantiated into individual objects, a model needs to be instantiated into individual components.

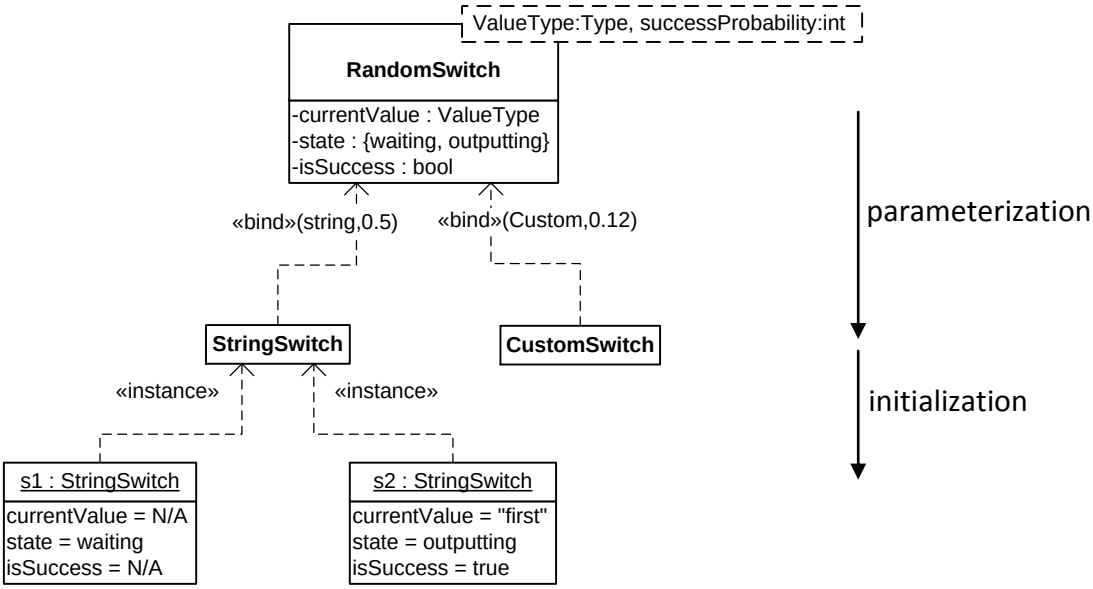


Figure 30. Sample instantiations of the RandomSwitch model.

Using UML notations, Figure 30 shows sample instantiations of RandomSwitch, obtained in two steps: parameterization, where parameters are bound to actual arguments, and initialization, where the initial state is set.

This two-step instantiation can be simplified by considering model parameters as parameters for the initialization. If the value of some parameters must be kept for further use, for instance in transition functions, it can be stored in a corresponding state variable. For instance, in our example, we would add a state variable for the success probability, initialized with the model argument.

This approach has very little impact on the formalism. It boils down to adding a set of parameters that can be defined similarly to ports, i.e. as a structured set (cf. Chapter 2.II.1.1) to ease its manipulation, and an initialization function, which provides the initial state depending on the parameters. Formally, an atomic model becomes a tuple

$$M = \langle \boldsymbol{P}, X, Y, S, \boldsymbol{init}, ta, \delta_{int}, \delta_{ext}, \lambda \rangle$$

where

- X, Y, S, ta, δ_{int}, δ_{ext} and λ are as before.
- P is the set of parameters, possibly in the form of a structured set.

- *init* is the initialization function: $init(P) \rightarrow \mathrm{Q}$. This function provides the initial total state of the model, depending on the arguments passed to the model. The choice of *Q* over *S* is motivated by the not-so-uncommon need to initialize the elapsed time with a positive value. This is notably useful when we want to initialize a model to some state previously encountered during simulation, to reproduce some results or to skip a warm-up period.

To allow transmission of parameters down the model hierarchy, we also need to add some elements to the specification of coupled models. Like atomic models, they will define a set of parameters and an initialization function, but the latter will be different. Instead of providing some initial state, the initialization function of coupled models determines what arguments should be given to their components. Formally, we end up with the following definition for coupled models:

$$N = \langle \boldsymbol{P}, X, Y, D, \{M_d | d \in D\}, \boldsymbol{init}, EIC, EOC, IC, Select \rangle$$

where

- *X*, *Y*, *D*, $\{M_d\}$, *EIC*, *EOC*, *IC* and *Select* are as before.
- *P* is the set of parameters.
- *init* is the initialization function: $init(P) \rightarrow \{a_d | d \in D\}$, with $a_d \in P_D$ being the arguments for component *d*.

Regarding simulation, the only operations impacted are (obviously) the handling of initialization messages. In addition to initializing their times of last and next event, simulators tell their atomic models to initialize themselves with the arguments received:

```
when receiving an initialization message at time t with arguments args
  | (component.s, component.e) = component.init(args)
  |
  | tl = t - component.e;
  | tn = tl + component.ta(s)
```

Code 46. Updated algorithm for initialization of simulators.

For their part, coordinators must fetch the arguments for each component, and send them to each child processor:

```
when receiving an initialization message at time t with arguments args
  | {a_d} = component.init(args)

  | for each child in children
  |   | a = a_i | a_i in {a_d} and
  |   |          child.component.id == i
  |   | send initialization message to child with value (t, a)

  | tl = max(child.tl | child in children)

  | tn = min(child.tn | child in children)
```

Code 47. Updated algorithm for initialization of coordinators.

Adding this small extension to our metamodel is pretty straightforward: we just modify the Model class so that it aggregates a set of parameters, which are typed and named elements, and we add initialization functions to both AtomicModel and CoupledModel. In the former, the function is defined using the semi-generic language already presented, to allow the initialization operations to be as complex as needed; in the latter, we could perhaps be more restricting, for instance by only allowing literals and arguments of the coupled model to appear in the set of component arguments. This way, we could define the mapping between coupled model arguments and components arguments structurally, without resorting to dynamic behavior. However, this approach could prove to be too limited, so we prefer using once again our semi-generic language.

II.2.3.2. Representation of trajectories

The representation of trajectories is another aspect that deserves attention. In DEVS, trajectories denote the occurrence of events in a timeline. Input trajectories are fed to model, which generates in response output trajectories. Similarly, the evolution of the state of a model can be represented by a state trajectory, recording all modifications of the state during simulation.

In our opinion, it is important to devise a standard format for representing these trajectories, for several reasons:

- Model comparison. Given some system and some questions to answer about it, different teams can come up with different models to answer these questions. To compare these models with all other things being equal, experiments must be

performed using the same parameters and inputs. Without an agreed-upon format for representing these inputs, correctly carrying out such comparisons is very difficult.

- Using outputs of a simulation as inputs to others. Leveraging knowledge acquired by previous people is an important part of research; when developing a model based on previous works, it should be possible to reuse the results obtained in these works.
- Results analysis. The outcome of simulations must often be further processed to extract useful information, for instance by performing statistical analysis over results. With a standard format for representing results, it becomes possible to develop generic analysis tools that can be used by many research teams.
- Visualization. It is an important part of simulation, especially when it comes to model validation: it is often interesting to complement raw results with some graphical representation, which can be used by experts to roughly assess the validity of a model. Such graphical representation can range from simple trajectory diagrams to full-fledged simulation animation. Once again, the development of generic visualization tools depends on the availability of a standard trajectory format.

For all these reasons, we think that a metamodel of trajectories is essential. However, such metamodel should probably be separated from the metamodel of DEVS models, as it deals with an orthogonal issue.

The easiest way to represent a trajectory is to store a sequence of values associated with timestamps. In the case of a state trajectory, a pair $(t, value)$ denotes a modification of the state, meaning that at time t, the state changed to become *value*. In the case of input and output trajectories, a pair $(t, value)$ indicates that an event occurred at time t, with value *value*. Usually, the state of models is defined as a set of state variables, and the input/output interface is decomposed into ports. Consequently, values contained in a state trajectory should in fact be a collection of named values, each one corresponding to a state variable, and values contained in input/output trajectories should indicate which port received the event.

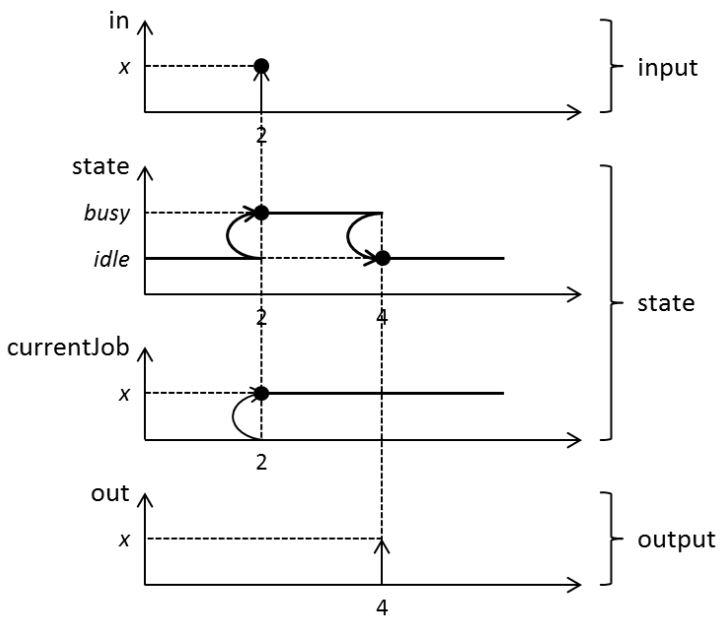


Figure 31. Sample trajectories of a Processor model.

As an example, consider the trajectories in Figure 31, corresponding to a possible simulation of a Processor model. These trajectories show an input event occurring on port *in* at time 2 with value *x*, triggering a state change in the model. Later, at time 4, the model undergoes an internal transition which modifies its state once again and triggers an output event on port *out*, with value *x*.

Code 48 gives a representation of these trajectories using a possible concrete syntax for the trajectory metamodel. This syntax is probably not the best, but we use it here only to show the overall structure of the corresponding metamodel.

```
input:
2: in -> x

state:
2: { state -> busy, currentJob -> x }
4: state -> idle

output:
4: out -> x
```

Code 48. Sample textual format for representing trajectories.

An issue to address when defining a metamodel for trajectories is the representation of values. Indeed, values can be anything, ranging from primitive types to complex data

aggregates. To correctly read these values, a program must have some information about their structure. The problem is similar to the one encountered in XML, and can be tackled in two different ways: simply write/read the values as structured data, without additional information (see the Document Object Model (DOM) and the Simple API for XML (SAX) for examples [Møller and Schwartzbach 2006]), or define the structure of the data in a schema, which can be used to generate the corresponding classes (see XML Schema for instance [Walmsley 2001]). In our case, the schema/metamodel can already be available in the specification of the DEVS model, so it can be used to facilitate the (de)serialization of the data. When dealing with platform-dependent types, the issue is a bit more complex, but can be solved using existing data binding frameworks.

Finally, it would be interesting to allow more complex visualizations [Venhola and Wainer 2006] by relying on wrapper models, akin to those used in GMF, to map data of the trajectory with graphical elements.

II.3. Model-to-model transformations

As we said in Chapter 3, a major feature of MDE is the possibility to easily define model transformations. In this section, we will present some model-to-model transformations we developed for our DEVS metamodel, as well as some ideas for future transformations.

II.3.1. DEVS2SVG: Coupled model diagram generation

To grasp the structure of a coupled model, it is often easier to look at a graphical representation rather than a plain textual specification. Usually, coupled models are depicted in diagrams showing the relations between components (as in Figure 2, for instance). In these diagrams, components are denoted by boxes, with some shapes on their perimeter to represent ports. The couplings between components are symbolized by lines connecting source ports to destination ports.

With a model specification written in a well-defined format, such as our metamodel, it is possible to automatically generate the corresponding diagram. To do so, we need a transformation from our metamodel to some image file format. Ideally, we would like to avoid M2T transformations, as they are more complex to write and less “model-driven”. Consequently, we need a destination format with a well-defined metamodel.

We chose to use Scalable Vector Graphics (SVG), the graphics format developed by the W3C [SVG]. Indeed, in addition to being an open standard widely deployed and supporting high-quality images, SVG is an XML language defined by a RELAX NG schema. Thanks to this, we can obtain an SVG metamodel compatible with EMF by using the provided XML Schema importer[33]. Conveniently, the serialization of SVG models conforming to the generated SVG metamodel will produce valid SVG files, thereby avoiding the need for an additional M2T transformation.

With metamodels for both DEVS and SVG, it becomes possible to create an M2M transformation to map one to the other. To do so, we used the Operational Query/View Transformation, provided by EMP. The idea behind QVTO transformations is to take elements from the source model and map them to elements in the target model. Of course, the mappings can be more complex than simple one-to-one association. To give a glimpse of what QVTO transformations look like, we will explain hereafter a fragment of our DEVS2SVG transformation.

```
modeltype devs "strict" uses 'http://www.isima.fr/limos/devs/1.0';
modeltype svg "strict" uses 'http://www.w3.org/2000/svg';

transformation devs2svg(in inModel : devs, out outModel : svg);
```

A QVTO file starts with the definition of the metamodels used in the transformation. In our case, two metamodels are involved, `devs` and `svg`. Then, a signature declares the transformation defined in the file. The transformation has a name and operates on some parameters, which are instances of the metamodels specified previously. Often, a transformation simply transforms one input model to one output model, but it is possible to have several inputs, several outputs and even “inout” parameters, to allow in-place transformations.

[33] Since RELAX NG is not supported by EMF yet, we had to use a conversion of the official schema to XML Schema. Instead of doing the conversion ourselves, we used a version found on the SVG mailing list.

```
main() {
  -- select all coupled models and invoke the "toSvg" mapping on them
  inModel.rootObjects()[CoupledModel]->map toSVG();
}
```

As in many languages, `main` denotes the entry point of the transformation. Here, we simply fetch all the coupled model elements from the input model, and map them to SVG elements by invoking the following mapping:

```
mapping CoupledModel::toSVG() : SvgType {

  -- define some helpers variables
  var coupledWidth : Integer :=
    self.components->size() * totalComponentWidth;
  var coupledHeight : Integer :=
    self.components->size() * totalComponentHeight;

  -- set the size attributes of the target svg object
  width := (coupledWidth + portSize).toString();
  height := coupledHeight.toString();

  g += object GType {          -- create a group and add it to the svg
    rect += object RectType { -- create a rectangle and add it to the group
      x := "0";
      y := "0";
      width := coupledWidth.toString();
      height := coupledHeight.toString();
      style := "fill:none;stroke:black;stroke-width:5";
    };

    -- map each component to its representation and add it to the svg
    g += self.components->map toSubRectangle();

    -- map each port to its representation and add it to the svg
    g += self.ports->map toTriangle(0, 0,                        -- position
                                    coupledWidth, coupledHeight,-- size
                                    "__this__");                 -- name

    -- map each coupling to a connecting line and add it to the svg
    line += self.couplings->map toLine();
  }
}
```

As its signature states, this mapping maps a *CoupledModel* element to an *SvgType* element. The first lines simply define some local variables containing the width and height of the coupled model in the resulting diagram. The actual mapping begins right after, starting with the assignments of `width` and `height`, which are two features of the target element. Then, an object *GType* is created and added to the `g` collection of the target element (in SVG, `<g>`

elements are used to group shapes together). In this group, a rectangle is created, which corresponds to the box of the coupled model itself. One can see how simple it is to create elements in the target model, and to set some of their attributes with values derived from the source model. Finally, instead of iterating explicitly over each element of the coupled model to process it, we invoke mappings on the various collections. In the end, each component will be mapped to a rectangle, each port to a triangle and each coupling to a line. A sample SVG automatically generated by this transformation is provided in Chapter 5.III.2.

II.3.2. DEVS2XHTML: Documentation generation

Diagrams are useful to better understand models, but are not sufficient. As every complex artifact, models need to be thoroughly documented. It is especially true in modeling, where explaining hypotheses and modeling choices is essential. Of course, it is impossible to automatically generate semantic comments about a model, but what we can do is automate the creation of documentation from a model annotated by the practitioner, akin to what is done by Javadoc or Doxygen.

To do so, we started by adding an optional *documentation* feature to all elements of our DEVS metamodel. This gives a place to the practitioner where he can document the purpose of models, ports, state variables and so on. Then, we determined a proper output format for the generated documentation. The first one that comes to mind is the HyperText Markup Language (HTML), which is universally supported and particularly well-suited to display and format cross-referenced data. However, the lax syntax of HMTL makes it not very-well suited to MDE applications. A better candidate is its successor, the eXtensible HyperText Markup Language (XHTML), which is nothing more than a reformulation of HTML as an XML language. XHTML is defined in an XML Schema, enforcing rules that were previously less strict. A practical advantage is that XHTML can be readily imported into EMF for integration in our model-driven environment.

The DEVS2XHTML transformation we defined is quite similar to the DEVS2SVG transformation we presented previously. Each model is processed to generate the corresponding XHTML document, where each model element is mapped to an XHTML element presenting its various features as well as its documentation. Pages are cross-

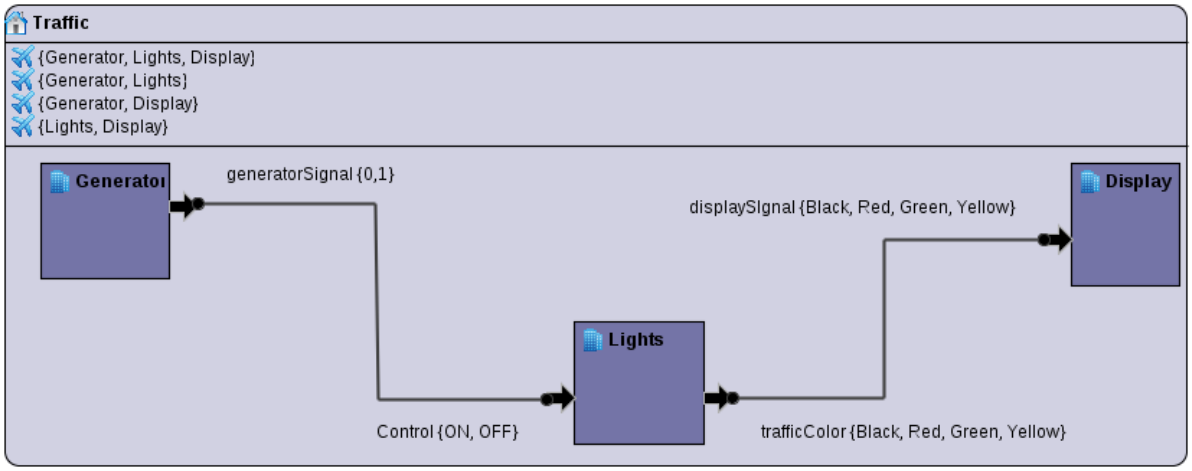


Figure 32. Sample coupled model in Eclipse-DDML. [Ughoroje 2010]

referenced to facilitate navigation, for instance between a component and its model. Additionally, in the case of coupled models, the SVG diagram generated previously is included in the documentation. Chapter 5.III.2 provides an example of generated documentation.

II.3.3. DDML2DEVS: Graphical edition of DEVS models

In Chapter 3.IV.2.1, we presented GMP, the EMP project focusing on automatic generation of graphical editors for domain-specific modeling languages. One of our goals is to leverage this framework to provide a concrete graphical syntax for our metamodel.

A member of our team, Ufuoma B. Ighoroje, developed an editor for the DEVS-Driven Modeling Language (DDML) [Traoré 2009], a graphical notation for representing both coupled and atomic DEVS models. This editor, named Eclipse-DDML, is defined as an Eclipse plug-in based on GMF [Ighoroje and Traoré 2011]. Figure 32 and Figure 33 show respectively a sample coupled model and a sample atomic model, specified with the diagramming features provided by Eclipse-DDML.

Eclipse-DDML has its own metamodel, based on DDML. To integrate the edition capabilities of Eclipse-DDML and the transformations presented in this thesis, we will need to build a bridge between the two metamodels. As they are both specified in Ecore, we will be able to use an M2M transformation to map DDML elements to elements of our DEVS metamodel. Normally, the transformation should be pretty simple since the discrepancies are small. The

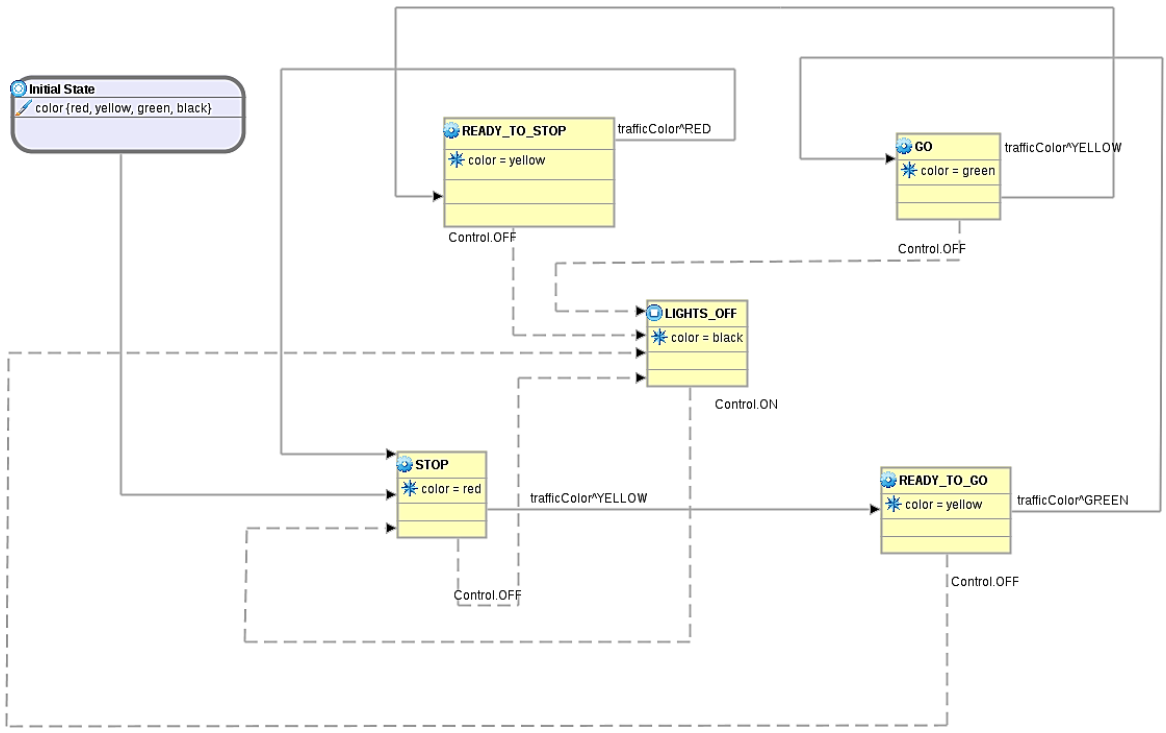


Figure 33. Sample atomic model in Eclipse-DDML. [Ughoroje 2010]

aspect that could be a bit challenging is the transformation from the DDML state diagram notation to semi-generic code.

II.3.4. Inter-formalism transformations

Another kind of M2M transformations we would like to experiment with are formalism transformations. In Chapter 2.II.1.2, we stated that DEVS is a good candidate for being a "universal" intermediate language for simulating models written with heterogeneous formalisms. The Formalism Transformation Graph [Vangheluwe 2000] in Figure 1 showed several possible transformations between various formalisms, eventually leading to DEVS as a common denominator.

Having a metamodel for DEVS gives us a great opportunity to experiment with these transformations. We can develop or reuse metamodels for the various formalisms, and try to develop semantic-preserving M2M transformations from these formalisms to DEVS. Eventually, such transformations could be used in SimStudio to allow multi-paradigm modeling [Vangheluwe 2002], meaning modeling a single system through multiple models using different formalisms.

Our first experiment will be done on Petri Nets, which is a simple formalism for which metamodels are readily available[34] [Wachsmuth 2007].

II.4. Code generation

One of the motivations for our DEVS metamodel was to make models usable in any DEVS environment. To achieve this, we devised a platform-independent representation for DEVS models, but that is just one part of the story. The next step is to provide conversions from our metamodel to the existing implementations so that a practitioner can benefit from their features. Most notably, generating platform-specific versions of a model are essential to be able to simulate it[35].

Since the definition of our semi-generic language is still in process, the generation of fully executable specifications is not possible yet. However, we already implemented preliminary features that generate the entire structure of models, leaving only the functions of atomic models to be defined. At the time of this writing, the environments supported are CD++, PyDEVS and DEVS-MS, the simulator presented in Chapter 6.

To perform the generation, we rely on M2T transformations written in Xpand. Like most text generation languages, Xpand is based on templates: the desired output is written as is, but can embed expressions that are processed during execution to customize the resulting file depending on data provided as input. In addition to basic expressions, Xpand provides several statements to create files, iterate over collections, generate errors, etc. Xpand discriminates statements and expressions from static text by enclosing them in French quotation marks (« and »).

To explain the major concepts of Xpand, here are some fragments of our DEVS2CD++ template file:

[34] http://www.emn.fr/z-info/atlanmod/index.php/ZooFederation#PetriNet_5.0 (last accessed 21/06/2012)
[35] Another solution would be to implement a simulator capable of directly interpreting models specified with our metamodel, but given the numerous simulators already available, this would probably be a waste of time.

```
«IMPORT DEVS»

«EXTENSION transformations::model»
```

An Xpand file starts with any number of “import” statements, possibly followed by “extension” statements. Imports bring into scope the names of a namespace, to avoid using fully-qualified names throughout the file. Extensions are a way of adding operations to model elements to facilitate their manipulation. They are defined in their own file, using another language named Xtend. In our case, we extended some elements of our metamodel to ease the retrieval of ports, couplings and components of a certain kind (e.g. retrieve only the input ports of a model).

```
«DEFINE Main FOR AtomicModel»
«EXPAND atomicHeader»
«EXPAND atomicImplementation»
«ENDDEFINE»
```

The rest of the file contains “define” statements. Each “define” block defines a template for some element of the source metamodel, which can be expanded (invoked) by other templates. To initiate the generation, a main template must be defined.

Xpand supports polymorphism: several templates can have the same name as long as they are defined for different model elements. The version invoked when expanding the template depends on the actual type of the argument provided. For instance, in the DEVS2CD++ transformation, we defined two main templates, one for atomic models and one for coupled models.

Here, the main template delegates all the work by expanding two other templates, one to generate the header file of the model, i.e. its declaration, and one to generate its implementation file, i.e. its definition.

```
«DEFINE atomicHeader FOR AtomicModel»
«FILE name + ".h"-»
#ifndef «name.toUpperCase()»_H
#define «name.toUpperCase()»_H

[...]

«LET name.toFirstUpper() AS class-»
class «class» : public Atomic
{
    [...]

    «FOREACH inputPorts() AS port-»
    /**
     * «port.documentation»
     */
    const Port & «port.name»;
    «ENDFOREACH-»

    [...]
}; // class «class»

«ENDLET-»
«ENDFILE-»
«ENDDEFINE»
```

Now here is a template with actual content. This definition starts by instructing the processor to create a file, named as the input atomic model. The “file” block defines what will be written to this file. The static text, in blue, will be kept as is while the statements/expressions enclosed in quotation marks will be evaluated and replaced with content depending on the input model. For instance, the name of the class will be constructed from the name of the atomic model, where the first letter will be set in upper-case.

This template also exemplifies the use of looping constructs. To generate the code for each port, we iterate over a collection of elements obtained from the input model, using a “foreach” statement. The sample case presented in the next section shows examples of resulting code.

As we saw, defining M2T transformations with Xpand is quite simple. To target new DEVS environments, one just needs to write the appropriate templates. Though we currently only provide transformations to CD++, PyDEVS and DEVS-MS, we plan to add support for DEVSJava, James II and VLE in a close future.

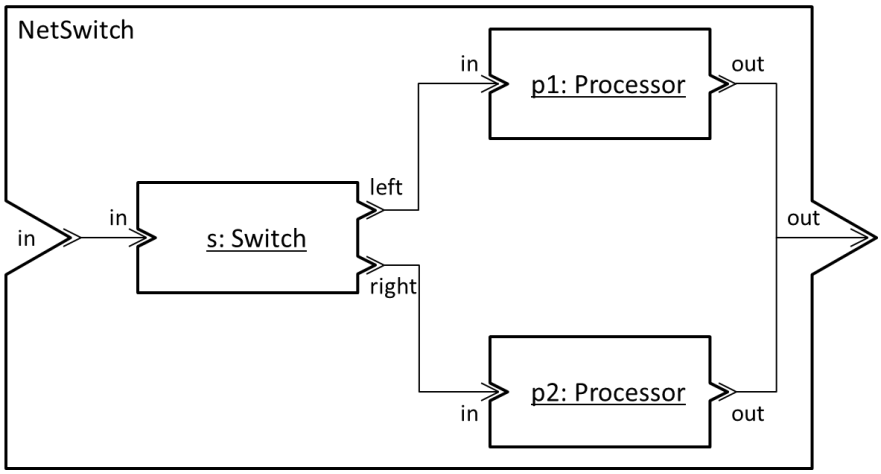


Figure 34. NetSwitch coupled model.

To illustrate the various features just described, the following section provides a sample application of our model-driven approach.

III. Application to a switch network model

To keep things simple, we will use a simple and well-known model to illustrate our work, namely the switch network proposed in [Zeigler et al. 2000]. The NetSwitch is a coupled model with three atomic components: one switch and two processors. A switch receives inputs on its port and immediately sends them to one of its output ports, alternating between the two. A processor receives jobs, processes them during a given amount of time, and sends the processed job on its output port. The NetSwitch model combines these models to process jobs alternatively with one processor or the other. Its structure is depicted in Figure 34.

III.1. Model creation/edition

EMF provides several ways to create instance models from an Ecore metamodel. The first one is to do it programmatically; indeed, EMF generates Java classes corresponding to each element of the metamodel. These classes are used internally by EMF, but can be used directly by the toolsmith if need be. However, in the context of DSL development, it is better to keep the practitioner unaware of this kind of low-level details. Even the toolsmith is better off leaving them aside if he wants his metamodel to be easily portable to other MDE environments. Indeed, as long as we manipulate only high-level model-driven languages, it is

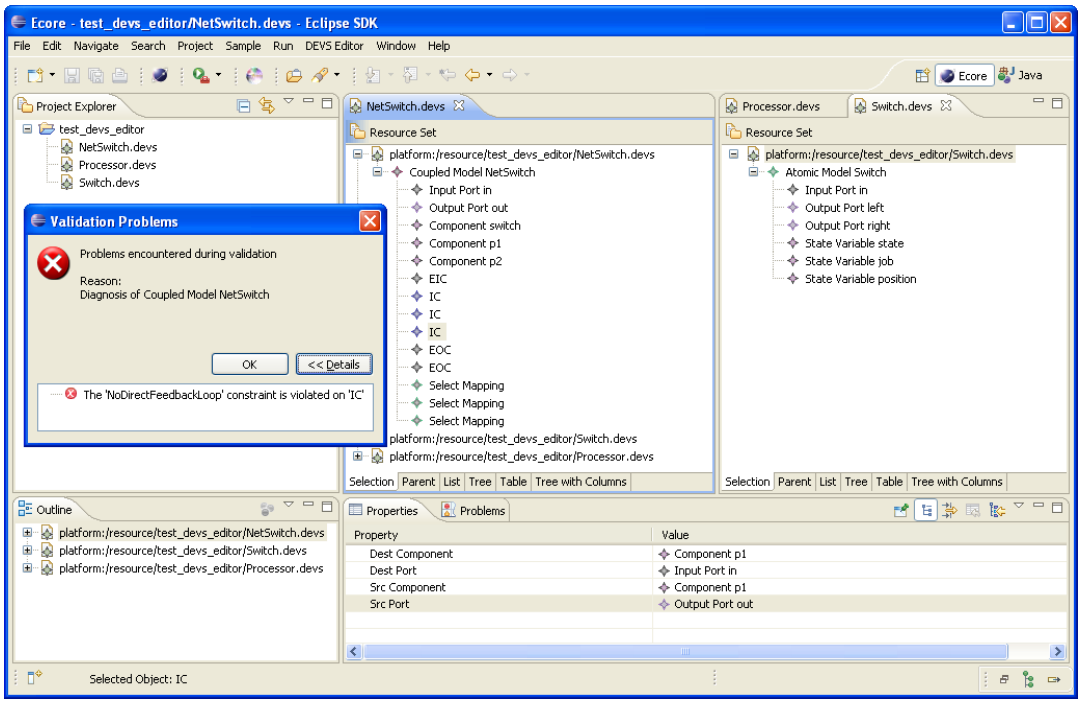


Figure 35. DEVS model editor generated by EMF.

always possible to use automatic transformations to transition to other tools, for example by translating Ecore models to KM3[36], or ATL transformations to QVTO [Jouault and Kurtev 2006]. Consequently, we try to ignore as much as possible the code generated by EMF.

Since we do not want to manipulate Java code to create our DEVS model, a better solution is to leverage the reflective framework provided by EMF. Indeed, EMF is capable of manipulating metamodels without generating the corresponding code beforehand. This reflexivity allows us to create *dynamic instances*, meaning models that are created on-the-fly and can be edited using the provided reflective editor. This editor presents a model as a tree of elements, whose features can be set individually.

Even though creating dynamic instances is valuable to test metamodels and transformations, it is mainly a convenience feature for the toolsmith. To make the environment available to practitioners, one must generate an Eclipse plug-in containing the editors, transformations and text generators surrounding the metamodel. EMF completely automates the generation of such plug-in, including a full-fledged editor for models. As an example, Figure 35 shows the editor generated by EMF from our DEVS metamodel.

[36] Transformation available at http://dev.eclipse.org/svnroot/modeling/org.eclipse.gmt.am3/dsls/trunk/KM3-2005/Compiler/Ecore2KM3.atl (last accessed 20/06/2012)

This editor allows us to create coupled or atomic models, to add elements to them, and to specify their properties. The screenshot shows the definition of the NetSwitch and Switch models, as well as the properties of one of the internal coupling. An interesting feature provided by the generated editor is model validation. Remember that we annotated our metamodel with several constraints describing the invariants that should be respected by DEVS model. In this example, we voluntarily introduced an incorrect coupling, which connects a component with itself. When validating the model, the editor warns us that we violated the “No direct feedback loop” constraint, making the model incorrect. This kind of early verification of models is very valuable, as it provides immediate feedback to the practitioner instead of failing later, during simulation (or worse, going unnoticed for a long time). Thereby, the modeler can quickly correct the minor implementation mistakes and focus on more important topics.

Finally, the last way available out of the box to edit models is to directly edit their XMI file. However, XMI is a serialization format, not really meant for human manipulation. As a consequence, manually editing the XMI representation of a model is rather error-prone, and should be kept for exceptional cases. Nevertheless, XMI is an XML language that is readable by a human, despite its relative verbosity. As an example, the beginning of the NetSwitch XMI file is given in Code 49 (with a bit of custom formatting).

```
<?xml version="1.0" encoding="us-ascii"?>
<devs:CoupledModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:devs="http://www.isima.fr/limos/devs/1.0"
    xsi:schemaLocation="http://www.isima.fr/limos/devs/1.0 DEVS.ecore"
    name="NetSwitch"
    documentation=
        "Simple switch network.
         Jobs received are sent alternatively to p1 and p2 for processing.">
  <ports xsi:type="devs:InputPort" documentation="Receives jobs to process."
         name="in" type="string"/>
  <ports xsi:type="devs:OutputPort" documentation="Sends processed jobs."
         name="out" type="string"/>
  <components documentation="Switch sending jobs alternatively to p1 and p2."
              name="s">
    <type xsi:type="devs:AtomicModel" href="Switch.xmi#/"/>
  </components>
```

Code 49. Fragment of the XMI file of the NetSwitch model.

NetSwitch coupled model documentation

Simple switch network. Jobs received are sent alternatively to p1 and p2 for processing.

Components

Name	Type	Description
p1	Processor	First processor.
p2	Processor	Second processor.
s	Switch	Switch sending jobs alternatively to p1 and p2.

Input ports

Name	Type	Description
in	string	Receives jobs to process.

Output ports

Name	Type	Description
out	string	Sends processed jobs.

Couplings

p1.out	=>	self.out
s.outLeft	=>	p1.in
self.in	=>	s.in
s.outRight	=>	p2.in
p2.out	=>	self.out

Figure 36. XHTML documentation generated from the NetSwitch model.

III.2. Documentation generation

Now that our NetSwitch model is defined, in conformity with our metamodel, we can apply transformations on it, for instance to generate documentation. Figure 36 shows the XHTML page automatically generated from the model specification, styled with a Doxygen-like Cascading Style Sheet (CSS).

This page summarizes the main characteristics of the coupled model, including the annotations embedded in the model to provide some insight into the purpose of the various elements. The pages of the sub-models are linked to facilitate navigation across the documentation. Finally, the documentation also includes the SVG diagram of the coupled

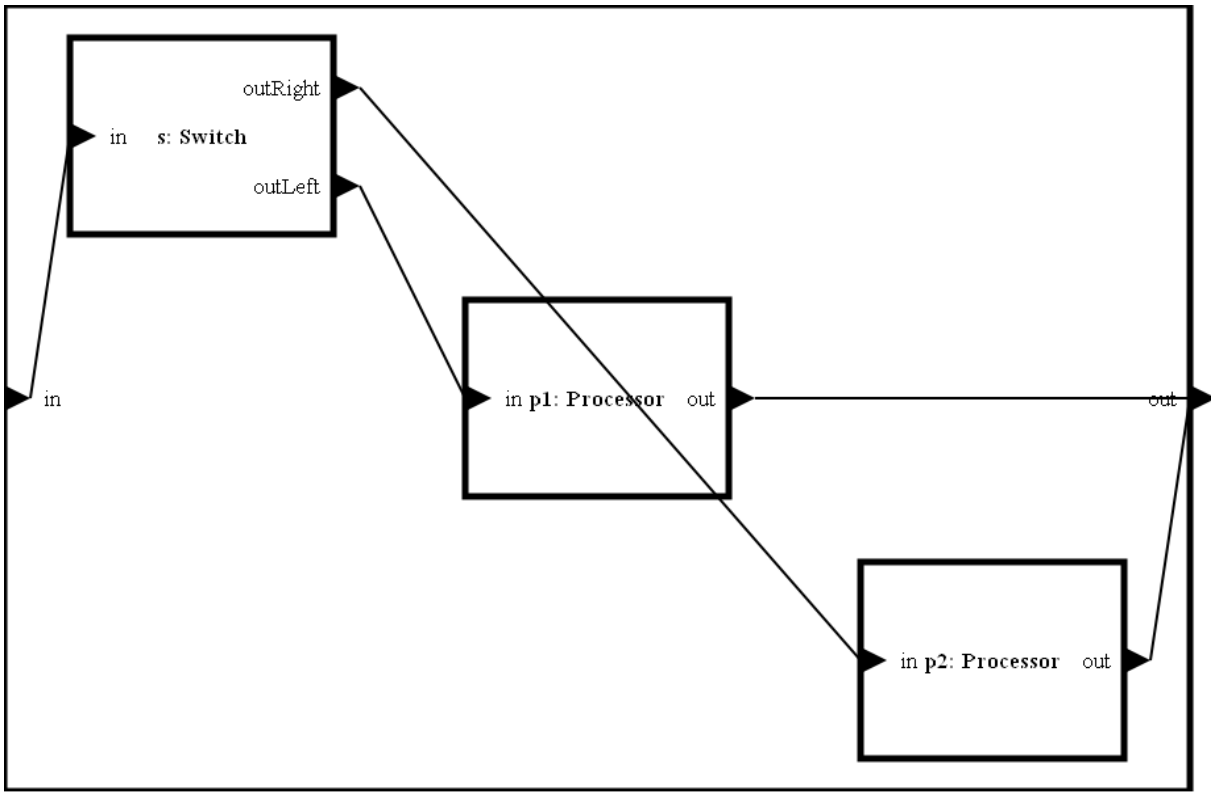


Figure 37. SVG diagram generated from the NetSwitch model.

model, shown in Figure 37, which is automatically generated by another M2M transformation. The layout of the diagram could obviously be improved, either by implementing some layout algorithms in our transformation or by targeting a format more suited to graph representation than SVG. A good candidate would be DOT, the Graphviz[37] description language, whose metamodel is readily available[38].

III.1. Code generation

To actually simulate the switch network we just specified, we need a representation of the model that is understandable by a simulator. Depending on the framework we wish to use, the representation will greatly vary: usually, models are specified as classes in the programming language of the framework, and must fulfill certain requirements such as deriving from a particular base class or overriding specific methods. In other cases, the

[37] http://www.graphviz.org/ (last accessed 20/06/2012)
[38] http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/visualization.html (last accessed 20/06/2012)

model description will be written with a custom language, as in CD++ where coupled models are defined in .ma files.

Thankfully, we do not need to care about all these implementation details. Thanks to the platform-independency of our metamodel and to the M2T transformations we developed, we can automatically generate various versions of the NetSwitch model specific to each implementation supported, namely CD++, PyDEVS and DEVS-MS for the time being. This generation is fully generic, i.e. any DEVS specification can be transformed into code without any intervention from the practitioner.

Giving in this document the whole code generated from the NetSwitch specification would be unreasonable. Therefore, we will only provide small fragments of the Switch atomic model and NetSwitch coupled model in each implementation, simply to show the wide variations there can be between DEVS frameworks.

III.1.1. Generated CD++ code

In CD++, atomic models are defined as C++ classes. A C++ good practice is to separate declaration from definition; we follow this guideline when generating code by splitting the model specification into a header file containing the class definition along with its method prototypes, and an implementation file containing the methods definition. Code 50 partially shows the definition of the Switch class, and Code 51 the definition of the class constructor, which correctly initializes the various data members.

A particularity of CD++ is that it uses a custom language to describe coupled models. Specifications written in this language are stored in .ma file, which contains a top model composed of components, input and output ports, and links (couplings) between components. The list of components is made of atomic components, linking to their model class, and coupled components, which are defined below in the file. Code 52 shows the generated .ma file for the NetSwitch model, which is alone in the file since it only aggregates atomic components.

```
/**
 * Simple switch.
 */
class Switch : public Atomic
{
public:
    Switch( const std::string & name = "Switch" );
    virtual std::string className() const;

protected:
    Model & initFunction();
    Model & externalFunction( const ExternalMessage & );
    [...]

private:
    /**
     * Input.
     */
    const Port & in;

    /**
     * Left output.
     */
    Port & outLeft;
    [...]
    /**
     * Current value.
     */
    string value;
}; // class Switch
```

Code 50. CD++ – Switch class definition.

```
Switch::Switch( const std::string & name )
  : Atomic( name ),
    in( addInputPort( "in" ) ),
    outLeft( addOutputPort( "outLeft" ) ),
    outRight( addOutputPort( "outRight" ) )
{
    // Param recup
}
```

Code 51. CD++ – Switch constructor definition.

```
[top] % Simple switch network. Jobs received are sent alternatively to p1
and p2 for processing.
components : p1@Processor p2@Processor s@Switch
in : in
out :  out
link : in in@s

link : outLeft@s in@p1
link : outRight@s in@p2

link : out@p1 out
link : out@p2 out
```

Code 52. CD++ – NetSwitch specification.

III.1.2. Generated PyDEVS code

In PyDEVS, atomic models are specified as classes inheriting from an AtomicModel base class. This abstract base class defines a `state` data attribute for storing the state of the model. Consequently, when a model has a state composed of several variables, it is often associated with a custom class representing this state. As shown in Code 53, we generate such classes with several convenient features such as a parameterized constructor to initialize the state, accessors to the different variables, and a default implementation for the special method `__str__` used to obtain a string representation of an object.

```
class SwitchState:
  """Switch's state
  """

  def __init__(self , outputLeft, value):
    """Constructor (parameterizable)
    """
    self.setOutputLeft(outputLeft)
    self.setValue(value)

  def setOutputLeft(self, value):
    self.__outputLeft = value

  def getOutputLeft(self):
    """ True if the next output will be on the left output port.
    """
    return self.__outputLeft
```

```
  def setValue(self, value):
    self.__value = value

  def getValue(self):
    """ Current value.
    """
    return self.__value

  def __str__(self):
    return str(self.__outputLeft) + str(self.__value)
```

Code 53. PyDEVS – State class for the Switch model.

This state class is then used in the actual atomic model. Code 54 shows the constructor generated for the class, which initializes the total state of the model and creates the various ports.

```
class Switch(AtomicDEVS):
  """ Simple switch.
  """

  def __init__(self, name=None):
    """ Constructor (parameterizable)
    """

    AtomicDevs.__init__(self, name)

    self.state = SwitchState()

    self.elapsed = 0.0

    self.IN = self.addInPort(name="IN") # Input.

    self.OUTLEFT = self.addOutPort(name="OUTLEFT") # Left output.
    self.OUTRIGHT = self.addOutPort(name="OUTRIGHT") # Right output.
```

Code 54. PyDEVS – Switch constructor.

The definition of coupled models is rather similar to that of atomic models. It boils down to writing a class deriving from the proper base class and initializing the needed elements in the constructor, i.e. creating ports, components and couplings. The full PyDEVS definition of the NetSwitch model is provided in Code 55.

```
class NetSwitch(CoupledDEVS):
  """ Simple switch network. Jobs received are sent alternatively to p1 and
p2 for processing.
  """

  def __init__(self, name=None):

    CoupledDEVS.__init__(self, name)

    self.IN = self.addInPort(name="IN") # Receives jobs to process.

    self.OUT = self.addOutPort(name="OUT") # Sends processed jobs.

    self.P1 = self.addSubModel(Processor(name="P1")) # First processor.
    self.P2 = self.addSubModel(Processor(name="P2")) # Second processor.
    self.S = self.addSubModel(Switch(name="S")) # Switch sending jobs
alternatively to p1 and p2.

    self.connectPorts(self.IN, self.S.IN)

    self.connectPorts(self.S.OUTLEFT, self.P1.IN)
    self.connectPorts(self.S.OUTRIGHT, self.P2.IN)

    self.connectPorts(self.P1.OUT, self.OUT)
    self.connectPorts(self.P2.OUT, self.OUT)

  def select(self, immList):

    immSet = frozenset(immList)

    if frozenset([self.S]) <= immSet:
      return self.S

    if frozenset([self.P1, self.P2]) <= immSet:
      return self.P1
```

Code 55. PyDEVS – NetSwitch definition.

III.1.3. Generated DEVS-MS code

The DEVS-MetaSimulator being the subject of the next chapter, we will not go into much detail here. We simply provide some of the generated DEVS-MS code to be thorough, and to highlight the fact that two DEVS environments sharing the same programming language can nevertheless have vastly different ways of representing the same things. Indeed, DEVS-MS is written in C++ like CD++, but comparing Code 56 and Code 57 with their CD++ counterparts show that they have little in common.

```
DECLARE_PORT((Switch), in)
DECLARE_PORT((Switch), outLeft)
DECLARE_PORT((Switch), outRight)

/**
 * Simple switch.
 */
class Switch
    : public devs::classic::AtomicModel<
        PORTS(
            ((Switch::in, string)) // Input.
        ),
        PORTS(
            ((Switch::outLeft, string)) // Left output.
            ((Switch::outRight, string)) // Right output.
        )
    >
```

Code 56. DEVS-MS – Beginning of the Switch class definition.

```
DECLARE_PORT((NetSwitch), in)
DECLARE_PORT((NetSwitch), out)

DECLARE_COMPONENT((NetSwitch), p1)
DECLARE_COMPONENT((NetSwitch), p2)
DECLARE_COMPONENT((NetSwitch), s)

namespace NetSwitch {

    typedef devs::classic::CoupledModel<
        COMPONENTS(
          ((p1, Processor)) // First processor.
          ((p2, Processor)) // Second processor.
          ((s, Switch)) // Switch sending jobs alternatively to p1 and p2.
        ),
        COUPLINGS(
          // External Input Couplings
          (( (devs::This, in), (s, in) ))

          // Internal Couplings
          (( (s, outLeft), (p1, in) ))
          (( (s, outRight), (p2, in) ))

          // External Output Couplings
          (( (p1, out), (devs::This, out) ))
          (( (p2, out), (devs::This, out) ))

        ),
        SELECT(
          ( (s)  )( (p1) (p2) )
        ),
        PORTS(
          ((NetSwitch::in, string)) // Receives jobs to process.
        ),
        PORTS(
          ((NetSwitch::out, string)) // Sends processed jobs.
        )
    > base;
} // namespace NetSwitch
```

```
/**
 * Simple switch network. Jobs received are sent alternatively to p1 and p2
for processing.
 */
class NetSwitch
    : public NetSwitch::base
{

  public:

    NetSwitch()
        : NetSwitch::base(p1, p2, s),
          p1(),
          p2(),
          s()
    {}

  private:

    Processor p1; /// First processor.
    Processor p2; /// Second processor.
    Switch s; /// Switch sending jobs alternatively to p1 and p2.
};
```

Code 57. DEVS-MS – NetSwitch definition.

The wide variety of code we are able to generate shows how the use of a platform-independent metamodel along with model transformations greatly facilitates model reuse and sharing: from a single model specification, we were able to automatically generate implementations for three very different DEVS environments, and several others could be supported quite easily.

IV. Conclusion

In this chapter, we presented our model-driven approach to DEVS M&S, implemented using the Eclipse Modeling Project. This approach is based on a platform-independent metamodel for DEVS, which tries to be as generic as possible while still being practical to use. Achieving full genericity is particularly hard when it comes to specifying the behavior of atomic models. To solve this issue, we devised a semi-generic language that provides a set of generic constructs corresponding to the common denominator between most programming languages, but also allows extending its capabilities through abstract code snippets for which platform-specific mappings need to be explicitly specified.

From this metamodel, EMF provides many features out of the box, such as generation of the corresponding Java code, generation of graphical editors and (de)serialization of models

to/from a standard format. However, the full power of MDE is obtained by using model transformations. Indeed, MDE environments in general and EMP in particular include high-level languages for manipulating models. These languages allow a toolsmith to easily develop transformations over a metamodel, which could then be applied to any model conforming to this metamodel.

In our case, we used these languages to provide automatic generation of several artifacts. Given a model specification, SimStudio can generate documentation for this model, as a set of linked XHTML pages. In the case of coupled models, this documentation also includes SVG images depicting the model structure in a graphical way. More importantly, we also developed several code generators that automatically produce model specifications targeted to other DEVS environments. As of now, SimStudio can generate code for CD++, PyDEVS and DEVS-MS. This automatic generation of implementations allows the practitioner to ignore low-level details, reducing the risk of introducing programming errors when implementing a model on a specific platform. In addition, it makes transitioning between environments very easy, since it boils down to selecting a different target for the generation. Thereby, models can be more easily exchanged between practitioners, even if they use different DEVS tools.

Another important feature that we implemented in SimStudio is model verification. Using the validation framework provided by EMP, we included several constraints in our DEVS metamodel. These constraints define invariants that must be met by all DEVS models, and are automatically verified by the environment during the development of models. This way, several design errors are caught at an early stage, presented to the practitioner in form of comprehensive error messages allowing him to quickly identify the problems and make the necessary corrections.

To fully grasp the advantages of applying MDE to DEVS M&S, it is interesting to go back to the use cases for DEVS models described at the beginning of this chapter (Figure 20, page 159) and see how they are impacted by our MDE approach:

- Reuse: as in any DEVS framework, models defined in SimStudio are reusable components that can be included in larger models. Basically, reusing a model boils down to referecing its specification through the appropriate Uniform Resource Identifier (URI). Though, an interesting aspect is that the platform-

independence of the specification makes it possible to reuse the model in models defined with other tools, after converting it through the appropriate model transformation.

- Store/retrieve: EMF models are serialized to XMI, a standard and compressed format. Therefore, they can be saved as plain files or in document-oriented databases, and reloaded by any XMI-conformant tool. Moreover, their well-defined structure allows their properties to be queried, an essential feature for seting up searchable model repositories.
- Simulate (interoperability): regarding the simulation aspect, the MDE approach is less appropriate than using plain code, mainly because it focuses more on static data than dynamic behavior. This is why we relied on model transformations to delegate the simulation to external tools, better suited to the job. However, having a representation of the model at a higher level of abstraction still provides advantages: for instance, the knowledge about the model can be used to generate highly optimized code. More importantly, it allows models to be seamlessly ported to different platforms, providing the kind of interoperability described in Chapter 2.III.1.3.
- Visualize/edit: thanks to the use of a metametamodel, MDE frameworks provide tools that greatly facilitate the development of graphical and textual editors associated with metamodels. These meta-tools can be a great help to the DEVS toolsmith developping a DEVS environment.
- Analyze: by conforming to an explicit metamodel, DEVS models defined with an MDE approach can easily be processed by automated analysis tools. All properties of models are readily available, making it possible to implement various types of statis analysis such as verification or measures of complexity. However, dynamic analysis cannot be performed directly on the model and must be implemented in simulators.
- Share: as we said previously, SimStudio models are serialized in a standard format devised specifically for exchanging models between heterogeneous tools. Thereby, they can be exploited with any XMI-compliant tool. Moreover, thanks to the use of a platform-independent metamodel, our MDE approach makes DEVS

models compatible with a variety of simulators, allowing them to be shared by practitioners even if they do not use the same DEVS environment.

We already implemented several features relating to these various use cases, but there is still work to be done. The priority is to finalize the implementation of our semi-generic language. Indeed, even though we already defined the language metamodel and grammar, we still need to develop the subsidiary transformations, and notably handle the abstract snippets through the use of a mapping metamodel.

The next goal is to integrate the various elements presented in this chapter in a full-fledged extensible M&S environment, SimStudio. A first step to achieve this will be the development of an M2M transformation to bridge between our metamodel and Eclipse-DDML, a graphical editor for the DEVS-Driven Modeling Language developed in our team.

Finally, we want to experiment with some more theoretical aspects, such as formalism transformation or model analysis. Our DEVS metamodel provides a sound basis for such experiments by being very amenable to processing by software.

We showed in this chapter how MDE changes the way we think about software development. The MDE environments currently available provide many features that greatly facilitate the development of complex software systems, by raising the abstraction level like structured programming and object-oriented design did. This raise of abstraction allows new features to be developed very quickly, making SimStudio an environment that can be easily extended.

In the next chapter, we will apply another software development paradigm to DEVS M&S, namely metaprogramming. Even though metamodeling and metaprogramming share some similarities (beyond their prefix), we will see that the two approaches are quite different. Notably, we will operate on a lower abstraction layer, tackling implementation issues rather than abstracting implementation altogether. Nevertheless, we will see that like MDE, metaprogramming can provide many interesting benefits, like improved performance and early verification of program properties.

Chapter 6. DEVS-MetaSimulator: Enhancing DEVS Simulation through Metaprogramming

I. Introduction

In Chapter 2, we presented some of the numerous DEVS simulators currently available, and pinpointed some features we think could be improved, or at least approached differently. The first one is performance; indeed, the only approaches that have been thoroughly studied so far regarding performance of DEVS simulations are flattening and parallelization.

Flattening consists in limiting the simulation overhead by reducing the number of nested layers. This can be achieved by recursively transforming coupled model into atomic models, by directly connecting atomic models ports, bypassing coupled models, or by using a simulation algorithm that does not rely on a hierarchy of processors. Flattening is implemented in various forms and to various degrees in some DEVS tools, usually during the initialization of simulations.

Parallelization aims at improving the performance of simulations by executing parts of them concurrently. This approach has been studied extensively, and has shown good results. [Jafer 2011] provides a survey of the many works done in this domain, as well as new algorithms

for parallel simulation of DEVS models. However, it should be noted that parallelization is not always an option. For instance, the simulated model can have characteristics such that it does not lend itself to parallelization (the system under study can be inherently sequential, the model components can be so tightly coupled that the gain obtained by performing operations concurrently is counterbalanced by the cost of communication and synchronization between processing units, etc.). Sometimes, parallelization cannot be used because the simulation should run on a single computation resource. This can be the case when multiple resources are already needed to perform independent simulations (e.g. to conduct multiple experiments, or to perform replications to obtain statistically relevant data in the case of stochastic simulations). Consequently, it is important to devise optimizations techniques applicable even to sequential simulation. The approach we propose in this chapter aims at providing very efficient sequential simulations, close to what could be achieved by manually developing custom software specially crafted for the models at hand.

The second issue we want to tackle is that of model verification. To be correct, a DEVS model must fulfill a set of requirements, such as having no zero-delay loops between components or using unique names to denote its ports, for example. Ideally, these constraints should be verified automatically by the environment, and violations should be indicated to the modeler in an explicit way so that he can correct them rapidly. We saw in the previous chapter that such verification can be performed—at least partially—by model editors. However, the use of a full-fledged editor is not always practical, and a practitioner may prefer writing its models in code directly usable with a simulation library. Therefore, the simulator itself should perform the verification, either at initialization or during simulation. This approach is implemented to a certain extent in some DEVS tools, but the checks are often far from exhaustive. More importantly, they happen during execution of the simulator, making the detection of design errors uncertain. In this chapter, we propose an original approach that consists in performing as much verification as possible over the model during compilation, thereby detecting errors very early in the development cycle and with no risk of “missing” them, without even needing to run the simulation.

To improve these two aspects, performance and verification, we propose an approach based on metaprogramming where instead of using a generic simulator capable of simulating any DEVS models, we use a generator of simulators, which creates a specific simulator for each

model we give it. We named this generator DEVS-MetaSimulator (DEVS-MS), but retrospectively, a better name would have been DEVS-SimulatorTemplate. Indeed, like metadata is data about data, like a metamodel is a model of a model, a metasimulator would be a simulator simulating a simulator; here, we rather deal with a simulator template, a “mold” from which an endless number of simulators can be created.

We will start this chapter by presenting the general principle underlying DEVS-MS, explaining what our approach aims at providing, and how. Then, we will present in detail the implementation of DEVS-MS we made with C++ template metaprogramming. Finally, we will study a sample case, using a semantic cache model from a previous work [D’Orazio and Traoré 2009]. This application will allow us to expose the results obtained with our metaprogramming approach.

II. Rationale behind DEVS-MS: specializing simulators for their models

The main idea behind DEVS-MS is to remove as much simulation scaffolding as possible. Indeed, the behavior of a DEVS model is specified by the transition functions of atomic models, along with the interaction between components. Ideally, simulating such a model should boil down to successively invoking transition functions on components. All the other operations performed during simulation are just needed to correctly interpret the model, but bears no relevance to the simulated system.

This simulation overhead results from two things: the ability to define DEVS models in a hierarchical way, thanks to closure under coupling, and the clear separation between model and simulator, allowing a single simulator to handle any DEVS model. However, these two characteristics are only needed as a convenience for the modeler: hierarchical modeling allows him to specify models more easily, possibly reusing models across several projects, and the separation between model and simulator allows him to focus on modeling without worrying about the simulation aspect, leveraging existing implementations to perform the simulation. Even though these properties are essential from a modeling point of view, they are not required to actually simulate a model: given a specific DEVS model, it is possible to

write a specialized simulator for it that does not have a hierarchical structure, nor is capable of simulating other models.

Using such a specialized simulator instead of a generic, universal simulator provides several advantages. The first gain is in performance: since the simulator is dedicated to the model at hand, the simulation overhead can tend to zero, the whole program performing exclusively operations related to the model considered.

The second advantage relates to verification and error checking: the more generic a program is, the more input it can receive, and consequently the more potential errors it needs to take into account and handle. On the contrary, the more specific a program is, the less cases it needs to handle and the easier it is to check. For instance, in our case, a specific DEVS simulator can make the model components communicate directly, without going through generic simulation entities. If a component directly calls a method on another component, one can assert statically that this method effectively exists and that the arguments passed are compatible with the formal parameters. On the other hand, in a generic simulator, it is not possible to know before execution that an event will be sent to a port that actually exists, or that the value is expected by the destination component.

Finally, a last advantage of using a specialized simulator in place of a generic one is that the former does not impose meaningless requirements on the model, such as specifying an external transition function on a model with no input ports, while the latter needs to set these requirements since it must handle all models in a uniform way.

At this point, we made it clear that using specialized simulators provides many advantages over using a single universal simulator for all models. However, having to manually develop a specific simulator for each DEVS model is of course not conceivable. Our proposition is to automatically generate them through metaprogramming, thereby providing the best of both worlds: the practitioner can keep focusing on modeling, then generate a specialized simulator for his model, as close as possible to what he could have written manually.

To clarify what should be the output of our generator of simulators, we provide hereafter some comparisons between a generic implementation and the specialized program that we want DEVS-MS to generate.

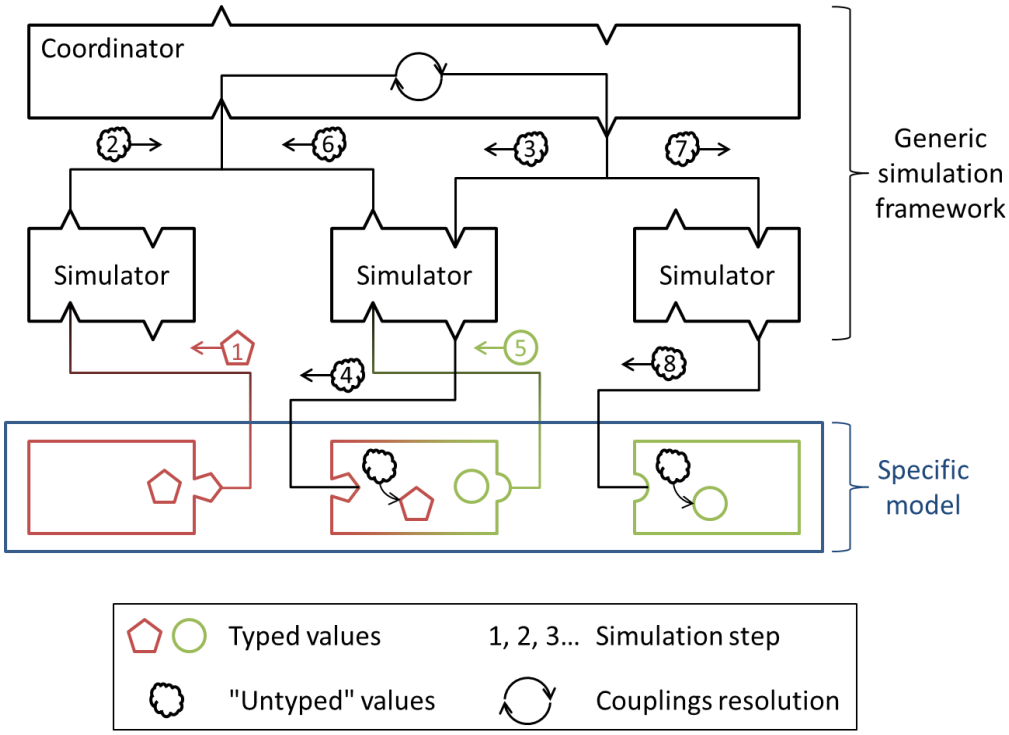


Figure 38. Generic simulation of a specific model.

II.1. Generic interface versus specific interface

First of all, we want the generated simulators to provide interfaces that are specific to the model at hand. In a generic simulator, all simulation entities (processors) have the same interface—and implementation—, regardless of the model they handle. They only obtain information about the model during execution, so they need to expose generic methods and manipulate generic values so as to accommodate every potential model. Figure 38 shows a generic simulation framework handling a specific coupled DEVS model composed of three atomic models, connected in series. The schema depicts the course of events from component to component, flowing through the simulation framework.

All simulators and coordinators in the processor hierarchy are identical (same interface, same implementation): they all have four slots, two for receiving/sending data to their associated component, and two for receiving/sending data to their parent processor. They have no knowledge about the model they simulate, at least not until execution; hence, they manipulate events and their values using some universal base type which carries no information about the actual type of values that will effectively be exchanged throughout the model. At runtime, this information will probably be embedded in the object, but at

compile-time, the type is entirely lost upon entering the simulator. This is schematized in Figure 38, where the value sent by the first component (a red pentagon) has its type erased by the corresponding simulator, before being forwarded to the second component. The second component receives an “untyped” value, which it needs to cast to the appropriate type in order to be able to use it. The same thing happens for the value sent by the second component to the third (a green circle).

There is no problem in this example, but what if the second component expected a green circle and not a red pentagon? This would mean that the couplings specified by the coupled model are incorrect, the input set of the second component being incompatible with the output set of the first. However, this incompatibility would go unnoticed until the second component actually receives an event from the first one. In this case, this would probably occur very soon during the simulation, but in more complex cases, the incorrect coupling could be used only on some occasion, thereby staying undetected for a long time and potentially leading to bugs hard to pinpoint and correct (especially in languages that do not check type casts).

Another noticeable consequence of these uniform interfaces is that all events transit through the same channels. Thereby, the routing of events must occur during execution since the only way to know the destination of an event is to dynamically determine its source and to query the coupling specifications to find out where it should be forwarded. Some implementations interrogate couplings only once, during initialization, and store in each port the list of its influencees. This technique makes the routing of event more efficient, but the connections are still determined dynamically, thereby forbidding further optimizations and leaving the compatibility issue unsolved.

Now, suppose that we could somehow specialize the interfaces and implementations of each processor, so that it adapts perfectly to the component it handles. The simulation framework obtained would be dedicated to the simulation of the model considered, unable to handle different ones. Such a situation is depicted in Figure 39, where the generic framework of Figure 38 has been replaced by a framework entirely specific to the simulated model.

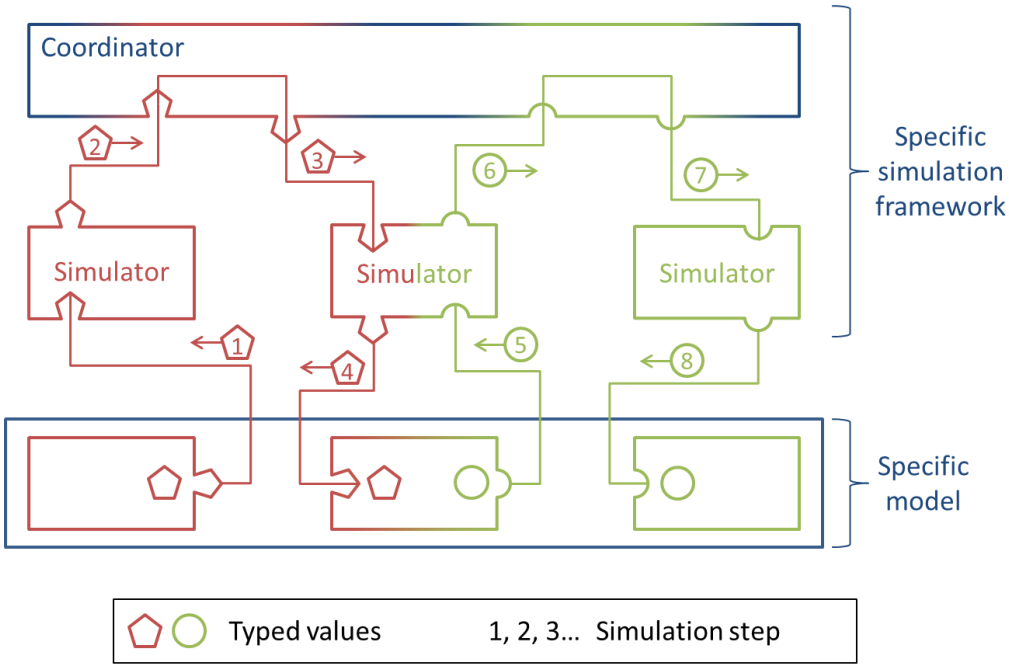


Figure 39. Specific simulation of a specific model.

The first thing to note is that processors are no longer uniform: each of them maps closely to its component, notably in terms of interface. Indeed, each simulator has slots corresponding to the ports of its component. For instance, the first simulator exposes a slot for receiving red pentagons, but since the atomic model has not input port, the simulator does not provide any slot for receiving events from its parent or sending values to the component. Similarly, the second simulator interface allows sending red pentagons to the component and receiving green circles from it, but nothing more.

Such specialization is also applied to the coordinator. Instead of accepting/sending values to its children in a generic way, it provides slots specific to each of its children: a red pentagon for receiving (resp. sending) values from (resp. to) the first (resp. second) simulator, and a green circle for receiving (resp. sending) values from (resp. to) the second (resp. third) simulator.

Thanks to this fine-grained interface, the coordinator does no longer need to resolve couplings dynamically. Instead, it can statically connect its slots according to the couplings specified by the coupled model. This can be seen in Figure 39 where the red pentagon flows from the first simulator to the second without any computation from the coordinator. In fact, the coordinator can be so specialized that it ends up being a mere middle man with no

additional value, simply delegating to its parent or children. In this case, it can be completely removed from the simulation framework, reducing the message passing overhead, as we will see shortly.

Another consequence of that specialization is that the type of values is never lost: it is forwarded throughout the processor hierarchy, removing the need for a universal base type and for runtime casts. Since the actual type of values is now retained and accessible statically, a suitable programming environment can use this information to perform verifications and assert the compatibility of connected ports. If the specialized simulator is written in a statically typed language, no additional work is needed since the compiler will perform all the necessary type checks, allowing us to leverage an existing type system instead of devising a custom one. A major advantage of this approach is that existing type systems are quite rich: in addition to basic equivalence, they handle subtyping, implicit conversions, and so on. By using such a type system, a simulation framework can provide a powerful couplings verification feature without extra work.

II.2. Hierarchical simulation versus flattened simulation

We explained previously how a coordinator specialized for a given coupled model did not need to resolve couplings during execution, and could instead forward events directly to the appropriate processors, without computation. If we increase even more the degree of specialization, up to the point where coordinators do not even have to dynamically compute the tie-breaking function, the coordinators become useless intermediaries that can be safely removed from the processor hierarchy, effectively resulting in a flattened simulator.

Such flattening of the processor hierarchy can greatly reduce the overhead induced by the simulation framework. To get some idea of the gain potentially achievable, consider the simple coupled model in Figure 40. It is composed of two coupled components, each of them containing a single atomic component. Events generated by component “a” are sent to component “b” via coupled components “c1” and “c2”.

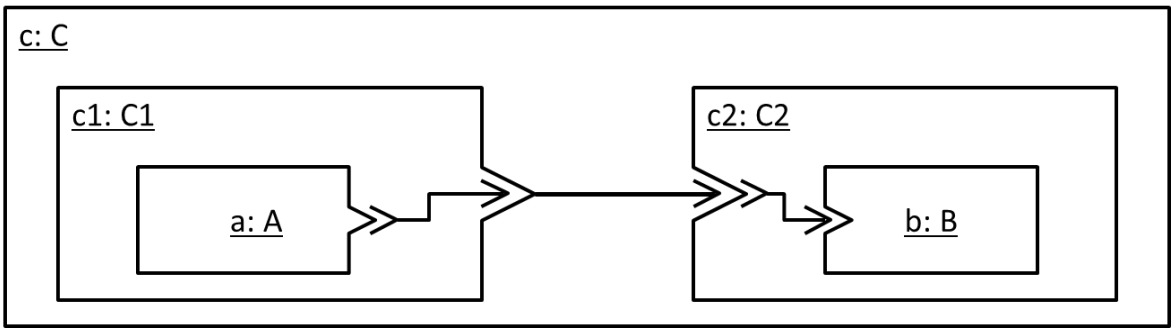


Figure 40. Sample coupled model.

Even though quite contrived, this model will allow us to illustrate the differences between hierarchical and flattened simulation in terms of message passing overhead. First of all, assume a simulation framework that directly implements the algorithms defined in [Zeigler et al. 2000] and exposed in Chapter 2.II.2. In this framework, each component, either atomic or coupled, is associated with a corresponding processor, and each simulation step necessitates numerous roundtrips up and down the processor hierarchy. Consider the sequence diagram in Figure 41, which represents the messages exchanged between entities to process a single internal event.

First, the root coordinator sends an internal state transition message to the main coordinator, `coord_c`. `coord_c` determines which processor should be activated (`coord_c1` in this instance), and forwards the message to it. Similarly, `coord_c1` determines that `sim_a` must be activated, and sends it the message. Only then does the actual computation related to the model begin. `sim_a` queries `a` to obtain the output generated. To forward this event to the appropriate components, `sim_a` passes it to its parent, `coord_c1`, which interrogates the couplings of `c1`. Since the event must "leave" the coupled component, `coord_c1` sends it to `coord_c`. In turn, `coord_c` interrogates the couplings of `c`, thereby determining that the event should be forwarded to `coord_c2`. `coord_c2` does the same and sends the event to `sim_b`, which can at last invoke the external transition function on `b`. When execution returns to `sim_a`, the `a` component can finally undergo its internal transition.

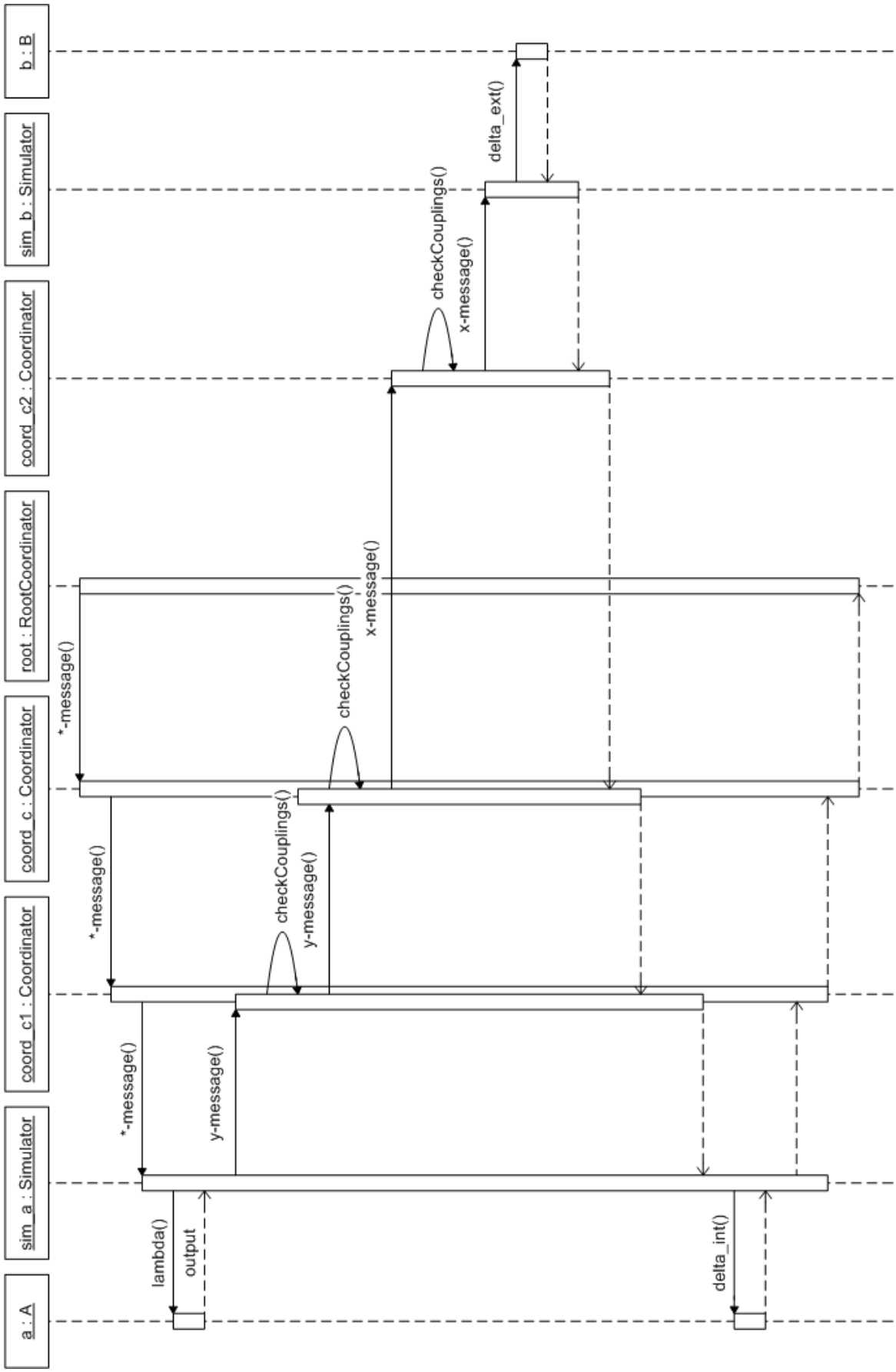


Figure 41. Sequence diagram of processing an internal event with a hierarchical simulator.

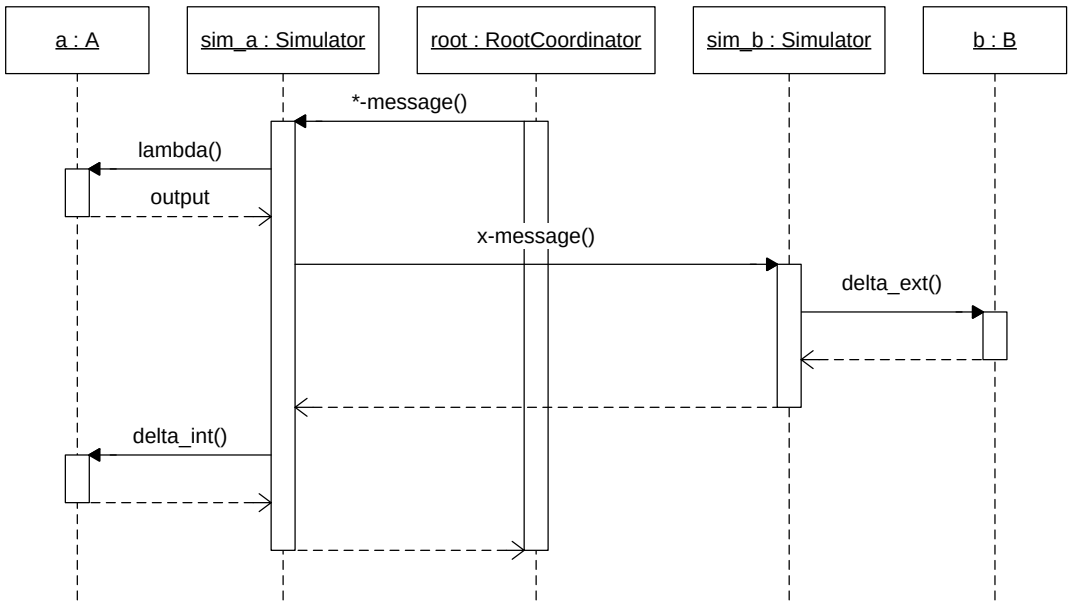


Figure 42. Sequence diagram of processing an internal event with a flattened simulator.

This example shows the important overhead induced by using a hierarchical simulator. Most of the operations are unrelated to the actual behavior of the model, which is mainly specified by the transition functions. Moreover, the sample model used here is very simple, but in more complex cases, the level of nesting can be much more important, leading to an increased simulation overhead. Of course, if the time needed to compute the transition functions is very important, this overhead will be insignificant. However, a common practice in component-based engineering is to start with fine-grained units of works, very cohesive and easily understandable, and to compose more complex systems mainly by coupling components, each layer of abstraction resting on lower ones. It is the same in DEVS modeling: it is easier to develop small atomic models, easy to grasp, and to incrementally build coupled models based on those than it is to specify huge atomic models with complex transition functions. As a consequence, the routing of events from component to component can often account for an important part of the simulation, and therefore should be handled efficiently.

Going back to our example, you can see in Figure 42 a sequence diagram of the same operation as Figure 41, but performed by a flattened simulator.

The difference with the previous version is that all coordinators have been removed from the processor hierarchy. Simulators communicate directly with one another instead of going through layers of intermediaries. The usual way to achieve this is to connect simulators at the beginning of the simulation, during initialization. With this approach, all simulators are identical, they are just initialized differently. In DEVS-MS, we propose a different approach: like we explained before, we will create specialized simulators for each component, meaning that the simulator for `a` will be completely different from the one handling `b`. This way, `sim_a` will "know" that it must send events generated by `a` to `sim_b`. This knowledge will be embedded in the structure of the simulator itself, thereby entirely removing the notion of couplings from the simulation.

This direct connection between simulators provides several advantages: in addition to the type safety evoked previously, it greatly reduces the number of messages needed to process events. Even when the depth of the model is small, like in our example, the gain is significant: here, we divided the number of messages by two. As the depth grows, the gain becomes more and more important since the complexity of the flattened simulator is constant while that of the hierarchical simulator is linear, i.e. the number of operations needed to process an event grows linearly with the number of layers in the model.

Messages exchanges are not the only source of overhead eliminated by direct connection. Since output ports are directly connected to the input ports of their influencees, there is no more need for resolving couplings during simulation. The routing of events is hard coded in the simulation framework, diminishing the amount of computation needed to make components communicate.

In fact, since we strive for minimum overhead, we can remove even more go-betweens from the simulation. Pushing specialization even further, we can merge simulators with their associated components, moving remaining simulation operations to the components (or model operations to the simulators). After doing this, all that is left is a root coordinator that merely activates the various components successively, according to their time of next event. Figure 43 shows the sequence diagram obtained after eliminating all simulation components.

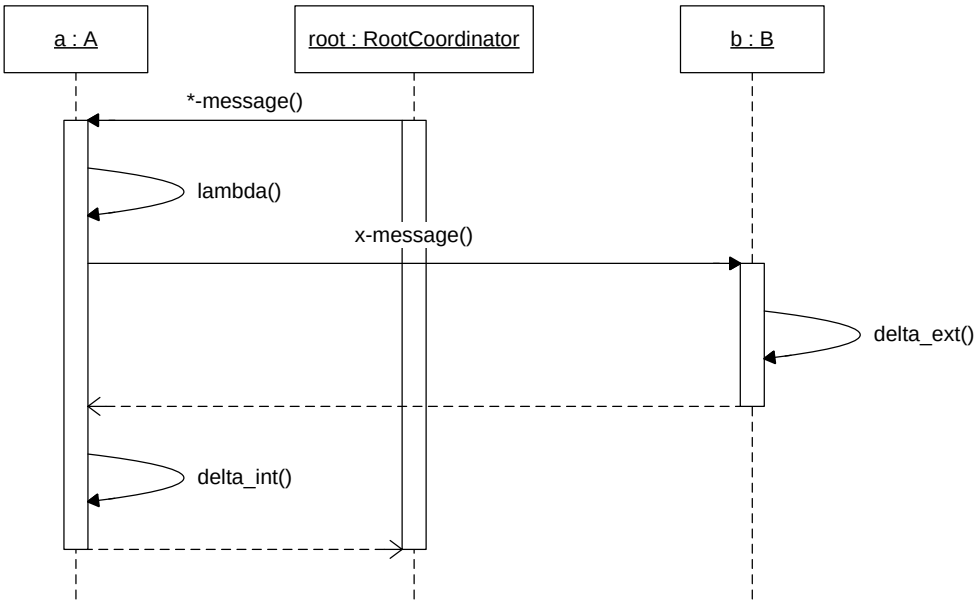


Figure 43. Sequence diagram of a simulation step after elimination of all processors.

Each component is now responsible for keeping track of its times of next and last event, and directly triggers external events on the other components. It should be noted that this optimization is less beneficial than the previous one, as it removes a layer of indirection but no actual computation: the same operations need to be performed, they have only been grouped in a single place instead of being split between component and simulator.

II.3. Sample code of a specialized simulation application

Up to this point, we have exposed the advantages provided by specializing a DEVS simulation for a particular model from a conceptual point of view. To give a clear understanding of what such specialized software would look like, we will now provide a concrete example. To illustrate the discussion, we will use a variant of the now familiar NetSwitch model (cf. Chapter 5.III), embedded into a basic experimental frame composed of a generator and a transducer, as depicted in Figure 44.

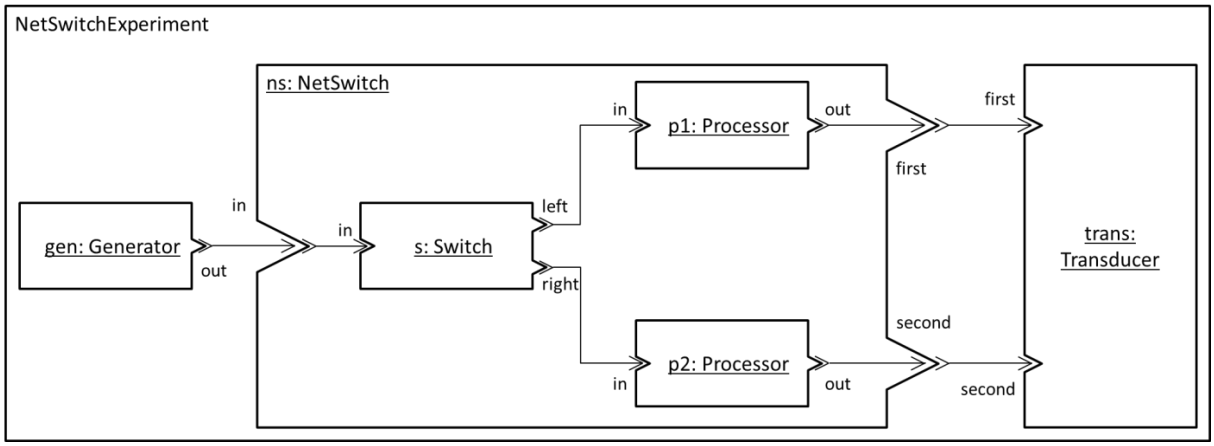


Figure 44. Sample model: NetSwitch with basic experimental frame.

If we wanted to write code especially crafted and optimized for the simulation of this model, we could write a class for each component. For instance, we could represent the `s` component with a class similar to the one shown in Code 58[39].

```
struct s
{
    time   tn;
    string output;
    bool   outputLeft;

    void proceed(time t) // *-message + lambda + delta_int
    {
        if (outputLeft)
        {
            p1.receiveIn(output, t);
        }
        else
        {
            p2.receiveIn(output, t);
        }

        tn = infinity;
    }

    void receiveIn(string value, time t) // x-message + delta_ext
    {
        output = value;
        outputLeft = !outputLeft;

        tn = t; // schedule internal transition immediately
    }
};
```

Code 58. Class specialized for the `s` component.

[39] For the sake of brevity, the codes in this section omit several irrelevant aspects, such as component initialization or data encapsulation.

This code exhibits several points of interest:

1. The component is responsible for keeping track of its time of next event. It updates it after each transition, internal or external.
2. Useless computations are omitted. For instance, the external transition function of the NetSwitch does not use the elapsed time; as a consequence, there is no need for computing it nor for keeping track of the time of last event (except if we want to verify that no synchronization errors occur, but that should never be the case).
3. The switch component directly triggers external transitions into the processors. Depending on its state, it influences either `p1` or `p2`.
4. The types of values received and sent by the component are statically specified in the function signatures, making for greater type safety. The interface of `s` allows only `string`s to be received.

The code for the `p1` component could be rather similar, as shown in Code 59.

```
class p1
{
    time           tn;
    queue<string> jobs;

    bool isIdle() { return jobs.empty(); }

    void proceed(time t)
    {
        trans.receiveFirst(jobs.front(), t);
        jobs.pop();

        tn = isIdle() ? infinity : t + processingTime;
    }

    void receiveIn(string job, time t)
    {
        if (isIdle())
        {
            tn = t + processingTime;
        }

        jobs.push(job);
    }
};
```

Code 59. Class specialized for the `p1` component.

In addition to the remarks already made about the previous code, we can note three things here:

1. In a classical implementation, a queued processor needs to store the time remaining in its current state (σ) and update it upon reception of external events. Here, the processor directly stores its time of next event, so there is need for neither the σ state variable nor the elapsed time nor the time of last event.
2. In the original model, events generated by the processor were sent to a port of the NetSwitch, which was connected to a port of the transducer. Here, the processor sends its output directly to the transducer, without going through the NetSwitch "membrane".
3. The method invoked on the transducer is specific to the receiving port (i.e. `first`): in the `p2` class, the method called would be `receiveSecond` instead of `receiveFirst`. In classical simulation libraries, the destination port is passed as argument to a generic function, which needs to dynamically determine which port receives the event. Here, this information is embedded in the function call itself. The distinction is similar to invoking a function using reflection (`invoke("function")`) versus calling it statically (`function()`).

To understand what this last point implies for the transducer part, consider the possible implementation given in Code 60.

```
class trans
{
    void receiveFirst(string value, time t)
    {
        log("At time " + t + ", the first processor generates " + value);
    }

    void receiveSecond(string value, time t)
    {
        log("At time " + t + ", the second processor generates " + value);
    }
};
```

Code 60. Class specialized for the `trans` component.

This code is quite different from the previous ones. Indeed, there is no `tn`, no `proceed` function, and several external transition functions. There are two reasons for these differences:

1. The transducer exposes two input ports, meaning that it can receive events on either one. To handle these two cases, it defines two different functions, one for each port. The meaning of this is that the component does not need to determine during execution which port received the event: this information is stored in the structure of the program itself.
2. The transducer is passive, it only reacts to external event. As a consequence, it does not need to define an internal transition function since it should never be called anyway. It does not even need a time of next event, since the latter would always be infinity. In a similar way, components with no input ports, such as `gen` in our example, do not need to define external transition functions; they would mean nothing.

We showed that specialized code allowed components to interact directly and handle most of the simulation, without relying on additional processors. However, we still need a "conductor", an entity responsible for successively activating the components until the end of the simulation. This entity is in fact a root coordinator, which can be specialized too. Code 61 gives a possible implementation for such a coordinator.

This code is quite straightforward: the root coordinator successively queries the time of next event of each component, and activate the appropriate one. In this case, we assumed a simple tie-break where `s` has priority over `gen`, which has priority over `p1`, which has priority over `p2` (`trans` will never be imminent, so it is entirely ignored by the coordinator). These simple priorities can be handled by properly ordering the tests, but a more complex select function would imply a bit more work. We will go back to this in the next section.

```
class rootNetSwitchExperiment
{
    void run(time duration)
    {
        while (t < duration)
        {
            t = s.tn;
            component = s;

            if (gen.tn < t)
            {
                t = gen.tn;
                component = gen;
            }

            if (p1.tn < t)
            {
                t = p1.tn;
                component = p1;
            }

            if (p2.tn < t)
            {
                t = p2.tn;
                component = p2;
            }

            component.proceed(t);
        }
    }
};
```

Code 61. Root coordinator specialized for the NetSwitchExperiment model.

These sample codes provide a concrete illustration of the notions exposed previously: specialized interface (statically typed), direct communication between components, no superfluous go-betweens, etc.

They also clearly show that the simulation overhead can be almost completely eliminated: the only simulation-related operations that cannot be removed are those depending on the state of components. Indeed, this state will keep changing during simulation, as opposed to immutable data such as couplings or input/output sets. As a consequence, the operations related to the state of the simulation cannot be “anticipated” like others.

The interesting thing is that there are in fact very few of these operations: basically, the only computation that absolutely needs to be done during simulation is the determination of the

next component to activate, which requires keeping track of the time of next events of each component. Some models also require storing the time of last event of certain components, when their external transition function needs the elapsed time since last event.

In the end, it is possible to come up with highly model-specific software that is almost entirely dedicated to performing computations related to the model, i.e. the computations that are actually of interest to the practitioner. Such software is more efficient but also more "safe" than a generic simulation library: it cannot include incorrect couplings, use wrong identifiers or modify states outside of transition functions.

II.4. Generic modeling and specific simulation: having the best of both worlds

At this point, you might be thinking that we, the authors, completely missed the point of DEVS. Indeed, two of the major strengths of DEVS are the ability to develop models without caring about how they will be simulated, relying on tried and tested tools for performing the simulation, and the ability to hierarchically build models by coupling well-encapsulated components, possibly reusing them in multiple compositions.

These two essential characteristics are lost with the specialization approach we just exposed: to simulate a model, the practitioner must develop a complete simulation application from scratch, possibly introducing programming errors; models cannot be reused at all, since every single component is tightly coupled to its surroundings; and finally, the hierarchical structure of the model is completely erased, making it much harder to understand.

However, the advantages provided by DEVS are most of the times only useful from a modeling point of view; from a simulation point of view, it often does not matter whether the simulators/processors are modular, hierarchical or reusable. Therefore, an interesting approach would be to combine the best of both worlds, allowing DEVS models to be specified as usual, but simulating them with specialized simulators. However, we obviously do not want the burden of creating these specialized simulators to be put on the practitioner. Ideally, this operation should be performed automatically, using some sort of simulator generator: given a DEVS model specification, this generator would transform it into a program dedicated to simulating this particular model, as schematized in Figure 45. An

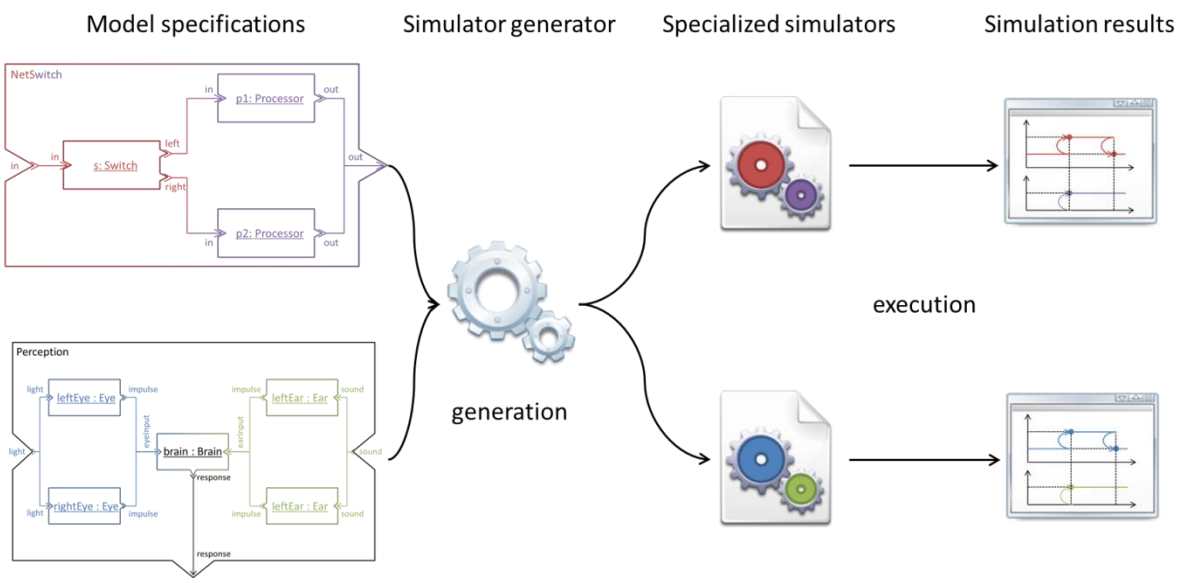


Figure 45. Automatic generation of specialized simulators from model specifications.

interesting way to think about this approach is through the lens of Model-Driven Engineering. Indeed, such generation could be seen as a model transformation, taking as input a model of a DEVS model and producing a model of a simulator[40].

With this solution, the practitioner could keep on focusing on modeling and not simulation, composing models as he sees fit, but the simulation would be performed by a specialized application more efficient and robust than a generic one.

In addition to generating optimized code, such a simulator generator could provide many interesting additional features. Notably, it could perform a limited form of model analysis, for instance checking the model for design errors. This way, such errors would be detected before simulation even begins, thereby reducing the need for debugging and shortening model development times.

Several approaches are conceivable for devising such generator. One of them would be to leverage the DEVS metamodel described in the previous chapter, and use an M2T transformation to generate the specialized code in some GPPL. Such transformation should be rather straight-forward to implement using the MDE tools we presented, and would

[40] In fact, instead of using the metaprogamming solution presented in this chapter, we could have equally implemented such transformation in the Model-Driven DEVS environment we presented in Chapter 5, SimStudio.

deserve being included in SimStudio. However, the approach we propose in this chapter is a bit different, and arguably more challenging: instead of writing a generator from scratch, we will write a C++ template metaprogram that performs parts of the simulation during compilation, effectively making the C++ compiler itself generate the specialized simulation code.

III. Implementation of DEVS-MS with C++ Template MetaProgramming

We implemented the ideas exposed in the previous section in a C++ simulation library called DEVS-MetaSimulator (DEVS-MS). Unlike existing DEVS simulators, DEVS-MS does not handle all DEVS models identically: it specializes itself for each model, at compile-time. The result of this specialization is a piece of software especially dedicated to the simulation of a specific model, providing more robustness and efficiency than a generic simulator. As such, DEVS-MS can be categorized as an active library (cf. Chapter 4.III.2.1), capable of handling any DEVS model but providing the same advantages of using a tailor-made simulator, namely error-proofing and efficiency.

To make the library perform this specialization, we used an approach that consists in displacing operations from runtime to compile-time. The algorithms implemented in DEVS-MS are the same as the abstract simulators [Zeigler et al. 2000] explained in Chapter 2.II.2, but they are split in two distinct parts: one is computed during execution as usual, but the other is computed by the compiler ahead of runtime. Thereby, the executable generated by the compiler contains only the computations that cannot be performed beforehand.

To make this possible, we need to transpose information from runtime to compile-time, i.e. make it static instead of dynamic, so that the compiler can access and manipulate it. In practice, this means we need to rely as much as possible on types and template parameters to represent model characteristics instead of relying on variables, member data or function parameters. This way, we will be able to process these characteristics with template metaprogramming, using the techniques and libraries we presented in Chapter 4.IV.

A preliminary step is to discriminate between parts of the simulation that can be made static, and parts that must stay dynamic. As a matter of fact, this discrimination is purely

arbitrary. For example, we could decide that everything is dynamic, allowing model specifications to be fed to the simulation engine at runtime (e.g. by reading a file). This very generic approach is implemented in some DEVS tools, but leaves no room for the specialization we strive for. At the other extreme, we could decree that every single piece of information about the model must be available at compile-time, including parameters and input data. In this case, the entire simulation can be performed during compilation, leaving nothing to be done during execution. The resulting executable would be extremely efficient, but rather useless as it would always give the same result.

We tried to find a middle-ground between these two extremes based on the usual lifecycle of models. Like any product, a model goes through several phases: conception (definition of the experimental frame, domain analysis, hypotheses formulation …), design (choice of formalism, specification of models …), realization (implementation, VV&T [Balci 1995] …), and eventually exploitation (prediction, exploration …). Usually, the goal of an M&S project is to produce a model, verified and as much validated as possible in its experimental frame, that can be used as a "virtual laboratory" to make experiments about the system under study. Thus, the simulation software produced by DEVS-MS should be exploitable to perform such experiments.

In fact, there is a quite natural boundary between the model and its inputs. When correctly designed, the model is specified once and run many times, with varying parameters and input data. As a consequence, it seems reasonable to decide the model specification to be static, while the rest remains dynamic. This means that once compiled, the model will remain the same and not be modified during execution, a restriction we deem acceptable. Based on this, we can classify model information into two categories, static data (available at compile-time) and dynamic data (available only at runtime). Table 2 summarizes this classification, showing what data we made static for both atomic and coupled models.

In the rest of this section, we will present the different elements of DEVS-MS. All data structures and algorithms in our simulation library are composed of both compile-time and runtime parts, sometimes intertwined; to clarify explanations, we will use pseudo-code highlighting these two stages. Static parameters will be formatted in red italics and enclosed

Static (compile-time)	Dynamic (runtime)
Atomic model	

• Input set (names and types of input ports)
• Output set (names and types of output ports)
• State set (names and types of state variables)
	• Values on input ports
	• Values on output ports
	• Values of state variables
	• Functions depending on these values[41]:
	  ○ Internal transition function
	  ○ External transition function
	  ○ Output function
	  ○ Time advance function

Coupled model	
• Input set (names and types of input ports) • Output set (names and types of output ports) • Component set (names and models of components) • Couplings • Tie-breaking function	• Values on input ports • Values on output ports • Components instances

Table 2. Classification of model information in static and dynamic data.

in angle brackets while dynamic parameters will be plain text surrounded by parentheses. Similarly, computations performed at compile-time will be denoted by red italics.

III.1. Message

The `Message` class, summarized in Code 62, represents an event in transit between components. It stores not only the value associated with the event, but also a port identifier, denoting the source or the destination of the event.

[41] It is a bit incorrect to say that functions of atomic models are dynamic data: in fact, their *definition* is known statically, but their *evaluation* can only be done at runtime since their arguments are not known until then. So strictly speaking, the functions themselves are indeed static data, but as they cannot be used by the metaprogram, we categorized them as dynamic to distinguish them from functions that can be evaluated statically, such as the tie-breaking function.

```
class Message<Ports = {(PortName0, PortType0), ..., (PortNameN, PortTypeN)}>
{
    PortsMap = {PortName0 => PortType0, ..., PortNameN => PortTypeN};

    result_of
    {
        IsValueOnPort<PortName> = PortsMap.HasKey<PortName>;
        GetValueOnPort<PortName> = PortsMap.GetValue<PortName>;
        PutValueOnPort<PortName, PortType> = Message<Ports U {(PortName, PortType)}>;
    }

    fusion::map<Ports> values = {PortName0 => PortType0, ..., PortNameN => PortTypeN};

    Message();
    Message<Sequence>(Sequence values);
    Message(PortType0 val0, PortType1 val1, ..., PortTypeN valN);

    bool isValueOnPort<PortName>();
    result_of::GetValueOnPort<PortName> getValueOnPort<PortName>();
    result_of::PutValueOnPort<PortName, PortType>
        putValueOnPort<PortName, PortType>(PortType val);
}

EmptyMessage = Message<>;
```

Code 62. Message class (pseudo-code).

We designed the Message class so that it can be reused for Parallel-DEVS. Therefore, we allowed messages to carry several values on multiple ports, even though this never happens in Classic DEVS. A message is thus parameterized with a static set of ports (an MPL sequence—cf. Chapter 4.IV.3.4) that specifies the names and types of the ports concerned. In Classic-DEVS, all messages will be parameterized with a single port and will contain a single value.

Since we want to manipulate port identifiers during compilation, we need to represent them so that they are statically available. Remember that the type of data that can be manipulated by a C++ metaprogram is rather restricted: template parameters can only be types, integral values, enumerators, and a specific kind of pointers. Enumerations could have been appropriate to define sets of identifiers, but they do not blend in very well with classical metaprogramming, which mainly manipulates types. Thus, we chose to use types to represent all identifiers in DEVS-MS. To identify a model element, one can use an existing type or create a dummy one. For instance, Code 63 declares a new port identifier “p1”, using a dummy struct, and creates a message with values on ports “bool” and “p1”.

```
struct p1;
...
Message< mpl::vector<
    mpl::pair<bool, bool>, // port "bool" with bool value
    mpl::pair<p1, int>     // port "p1" with int value
> > msg(false, 42);        // "bool" contains false, "p1" contains 42
```

Code 63. Using types as identifiers.

The `Message` class stores values by using a Fusion map (cf. Chapter 4.IV.3.5), which associates port identifiers (static data) with the values held on those ports (dynamic data). Looking up a value on a given port is thereby performed at compile-time: everything happens as if `Message` contained a data member for each port.

Like Fusion data structures, Message is a mixed class that involves both compile-time and runtime data and computations. As a consequence, most of the operations it provides can be considered from either point of view, either as a metafunction for use in a metaprogram, or as a regular function to invoke during execution. To distinguish between the two, we adopted the Fusion convention of encapsulating the compile-time version into a `result_of` namespace. To facilitate compile-time computations, the list of ports is transformed into an MPL map associating each port name with the corresponding type.

The operations provided are:

- `isValueOnPort`, which returns whether the Message contains a value on the given port (at compile-time in the form of a boolean wrapper (cf. Chapter 4.IV.3.4), at runtime as a boolean value).
- `getValueOnPort()`, which returns at compile-time the type of value held by the given port, and at runtime the actual value held on the port. Note that trying to invoke this function of a message that does not hold a value on the given port will result in a compiler error. Indeed, it would be like trying to access a data member that does not exist. We will see later that this has some consequences when writing external transition functions.
- `putValueOnPort`. The behavior of this function depends on how it is used: when used to modify the value on an existing port, it modifies the value in the message itself, and returns the message; when used to add a value on a new port, it creates a

new message from the original one and adds the new value. Indeed, in the latter case, the type of the message is modified, so it is not possible to do the modification in-place. The compile-time version of this function can be used to determine the resulting type, as shown in Code 64.

```
EmptyMessage empty;
EmptyMessage::result_of::PutValueOnPort<p1, int>::type msg =
    empty.putValueOnPort<p1>(42);
// can be simplified with C++11 auto:
// auto msg = empty.putValueOnPort<p1>(42);
msg.putValueOnPort<p1>(12);
```

Code 64. Combining functions and metafunctions.

Finally, the `Message` class provides several constructors to create a message with default-constructed values, from a Fusion sequence or from multiple parameters (as in Code 63 above). Some other operations are provided for use in P-DEVS, but are not relevant here.

The `Message` class plays a primordial role in DEVS-MS as it carries essential information for the resolution of couplings. Indeed, since port information is embedded in the message type itself, the processors will be able to route events statically, as we described in the previous section. It also guarantees static type safety, ensuring that any attempt to send incorrect values on a port will result in a compiler error. However, this class is mostly used internally by the simulation library; the practitioner seldom needs to manipulate it explicitly, except in the definition of external transition functions, as we will see later.

III.2. Models

Like many simulation libraries, DEVS-MS contains some classes for representing atomic and coupled models, which are mainly meant to be used by practitioners as base classes for their model specifications. However, these classes are in fact just a convenience for the user: there is no dynamic polymorphism involved in DEVS-MS, only static polymorphism (cf. Chapter 4.IV.2.3). Thus, a model does not need to inherit from a particular base class in order to be simulated by the library, it just needs to provide the appropriate interface. As a matter of fact, the following classes could be defined as mere concepts [Gregor et al. 2006] specifying the requirements a type must fulfill to be usable in DEVS-MS.

```
class Model<
    InputPorts = {(IPortName0, IPortType0), ..., (IPortNameN, IPortTypeN)},
    OutputPorts = {(OPortName0, OPortType0), ..., (OPortNameN, OPortTypeN)}
>
{
    InputPortsMap = {IPortName0 => IPortType0, ..., IPortNameN => IPortTypeN};
    OutputPortsMap = {OPortName0 => OPortType0, ..., OPortNameN => OPortTypeN};

    result_of
    {
        MessageOnInputPort<PortName> =
            Message<{(PortName, InputPortsMap.GetValue<PortName>)}>;
        MessageOnOutputPort<PortName> =
            Message<{(PortName, OutputPortsMap.GetValue<PortName>)}>;
    }

    result_of::MessageOnInputPort<PortName>
        createMessageOnInputPort<PortName>(InputPortsMap.GetValue<PortName> val);

    result_of::MessageOnOutputPort<PortName>
        createMessageOnOutputPort<PortName>(OutputPortsMap.GetValue<PortName> val);
}
```

Code 65. Model class (pseudo-code).

Nevertheless, we decided to provide base classes for two reasons: firstly, it greatly facilitates the use of the library, by providing most of the required interface as well as some helper functions for the practitioner; secondly, these base classes can perform a number of verifications at compile-time, based on their static parameters. These verifications could also be performed in the simulation classes, but that would make the detection of errors a bit delayed and a bit distant from the model code. Consequently, it is best to use these base classes, even though it is not mandatory.

The main classes provided are `AtomicModel` and `CoupledModel`, but we factorized some code in a third class, `Model`.

III.2.1. Model

The base class Model, summarized in Code 65, factorizes code related to ports so that it can be reused in `AtomicModel` and `CoupledModel`. Its main purpose is to store the input and output set of a model, as lists of ports.

Model is parameterized statically by a list of input ports and a list of output ports (an MPL sequence). Each element of these lists is a pair (an MPL pair) containing a name identifying

the port and a type specifying the set of values supported by the port. To facilitate further processing, these two lists are transformed at compile-time into associative sequences mapping port names to the corresponding types (MPL maps). Doing so also allows us to verify the uniqueness of port names in the model: if a model is defined with two ports having the same name, compilation will fail with an explanatory error message.

In addition to this, the `Model` class provides two helper member functions for use by the simulation processors and the user. These member functions allow creating `Message` instances simply by providing a port name and a value. This way, there is no need to specify a full list of ports, a single identifier suffices. Moreover, the function accepts only values of the correct type by looking up the appropriate one in the maps of ports. As before, these functions are provided both in compile-time fashion, returning the type of `Message` to be created, and runtime fashion, actually creating the `Message` instance and returning it. As we will see later, the `createMessageOnOutputPort` function can be used by the practitioner when defining output functions in atomic model specifications. Normally, he should never need to create input messages, but this function is provided for use by the coordinators. We will see that the metafunction can also be used when defining external transition functions.

III.2.2. AtomicModel

We established in Table 2 that few elements of atomic models can be made available at compile-time without sacrificing the usefulness of the resulting executable. As a consequence, the `AtomicModel` class of DEVS-MS is rather small, as can be seen in Code 66.

```
class AtomicModel<InputPorts, OutputPorts>
    : Model<InputPorts, OutputPorts>
{
    time e;

    AtomicModel(time e = 0);
}
```

Code 66. AtomicModel class (pseudo-code).

`AtomicModel` takes lists of input and output ports and simply forwards them to `Model`, inheriting the nested typedefs and the helper functions it defines, as well as taking advantage of the model verifications it performs.

In addition to port specifications, we could have parameterized `AtomicModel` with a list of state variables. However, the state of model is never needed by the simulation processors. Consequently, there is no need to make this information available to the simulation layer[42], so we chose to let the practitioner decide how he wants to store the model state. Most of the time, this would be done through member data in the user class.

More surprisingly, `AtomicModel` contains nothing related to time advance, output and transition functions. If we had used "classic" (dynamic) polymorphism, `AtomicModel` would have defined abstract methods, which would have been overridden by the practitioner in derived classes. However, in DEVS-MS, we used static polymorphism (cf. Chapter 4.IV.2.3), meaning that the dispatch is performed at compile-time and not at runtime. As a consequence, specifying the functions of atomic models through virtual methods is not needed. In fact, it could even incur some overhead, with no additional gain apart from making the interface of atomic models explicit.

However, even if `AtomicModel` defines no virtual methods, the user still needs to define the appropriate ones so that his classes can be handled by the simulation layer. Failing to provide the correct interface would result in a compiler error. A DEVS-MS class for an atomic model must provide the following member functions:

- `time ta() const` – the time advance function. Since it is not allowed to modify the model state, it must be marked as const.
- `void lambda(Simulator & sim) const` – the output function. This member function needs to be defined only for models with output ports. Like the time advance function, the output function must not modify the model state, so it must be const as well. For technical reasons, the lambda function cannot return the output directly; instead, we use a kind of callback by passing it an instance on which a method can be invoked to "return" the output. This point is explained in more detail in the section dealing with the `Simulator` class (cf. Chapter 6.III.4.1.2).
- `void deltaInt()` – the internal transition function.

[42] There are legitimate reasons for breaking model encapsulation and making them expose their state: for instance, a simulation library may provide automatic logging of the state of components, or (de)serialization features to stop and restart simulations. However, none of this is implemented in DEVS-MS yet, so for now we stick with high encapsulation.

- `void deltaExt(Message x)` – the external transition function. This member function needs to be defined only for models with input ports. As we will see in the next section, there can be multiple external transition functions to handle the reception of events on separate ports. The simulation layer statically determines the correct function to call.

In the end, the only addition of `AtomicModel` to `Model` is the definition of a data member for storing the elapsed time since the last event. This time is computed by the simulator before invoking external transition functions, but by storing it in components, we make it possible to initialize it to some positive value. This can be useful for testing the behavior of external transition functions under varying conditions, or for initializing the state of a model to a previously stored one.

III.2.3. CoupledModel

In contrast, the CoupledModel class is almost exclusively made of static data and computations, as illustrated in Code 67.

Like `AtomicModel`, `CoupledModel` is parameterized by a list of input ports and a list of output ports, which are forwarded to `Model` through inheritance. In addition to these, `CoupledModel` also needs a list of components, a list of couplings and a definition of the tie-breaking function.

The list of components is quite similar to a list of ports: it must be an MPL sequence of MPL pairs, each pair specifying a component identifier along with the corresponding model. This sequence is statically transformed into an MPL map to facilitate its processing. To store components at runtime, `CoupledModel` defines a Fusion map that associates a static component name with the corresponding runtime instance. In fact, the coupled model only stores references to components, not components themselves. This implies that the components must be created prior to the creation of the coupled model instance, and be provided upon construction. The coupled model could have created its own components, but that would have made parameterization and initialization more complex. A common solution is to use a subclass of `CoupledModel` that stores the components as data

```
class CoupledModel<
    Components = {(CompName0, CompType0), ..., (CompNameN, CompTypeN)},
    Couplings = {
        ((srcComponent0, srcPort0), (destComponent0, destPort0)),
        ...,
        ((srcComponentM, srcPortM), (destComponentM, destPortM))
    },
    Select = {{cX, c1, ..., cN}, ..., {cY, ..., cZ}},
    InputPorts,
    OutputPorts

>   : Model<InputPorts, OutputPorts>
{
    ComponentsMap = {CompName0 => CompType0, ..., CompNameN => CompTypeN};

    fusion::map<Components> components =
        {CompName0 => CompType0, ..., CompNameN => CompTypeN};

    CoupledModel(CompType0 c0, ..., CompTypeN cN);
}
```

Code 67. CoupledModel class (pseudo-code).

members, but this is not mandatory: the `CoupledModel` class can also be used directly, without subclassing it.

Couplings are specified with an MPL sequence containing a collection of couplings. Each coupling is a pair source/destination, where both source and destination are specified with a pair port/component. For external input couplings and external output couplings, a special identifier, `This`, is used to denote the coupled model itself. This way, the practitioner can specify all couplings uniformly, using `This` when a coupling relate to ports of the coupled model.

For the tie-breaking function (select), we used the same approach as in SimStudio, described in Chapter 5.II.2.1.7: the select function is defined by an ordered sequence of rules, each rule being a sequence of components that specify which component should be activated when all components in the rule are imminent. The only difference with the SimStudio approach is that to shorten select specifications, we decided that the component to activate would be the first of the sequence, instead of having both a set of components and the selected component. For instance, this select function from Chapter 5.II.2.1.7

$$selectRules = \left(\left(\{c_1, c_2, c_3\}, c_3\right), \left(\{c_1\}, c_1\right), \left(\{c_2\}, c_2\right)\right)$$

would be represented in DEVS-MS as

$$selectRules = \big((c_3, c_1, c_2), (c_1), (c_2)\big)$$

Note that this approach intrinsically enforces the constraint that the selected component must appear in the set of imminent components. Moreover, we also assert at compile-time that all components appearing in the select function belongs effectively to the coupled model at hand.

Several other requirements are checked in `CoupledModel`, always at compile-time. Regarding components, we assert the uniqueness of their identifiers: in a given coupled model, two components cannot have the same name. We also check that all component types are actually DEVS models and not some random types. When it comes to couplings, we verify that all sources and destinations are correct, i.e. that each pair component/port is made of a component belonging to the coupled model (or `This`) and of a port belonging to the component (or the coupled model). We also make sure that couplings do not define a direct feedback loop by connecting a component to itself, a pattern that is disallowed by the formalism.

The port compatibility constraint is implicitly enforced by the C++ type system itself: since all functions parameters are statically typed, based on port definitions, attempting to connect two ports with incompatibles types would be equivalent to calling a method with an incorrect argument, and would result in a compiler error. Relying on the C++ type system provides several advantages: it is not only possible to connect two ports with the same type, but also any ports for which a conversion from source to destination is possible. This encompasses up-casts (derived-to-base conversions), user-defined conversions (through constructors or conversion functions), and standard promotions (e.g. converting a char to an int, or a float to a double). In the latter case, the compiler will even warn us if a coupling may incur a loss of information (for instance when converting a double to an int).

In the end, the only dynamic content hosted by the coupled model is the component map. At runtime, it becomes a mere container with no behavior at all. Consequently, coupled models could be removed from the resulting executable without any loss. By studying the

```
mpl::vector<
    mpl::pair<
        mpl::pair<devs::This,   netswitch::In>,
        mpl::pair<netswitch::S, switch::In>
    >,
    mpl::pair<
        mpl::pair<netswitch::S,  switch::Left>,
        mpl::pair<netswitch::P0, processor::In>
    >,
    mpl::pair<
        mpl::pair<netswitch::S,  switch::Right>,
        mpl::pair<netswitch::P1, processor::In>
    >,
    mpl::pair<
        mpl::pair<netswitch::P0, processor::Out>,
        mpl::pair<devs::This,    netswitch::First>
    >,
    mpl::pair<
        mpl::pair<netswitch::P1, processor::Out>,
        mpl::pair<devs::This,    netswitch::Second>
    >
>
```

Code 68. Possible representation of the NetSwith couplings in DEVS-MS.

assembly code generated by the compiler on some test models, we found that the optimizer effectively performed this removal by inlining all function calls on coupled model instances.

III.3. Utility macros

The classes we just described take many template parameters. These parameters must be MPL sequences, often containing MPL pairs or even other sequences. The use of such metaprogramming constructs can quickly hinder code readability, and is quite cumbersome for the user. For instance, the couplings of the NetSwitch model from Figure 34 could be represented by the MPL vector in Code 68.

The actual coupling information is buried in a lot of boilerplate code, making it hard to understand at first glance. The same kind of code is needed for representing most of the static data. To reduce clutter, we leveraged Boost.Preprocessor to define a collection of utility macros to hide the metaprogramming constructs from the practitioner, and to make the syntax more user-friendly.

III.3.1. DECLARE_PORT/DECLARE_COMPONENT

DECLARE_PORT and DECLARE_COMPONENT are in fact two different names for the same macro, which is used to declare a compile-time identifier. To avoid naming collisions, it is a good practice to embed identifiers into namespaces. The macro allows that by taking two parameters: a list of namespaces, and the actual identifier to be defined. For instance, Code 69 defines the identifier "In" in the namespace "models::netswitch".

```
DECLARE_PORT((models) (netswitch), In) // defines the identifier models::netswitch::In
```

Code 69. Sample use of the DECLARE_PORT macro.

This macro does two things: first, it declares a dummy type representing the identifier; second, it specializes a class template that maps compile-time identifiers to runtime strings. Thanks to this, we can easily transform any static identifier into a string that can be used during execution, as illustrated in Code 70.

```
std::string id = devs::typeToName<models::netswitch::In>::value; // id == "In"
```

Code 70. Conversion from compile-time identifier to runtime string.

III.3.2. PORTS

The PORTS macro can be used to define a list of ports, to be used as argument to AtomicModel or CoupledModel. It takes a list of pairs identifier/type. For instance, the output ports of the NetSwitch model could be defined as in Code 71 (assuming the values handled by the NetSwitch are integers):

```
PORTS( ((netswitch::First, int)) ((netswitch::Second, int)) )
// list with two integer ports, First and Second
```

Code 71. Sample use of the PORTS macro.

III.3.3. COMPONENTS

The COMPONENTS macro is identical to the PORTS one: it takes a list of pairs identifier/type and generates the corresponding MPL sequence representing the set of components. Code 72 defines the component set of the NetSwitch model.

```
COMPONENTS(
    ((netswitch::S0, Switch))
    ((netswitch::P0, Processor))
    ((netswitch::P1, Processor))
) // three components: a switch S0 and two processors P1 and P2
```

Code 72. Sample use of the COMPONENTS macro.

III.3.4. COUPLINGS

COUPLINGS can be used to facilitate the specification of couplings. They are specified as explain before, as a list of pairs source/destination, but all the MPL boilerplate is generated by the macro. Using this macro, the couplings defined in Code 68 become much simpler to write and read, as shown in Code 73.

```
COUPLINGS(
    (( (devs::This,    netswitch::In),  (netswitch::S,  switch::In) ))
    (( (netswitch::S,  switch::Left),   (netswitch::P0, processor::In) ))
    (( (netswitch::S,  switch::Right),  (netswitch::P1, processor::In) ))
    (( (netswitch::P0, processor::Out), (devs::This,    netswitch::First) ))
    (( (netswitch::P1, processor::Out), (devs::This,    netswitch::Second) ))
) // This.In -> S.In ; S.Left -> P0.In ; ...
```

Code 73. Sample use of the COUPLINGS macro.

III.3.5. SELECT

Similarly, the SELECT macro makes the specification of the tie-breaking function clearer by hiding all MPL code. Code 74 shows a specification corresponding to the sample select function we used previously, composed of three rules.

```
SELECT(
    ( (model::C3) (model::C1) (model::C2) )
    ( (model::C1) )
    ( (model::C2) )
) // if {C1, C2, C3} imminents -> C3 ; else, C1 < C2 < C3
```

Code 74. Sample use of the SELECT macro.

III.4. Processors

Now that we presented the modeling-related entities of DEVS-MS, it is time to dive into the core of the library, that is the simulation layer. As we said previously, the algorithms implemented in DEVS-MS maps almost directly to the abstract simulators presented in

```
class Simulator<ComponentName, AtomicModel, ParentProcessor>
{
    AtomicModel component;
    ParentProcessor parent;
    time tl;
    time tn;

    Simulator(AtomicModel component, ParentProcessor parent);

    void init(time t); // i-message
    void proceed(time t); // *-message
    if <! AtomicModel.InputPorts.IsEmpty>
    {
        void forward<DestComponentName, Message>(Message x, time t); // x-message
    }
}
```

Code 75. Simulator class (pseudo-code).

Chapter 2.II.2, with the difference that some computations are moved from runtime to compile-time. Consequently, DEVS-MS relies on the three usual types of processors for simulating models: simulators, coordinators and root coordinator.

III.4.1. Simulator

Each atomic component in a model is associated to a simulator, i.e. an instance of the `Simulator` class summarized in Code 75.

The simulator is quite simple, because it cannot perform many computations at compile-time. It takes three static parameters: the identifier of the component it handles, the type of the component (which must be an atomic model) and the type of its parent in the processor hierarchy (which is either a coordinator or a root coordinator).

At runtime, the simulator holds a reference to its component, a reference to its parent, the time of last event and the time of next event. It provides an initialization function, an internal transition function, and an external transition function. An interesting point about the latter is that it is defined only if the component handled by the simulator has input ports; otherwise, the function does not exist, thereby making sure that any attempt to send an event to a component without input ports will result in a compiler error. To achieve this, we use SFINAE (cf. Chapter 4.IV.1.3) together with inheritance to conditionally include the member function.

We will now explain in detail the behavior of each of these functions.

III.4.1.1. Initialization

The initialization of the simulator involves only runtime computations. Consequently, it is a straight-forward mapping of the usual initialization algorithm, as shown in Code 76.

```
void init(time t)
{
    tl = t - component.e;
    tn = tl + component.ta();
}
```

Code 76. Simulator – Initialization algorithm (pseudo-code).

III.4.1.2. Internal transition

The handling of internal transitions is a bit more interesting. Consider the pseudo-code of the `proceed` function given in Code 77:

```
void proceed(time t)
{
    assert(t == tn);

    if <! AtomicModel.OutputPorts.IsEmpty>
    {
        parent.forward<ComponentName, Message>(component.lambda());
    }
    component.deltaInt();
    tl = t;
    tn = tl + component.ta();
}
```

Code 77. Simulator – Internal transition algorithm (pseudo-code).

This function uses a compile-time test to determine whether it needs to invoke the lambda function of the component. If the atomic model has no output ports, it cannot generate any output, so there is no need invoking its output function. Thanks to this, a practitioner creating a model without output ports does not even need to define a lambda function: it would never be used by the simulator.

If the atomic model does have output ports, the simulator invokes its lambda function and forwards the result to its parent, along with the name of the component (the source

component of the event) and the message type (which embeds information about the source port).

In fact, this step is a bit more complicated. Indeed, the type of output generated by the component depends on its state, which is composed of dynamic data. Consequently, it is not possible to define a return type for the lambda function, since this return type would vary during execution. To solve this issue, we use some sort of callback by passing a reference to the simulator to the lambda function, and make the atomic model sends its output by invoking a member function template on the simulator.

Practically, this implies adding a callback function in the simulator, as shown in Code 78. Once again, this function is only defined when the atomic model has some output ports.

```
class Simulator<ComponentName, AtomicModel, ParentProcessor>
{
    SelfType = Simulator<ComponentName, AtomicModel, ParentProcessor>
    [...]
    if <! AtomicModel.OutputPorts.IsEmpty>
    {
        void send<Message>(Message output);
    }
    [...]
}
```

Code 78. Simulator – Callback function for generating outputs (pseudo-code).

The `proceed` function is modified to look like Code 79. The simulator passes itself to the component so that the latter can invoke the callback function on it.

```
void proceed(time t)
{
    [...]
    if <! AtomicModel.OutputPorts.IsEmpty>
    {
        component.lambda<SelfType>(self);
    }
    [...]
}
```

Code 79. Simulator – Modification of the internal transition algorithm (pseudo-code).

Finally, in lambda functions, the generation of an output is performed by calling the callback function, as in Code 80:

```
void lambda<Simulator>(Simulator sim)
{
    [...]
    if (...)
    {
        sim.send<MessageP1>(createMessageOnOutputPort<p1>(42));
    }
    else
    {
        sim.send<MessageP2>(createMessageOnOutputPort<p2>("foo"));
    }
    [...]
}
```

Code 80. Sample lambda function using the callback mechanism (pseudo-code).

For this to be possible, lambda functions must be defined as function templates, parameterized with the simulator type. The callback function is called with a message created on some port, along with its type. Thankfully, in real code, the practitioner does not need to explicitly specify the template parameter for the message type: the compiler can deduce it from the argument, making the code less cluttered.

III.4.1.3. External transition

Like the initialization function, the external transition function cannot perform many operations at compile-time. Thereby, the algorithm is very classic, as shown in Code 81: the simulator computes the elapsed time since the last event, invokes the external transition function on the component, and finally updates the times of last and next event.

```
void forward<DestComponentName, Message>(Message x, time t)
{
    assert(tl <= t <= tn);

    component.e = t - tl;
    component.deltaExt<Message>(x);
    tl = t;
    tn = tl + component.ta();
}
```

Code 81. Simulator – External transition algorithm (pseudo-code).

Nevertheless, even though it performs all its computations dynamically, this function still manipulates some information at compile-time. Its first static parameter is the name of the component to which the event is addressed. For simulators, this parameter is useless as the

destination component is necessarily the one handled by the simulator, but having this parameter allows coordinators to handle both simulators and coordinators homogeneously, as we will see later.

More importantly, the external transition function is also parameterized by the `Message` type, which notably embeds the name of the destination port. Since this information is forwarded to the δ_{ext} function, the model can perform compile-time dispatching based on the port receiving the event. Practically, this means that a practitioner can write several δ_{ext} functions for handling different cases, and that the correct one will be chosen by the simulator at compile-time.

III.4.2. Coordinator

Unlike simulators, DEVS-MS coordinators are almost entirely composed of compile-time data and computations. The content of the Coordinator class is summarized in Code 82.

```
class Coordinator<ComponentName, CoupledModel, ParentProcessor>
{
    ProcessorsMap = CreateProcessorsMapFromComponentsMap<CoupledModel.ComponentsMap>;

    Couplings = CoupledModel.Couplings.ReplaceAll<This, ComponentName>;
    CouplingsMap =
        { srcComponent => { srcPort => {destComponent => destPort, ...}, ...}, ...};

    fusion::map<ProcessorsMap> processors =
        {CompName0 => ProcessorType0, ..., CompNameN => ProcessorTypeN};
    ParentProcessor parent;
    time tl;
    time tn;

    Coordinator(CoupledModel component, ParentProcessor parent);

    void init(time t);
    void proceed(time t);
    void forward<SrcComponentName, Message>(Message msg, time t);
}
```

Code 82. Coordinator class (pseudo-code).

Coordinators have the same parameters as simulators, i.e. the name of the component it handles, the corresponding coupled model, and the type of its parent in the processor hierarchy. Based on these parameters, the coordinator initializes several compile-time data structures to facilitate further processing.

First of all, the coordinator creates a map of processors that associate component names to the type of processor needed to handle said components. To do so, it iterates over the components map of the coupled model and determines for each component which processor it needs (either a simulator or a coordinator, correctly parameterized). This apparently simple operation in fact involves quite a bit of metaprogramming, but explaining it in detail would be a bit tedious. This MPL map is then transformed into a Fusion map that will hold the processor instances during runtime.

The next substantial operation performed by the coordinator at compile-time relates to couplings. Remember that couplings specifications in coupled models can contain a placeholder named `This` to denote the model itself, allowing homogeneous definition of EIC, IC and EOC. When a coupled model is instantiated as a component, the placeholder needs to be replaced with the actual name of the component. After doing this transformation, the coordinator creates an alternative representation of the couplings that is easier to query when forwarding an event. In this representation, each component is associated to a map that associates ports with destinations. Each destination is a pair component/port, where a component can appear only once as destination for a given source (in Classic-DEVS, it is illegal to connect an output port to two input ports of the same component as that would imply simultaneous reception of two events). Concretely, we end up with a map of component names and maps of port names and maps of component names and port names, which provides a convenient way of resolving couplings in spite of being awkward to describe in plain English.

In addition to these compile-time members, Coordinator also contains some dynamic data, namely the list of its child processors as a Fusion map, a reference to its parent processor, and the time of last and next event. An interesting point to note is that it does not store any reference to a coupled model instance. Indeed, the whole coupled model specification is passed statically to the coordinator; the only time a coordinator interrogates its coupled component at runtime is during the construction of the processor hierarchy, and it boils down to obtaining each individual components of the coupled model pass them to the corresponding processor. The processor hierarchy is constructed recursively, each coordinator creating its child processors, as shown in the constructor pseudo-code in Code 83.

```
Coordinator(CoupledModel component, ParentProcessor parent)
{
    self.parent = parent;

    for each <childComponent in component>
    {
        ChildName = childComponent.ComponentName;
        ChildProcessor = ProcessorsMap.GetValue<ChildName>

        processors.insert
            <ChildName, ChildProcessor>
            (new ChildProcessor(childComponent, self));
    }
}
```

Code 83. Coordinator – Constructor (pseudo-code).

This constructor allows us to introduce an interesting metaprogramming construct, which will come back often in the rest of this section. In this code, some runtime computations are nested into a compile-time statement. When describing the `Simulator` class, we saw that it is possible to conditionally include or exclude things from the object program depending on static data. Here, the metaprogram will go even further and will loop over static data to generate a succession of dynamic statements. In practice, this means that the iteration over the children of the coupled component will be entirely performed at compile-time, effectively producing a program with no trace of a loop. Since the Fusion map also works mainly at compile-time, the code generated for the constructor will end up being a mere succession of assignments to data members. For instance, assuming a coupled model with one coupled component C1 and two atomic components A1 and A2, the resulting program would be similar to the one obtained by compiling Code 84.

```
Coordinator(CoupledModel component ParentProcessor parent)
{
    self.parent = parent;

    self.coordC1 = new Coordinator(component.C1);
    self.simA1   = new Simulator(component.A1);
    self.simA2   = new Simulator(component.A2);
}
```

Code 84. Coordinator – Unrolled constructor for a sample coupled model (pseudo-code).

Such loop unrolling makes for increased performance of the resulting executable, not only by suppressing the loop overhead but also by giving more optimization opportunities to the compiler.

We will now present the simulation algorithms implemented by the Coordinator class

III.4.2.1. Initialization

The initialization algorithm, summarized in Code 85, is one of the rare parts of the coordinator where most computations happen at runtime.

```
void init(time t)
{
    tl = -∞;
    tn = ∞;

    for each <child in processors>
    {
        child.init(t);
        tl = max(tl, child.tl);
        tn = min(tn, child.tn);
    }
}
```

Code 85. Coordinator – Initialization algorithm (pseudo-code).

The only notable thing about this algorithm is the unrolling of the loop iterating over the child processors. Otherwise, it is rather dull: the coordinator initializes in turn each child processor, keeping track of the maximum time of last event and minimum time of next event along the way.

III.4.2.2. Internal transition

The algorithm for handling internal transitions is indubitably more interesting: as highlighted in Code 86, a great deal of computations is actually performed at compile-time.

This function is an intertwining of static and dynamic computations. First of all, the coordinator needs to construct the list of imminent components. To do so, it iterates at compile-time over the child processors, queries their time of next event at runtime, and accordingly updates the list of imminent components *at compile-time*. This round-trip between runtime and compile-time may seem a bit strange: how can dynamic computations impact static data? To clarify what is going on here, we need to scrutinize this part of the algorithm.

```
void proceed(time t)
{
    assert(t == tn);

    Imminents = empty sequence;
    for each <child in processors>
    {
        if (child.tn == t)
        {
            Imminents.Add<child.ComponentName>;
        }
    }

    if <Imminents.Size == 0>
    {
        assert(false); // no imminent components!
    }
    else if <Imminents.Size == 1>
    {
        ActivableComponentName = Imminents[0];
    }
    else
    {
        for each <Rule in Select>
        {
            if <Imminents.Include<Rule>>
            {
                ActivableComponentName = Rule[0];
            }
        }
    }

    processors.GetValue<ActivableComponentName>().proceed(t);

    tl = t;
    tn = ∞;

    for each <child in processors>
    {
        tn = min(tn, child.tn);
    }
}
```

Code 86. Coordinator – Internal transition algorithm (pseudo-code).

The first thing to note is that it is in fact not possible to modify the `Imminents` collection, because it is a static element. Remember that in C++ TMP, all data is immutable. The only way to iteratively construct such collection is to rely once again on recursion, using template parameters to carry the appropriate data. Code 87 provides the actual algorithm used in DEVS-MS to construct the list of imminent components at compile-time.

```
void proceed(time t)
{
    assert(t == tn);

    recursiveFilter<mpl::sequence<>, processors.Begin>(processors.begin());
}

// recursive case: it != end
void recursiveFilter
    <Imminents, ProcessorIterator>
    (ProcessorIterator it)
{
    if <ProcessorIterator == processors.End> // base case: it == end
    {
        proceedRemainder<Imminents>();
    }
    else                                     // recursive case: it != end
    {
        if (it->processor.tn == tn)
        {
            recursiveFilter
                <Imminents.Add<ProcessorIterator->Name>, ++ProcessorIterator>
                (++it);
        }
        else
        {
            recursiveFilter
                <Imminents, ++ProcessorIterator>
                (++it);
        }
    }
}

void proceedRemainder<Imminents>()
{
    if <Imminents.Size == 1>
    [...] // as before
}
```

Code 87. Coordinator – Recursive algorithm to construct a static list of imminent components depending on dynamic computations (pseudo-code).

The main idea of this algorithm is to use a recursive filter to rule out the components whose activation time has not come yet. The filter takes two parameters: a static list of imminent components, and an iterator into the map of child processors (which contains both static and dynamic information). If there is no child processor to process (i.e. if the iterator points to the end of the map), the filtering is over and the list of imminent components is complete. Since the list cannot be “returned” in a classical way, the filter calls a function that is

```
if (proc_c1.tn == tn)
{
    if (proc_c2.tn == tn)
    {
        if (proc_c3.tn == tn) proceedRemainder<{c1, c2, c3}>();
        else                  proceedRemainder<{c1, c2}>();
    }
    else
    {
        if (proc_c3.tn == tn) proceedRemainder<{c1, c3}>();
        else                  proceedRemainder<{c1}>();
    }
}
else
{
    if (proc_c2.tn == tn)
    {
        if (proc_c3.tn == tn) proceedRemainder<{c2, c3}>();
        else                  proceedRemainder<{c2}>();
    }
    else
    {
        if (proc_c3.tn == tn) proceedRemainder<{c3}>();
        else                  proceedRemainder<{}>();
    }
}
```

Code 88. Coordinator – Combinations generated by the metaprogram and selected by the object program (pseudo-code).

responsible for handling the rest of the internal transition, passing it the static list of imminent components. If there are still processors to process, the filter queries the current processor for its time of next event. If this time is equal to the current simulation time, the filter calls itself recursively with a list updated with the imminent component name and an iterator to the next processor; otherwise, it calls itself with an unchanged list and an iterator to the next processor. Given this definition, the coordinator simply needs to invoke the recursive filter with an empty compile-time sequence and an iterator to the first element in the map of child processors.

If we think about what this algorithm actually does, we will notice that there is in fact no impact from the runtime to the compile-time (if that was the case, we would have an embarrassing temporal causality loop!): the metaprogram simply generates all possible cases, and the object program merely selects the appropriate one. For instance, assuming a coupled model with three components “c1”, “c2” and “c3”, the code generated by the recursive filter would boil down to something similar to Code 88.

Once the list of imminent components has been constructed, the coordinator needs to determine which of them will be activated. There should always be at least one imminent component, so if the list is empty, we raise a runtime assertion to indicate that the library contains a bug. If there is only one component in the list, then this component is selected to be activated. Otherwise, the select function is interrogated to break the tie. Since the select function and the list of imminent components are both known at compile-time, the choice of the component to activate is entirely performed statically. The coordinator can finally trigger an internal transition on the appropriate coordinator, and updates the times of last and next event.

III.4.2.3. Input message and output message

In addition to triggering internal transitions in their children, coordinators are responsible for routing events throughout the model. Usually, coordinators use two distinct algorithms to handle input messages, corresponding to an event received on an input port of the coupled model, and output messages, corresponding to an event generated by a component of the coupled model. In DEVS-MS, we use a single function to handle both cases, mainly for the sake of code factorization. The algorithm of this function is outlined in Code 89.

This function has two static parameters:

- The name of the source component for the event. In the case of an input message, the name is that of the coupled component itself; in the case of an output message, it is the identifier of one of its child components.
- The type of message received, which embeds both the type of value carried by the event and the source port. In the case of an input message, the port is an input port of the coupled model; in the case of an output message, it is an output port of one of its components.

Based on the source component and source port, the coordinator statically retrieves the destinations for the event. This is easily done thanks to the map representation of couplings constructed by the coordinator upon initialization. This map contains all couplings, regardless of their nature (EIC, IC or EOC). To handle all these homogeneously, the coordinator creates at compile-time an extended components map, containing not only child

```
// to handle EIC, IC and EOC homogeneously
ExtendedComponentsMap =
    CoupledModel.ComponentsMap.Insert<ComponentName, CoupledModel>;

void forward<SrcComponentName, Message>(Message msg, time t)
{
    if <SrcComponentName == ComponentName> // if x-message
    {
        assert(tl <= t <= tn);
    }

    SubMap = CouplingsMap.GetValue<SrcComponentName>;
    SrcPortName = Message.Ports[0].Name; // CDEVS, only one value on one port

    Destinations = SubMap.GetValue<SrcPortName>;

    // to handle EIC, IC and EOC homogeneously
    extendedProcessors = processors.insert<ComponentName, ParentProcessor>(parent);

    for each <processor in extendedProcessors>
    {
        if <Destinations.HasKey<processor.ComponentName>>
        {
            DestComponentName = processor.ComponentName;
            DestPortName = Destinations.GetValue<DestComponentName>;
            DestModel = ExtendedComponentsMap.GetValue<DestComponentName>;

            if <DestComponentName == ComponentName> // if EOC
            {
                DestMessage = DestModel.MessageOnOutputPort<DestPortName>;
            }
            else // EIC or IC
            {
                DestMessage = DestModel.MessageOnInputPort<DestPortName>;
            }

            // convert source message to destination message
            DestMessage destMsg = new DestMessage(msg.getValueOnPort<SrcPortName>());
            processor.forward<DestComponentName, DestMessage>(destMsg);
        }
    }

    if <SrcComponentName == ComponentName> // if x-message
    {
        tl = t;
        tn = ∞;

        for each <child in processors>
        {
            tn = min(tn, child.tn);
        }
    }
}
```

Code 89. Coordinator – Algorithm for handling input and output messages (pseudo-code).

components but also the coupled component itself. Similarly, it creates at runtime an extended processors map, where its component is associated with its parent processor. This way, all events routed on output ports of the coupled component will be sent to the parent processor.

With these little adjustments, the coordinator can statically iterate over the extended processors map, and test for each one if its component appears in the list of destinations. When this is the case, the coordinator retrieves the model associated with the component (from the extended components map), and uses it to create the new message type. If the destination component is the coupled component itself, the new message is created based on its output set; otherwise, it is created based on the input set of the destination child. Note that the new message will be entirely different from the original one: not only will the message be modified to include the destination port instead of the source port, but the type of value itself can be different. Indeed, two connected ports can have different types, as long as they are compatible. By interrogating the input and output sets, the coordinator makes sure that the value is correctly converted, if need be, before being sent to the destination.

Finally, the `forward` function updates the times of last and next event if need be, i.e. if dealing with an input message.

The important thing to note is that couplings are actually resolved at compile-time. At runtime, the coordinator simply creates the new message and sends it to the destination processor, which can be a simulator, a child coordinator, or the parent coordinator. In fact, in the resulting executable, everything happens as if the coordinator had a unique function for each potential event source, which forwarded events directly to the appropriate processors.

In the end, DEVS-MS does not explicitly remove coordinators and simulators from the simulation. However, the metaprogram reduces them to the bare minimum: they contain no dynamic data structures, no loops, no virtual function calls, no dynamic indirections whatsoever, etc. All that is left are classes with almost no overhead; the only potential overhead lies in the function calls between the processors of the hierarchy. However, the function bodies have been extremely simplified by the metaprogram, to a point where they merely consists in plain method invocations and simple arithmetic computations. Thanks to

this, the optimizer is very likely to inline these functions if it predicts that this will improve performances, effectively removing the processors from the resulting executable.

We have no real control over inlining, the optimizer is the sole decision-maker in this regard. We did some experiments to know if it was actually performed, compiling simple calls and analyzing the generated assembly code, and these tests showed that most coordinator functions were indeed inlined. However, there is no guarantee that this will always be the case. Anyway, the optimizer is usually better than developers at deciding whether an optimization will be beneficial or not. In the end, DEVS-MS performs a sort of high-level optimization by computing many simulation operations at compile-time, but leaves the compiler deal with lower-level optimizations.

III.4.3. Root

Finally, DEVS-MS provides a `Root` class for handling the main simulation loop. This is the only processor that is manipulated directly by the practitioner, and it is very simple, as shown in Code 90.

```
class Root<ComponentName, Model>
{
    SelfType = Root<ComponentName, Model>;
    TopProcessor = GetProcessorFor<ComponentName, Model, SelfType>;

    TopProcessor child;

    Root(Model component);

    void run(time initialTime, time endTime);
    void forward<SrcComponentName, Message>(Message msg, time t);
}
```

Code 90. Root class (pseudo-code).

The root coordinator is parameterized by a component name and a model, which can be either atomic or coupled. From these, it statically determines the appropriate processor, either a simulator or a coordinator. To do so, it uses the same metafunction used by coordinators to determine the types of their child processors. At runtime, the top processor is constructed with the model instance given to the `Root` constructor.

The main simulation loop is handled by the `run` member function, outlined in Code 91. First, the top processor is initialized with the initial simulation time provided by the caller. Then, it is repeatedly activated until the current simulation time reach the end time specified by the caller.

```
void run(time initialTime, time endTime)
{
    child.init(initialTime);
    time t = child.tn;

    while (t < endTime)
    {
        child.proceed(t);
        t = child.tn;
    }
}
```

Code 91. Root – Main simulation loop (pseudo-code).

The `Root` class also provides a function for catching “leaking” events, meaning events that are sent to output ports of the top component and consequently received by no one. This function simply logs the events caught, along with its source and time of occurrence.

```
void forward<SrcComponentName, Message>(Message msg, time t)
{
    SrcPortName = Message.Ports[0].Name;

    string srcComponentName = typeToName<SrcComponentName>::value;
    string srcPortName      = typeToName<SrcPortName>::value;

    log(t, srcComponentName, srcPortName, msg.getValueOnPort<SrcPortName>());
}
```

Code 92. Root – Default handling of "leaking" events (pseudo-code).

Now that we have exposed the implementation of DEVS-MS, we will present the sample model we used to test it, along with the results obtained.

IV. Sample application: semantic pervasive dual cache

To benchmark DEVS-MS, we implemented a model from a previous work [D’Orazio and Traoré 2009]. This model was used to evaluate a semantic cache for pervasive grids. Before describing the model, we will quickly present the system it represents.

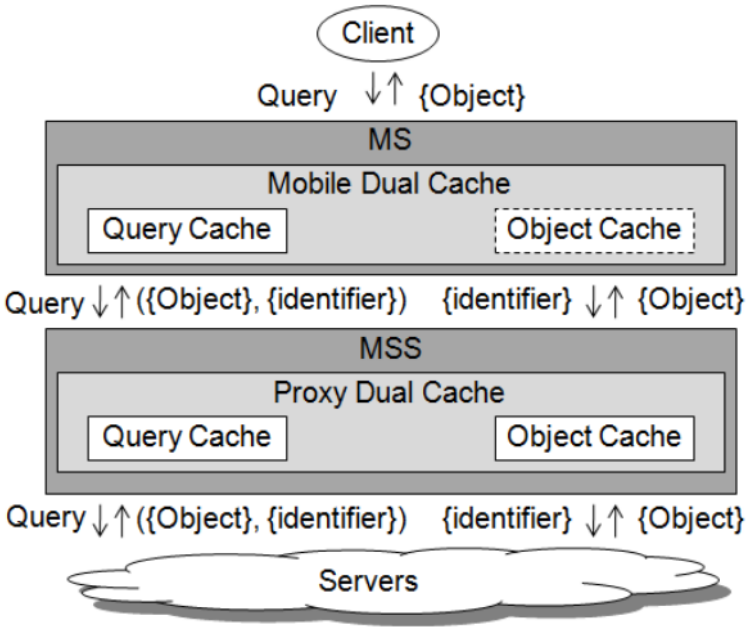


Figure 46. Architecture of a pervasive dual cache [D'Orazio and Traoré 2009].

The idea behind pervasive computing [Poslad 2009], also known as ubiquitous computing, is to transition from the classical desktop paradigm, where interacting with computers implies an explicit effort from humans, to a world where the machines become ubiquitous entities that we use without friction, possibly even without being aware that we do so. This vision usually involves having numerous smart devices distributed in our environment, connected through a network, and providing services to the users around them.

In this context, the pervasive grid aims at providing a link between ubiquitous computing and grid computing [Foster and Kesselman, 1998], which consists in distributing data and computations over a network of heterogeneous devices, often distributed over the world. Thus, the goal of the pervasive grid is to integrate pervasive devices into such worldwide systems, in order to make the information captured by devices available to the grid, and the resources of the grid available to devices.

To be scalable and efficient, a pervasive grid must limit the number of queries sent to servers. A natural solution to this issue is to use caches to avoid a round-trip to the servers when a query can be answered from the results of previously executed ones. Figure 46 [D'Orazio and Traoré 2009] shows the architecture of a pervasive dual cache, where two layers of caching are applied between clients and servers. The first cache (mobile dual cache)

is hosted in the device used by the client (Mobile Station (MS)) and stores queries and results. The second cache (proxy dual cache) is hosted in the Mobile Support Station (MSS), a more powerful device that serves as a gateway to the servers.

Using a mobile cache with semantic capabilities, several scenarios are possible when a client submits a query:

- The query can be answered from the content of the mobile cache (exact hit). In this case, the MS directly provides the results to the client.
- The query can be partially answered from the mobile cache (partial hit). In this case, the query is divided into a probe query, which is used locally to obtain the results available, and a remainder query, which is sent to the MSS for further processing.
- The query is completely unrelated to queries saved in the mobile cache (cache miss). The complete query is forwarded to the MSS.

The same alternatives apply to the proxy cache: it can send the results back to the MS directly, or send a remainder query or a complete query to the servers.

DEVS models have been developed to compare this caching strategy with others. We will now present the model we used to test DEVS-MS, which represents the semantic pervasive dual cache just described.

IV.1. Model of a semantic pervasive dual cache system

Since its original purpose was to compare the mean response time provided by various caching strategy, the model we developed does not take into account servers. Indeed, the latter have the same impact on the response time regardless of the caches used. Thus, the model represents only the mobile and proxy caches.

One of our goals with this model is to test the impact of using DEVS-MS on performances. Consequently, we decomposed it in very small components so that the simulation overhead takes an important part of the total execution time.

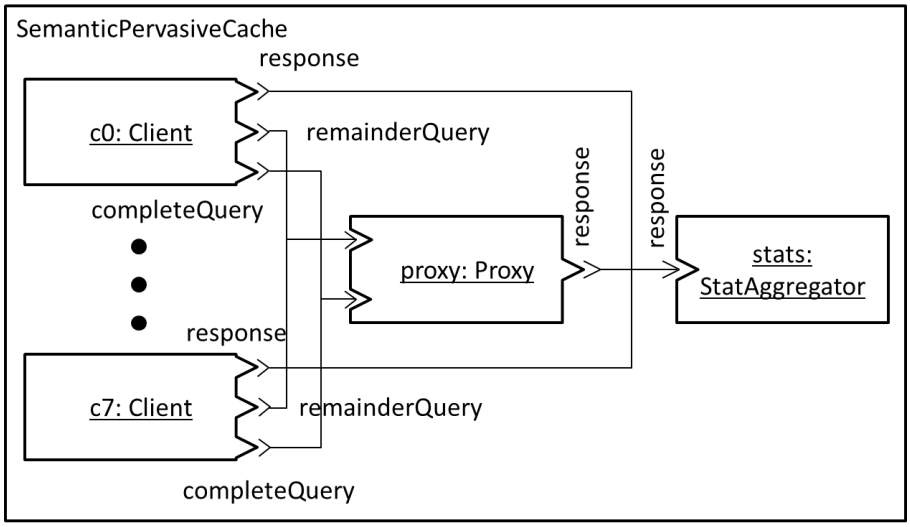


Figure 47. Semantic pervasive dual cache model.

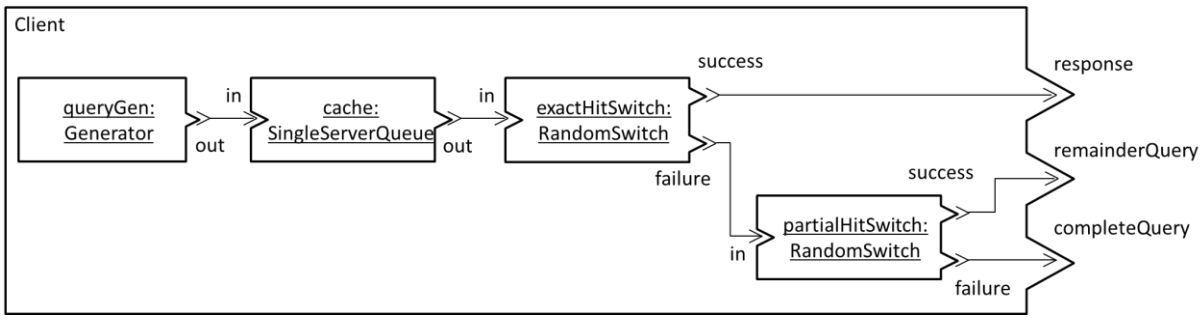


Figure 48. Client model.

Figure 47 depicts the overall coupled model of the semantic pervasive cache. It is composed of eight clients (MS) that are connected to a single proxy (MSS). All responses provided by caches are forwarded to a statistic aggregator that is responsible for computing the mean response time.

As shown in Figure 48, each client is composed of the following elements:

- A query generator, which randomly generates queries following an exponential distribution.
- A single server queue, which processes requests according to exponentially distributed service times. (Together with the generator, it forms an M/M/1 queue [Kendall 1953].)
- A random switch modeling the probability of exact hits. If the query is an exact hit, an event is sent to the “response” port of the client.

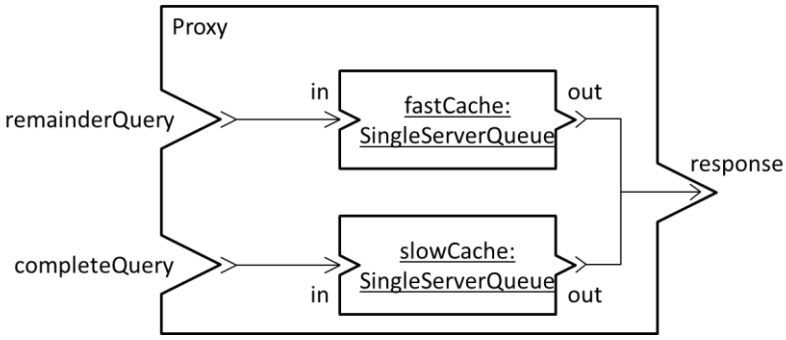


Figure 49. Proxy model.

- A random switch modeling the probability of partial hits. If the query is a partial hit, an event is sent to the “remainderQuery” port of the client. Otherwise, an event is sent to the “completeQuery” port.

Retrospectively, a better design would probably have been to decouple the query generator from the elements representing the mobile cache, instead of embedding them all in a single coupled model.

Finally, as depicted in Figure 49, the semantic proxy cache model simply contains two single server queues for representing the processing of remainder and complete queries. When a query has been preprocessed by the mobile cache, the proxy cache is able to treat it more rapidly. In either case, the response is sent to the output port of the coupled model.

To sum up, the semantic pervasive dual cache model presented here is composed of 35 atomic models, 9 intermediate coupled models, and 1 top level coupled model.

IV.2. Implementation in DEVS-MS

Giving the full implementation of this model in DEVS-MS would be rather tedious, but it is interesting to see some samples to understand how DEVS-MS is used from a practitioner perspective. To keep it short, we will just show the code of one atomic model and one coupled model, namely the RandomSwitch and the Proxy models.

IV.2.1. RandomSwitch model in DEVS-MS

Code 93 provides a DEVS-MS implementation of a random switch, a simple atomic model with one input port and two output ports, which dispatches events according to some probability given as parameter.

The code starts with the declaration of the port identifiers used by the switch. Those are included within a namespace specific to the switch model, to prevent name clashes. Then, the actual class specifying the model is defined. It inherits from `AtomicModel`, providing as template arguments the input and output sets.

There is not much to say about the constructor: it simply takes the success probability as a parameter, and initializes the state of the switch to the appropriate values. The state is stored in data members, listed at the end of the class definition. It is composed of a variable indicating whether the switch is idle (waiting) or dispatching an event (outputting), the event being dispatched, a boolean determining on which port the event will be sent, and a pseudo-random number generator. The model also stores its parameter for use in transition functions, as a constant data member.

The time advance function is rather straight-forward as well, and returns either “infinity” or “0” depending on the current state of the switch. Similarly, the internal transition function is very simple and merely switches the model back to the “waiting” state.

The output function illustrates the callback mechanism we evoked previously. Depending on the port where the event must be sent, the model creates a message of the appropriate type with the `createMessageOnOutputPort` function provided by `AtomicModel`, and sends it to the simulator passed as parameter.

The external transition function is parameterized so that it accepts only messages on the In port, along with a value of the appropriate type (in this case, an integer). To do so, it uses the `MessageOnInputPort` metafunction defined in `AtomicModel`. If the model had more input ports, it could define several overloads for each port, the appropriate one being selected at compile-time.

```
DECLARE_PORT((semCache) (randomSwitch), In)
DECLARE_PORT((semCache) (randomSwitch), Success)
DECLARE_PORT((semCache) (randomSwitch), Failure)

namespace semCache {

struct RandomSwitch
  : devs::classic::AtomicModel<
        PORTS( ((randomSwitch::In, int)) ),
        PORTS( ((randomSwitch::Success, int)) ((randomSwitch::Failure, int)) )>
{
    explicit RandomSwitch(double successProbability)
        : successProbability(successProbability),
          randomGenerator_(),
          state_(waiting)
    {}

    double ta() const
    {
        if (state_ == waiting)
            return std::numeric_limits<double>::infinity();

        return 0;
    }

    void deltaInt()
    {
        state_ = waiting;
    }

    template <typename Simulator>
    void lambda(Simulator & s)
    {
        if (success_)
            s.forward(createMessageOnOutputPort<randomSwitch::Success>(current_));
        else
            s.forward(createMessageOnOutputPort<randomSwitch::Failure>(current_));
    }

    void deltaExt(const result_of::InputMessageOnPort<randomSwitch::In>::type & msg)
    {
        current_ = msg.getValueOnPort<randomSwitch::In>();
        state_   = outputting;
        success_ = randomGenerator_() < successProbability_;
    }

private:
    const double successProbability_;

    enum {waiting, outputting} state_;

    int    current_;
    bool   success_;
    rand01 randomGenerator_;
};

} //namespace semCache
```

Code 93. RandomSwitch atomic model in DEVS-MS.

The syntax for defining output function and external transition function is a bit heavy. This is especially the case when the model is defined as a class template. Here, the values handled by the random switch are integers, but in our actual code, we used a class template to make the switch generic and reusable with any input/output set. When doing so, invoking (meta)functions defined in the base class implies using the `typename` and `template` keywords, making the code harder to write and read. This complexity comes mostly from the use of the `Message` class, which provides functionalities that are essential for PDEVS but useless for CDEVS. To simplify the syntax, we recently made some modifications to DEVS-MS to hide this complexity from the user. To do so, we relied on tag dispatching to make the definition of overloads easier for the user, as well as the generation of outputs. In this new version, the value associated with the event and the port information are passed separately instead of being merged into a `Message`. Code 94 shows the definition of the `lambda` and `deltaExt` functions using this new syntax. It is arguably simpler, especially for defining the external transition function.

```
template <typename Simulator>
void lambda(Simulator & s)
{
    if (success_)
        s.forward(current_, randomSwitch::Success());
    else
        s.forward(current_, randomSwitch::Failure());
}

void deltaExt(int value, randomSwitch::In)
{
    current_ = value;
    state_   = outputting;
    success_ = randomGenerator_() < successProbability_;
}
```

Code 94. Definition of the output function and external transition function using the simplified syntax.

The only drawback of this approach is that port types are explicitly stated in the parameter list of external transition functions, while they are already specified in the lists of input and output ports. This redundancy can cause problems if the type is updated in one place but not the other. If the practitioner wants to avoid this duplication, we provide a metafunction in `AtomicModel` to obtain the type associated with a given port.

These modifications only impact the interface of the library, and have no impact on its inner workings.

IV.2.2. Proxy model in DEVS-MS

As we explained previously, a coupled model specification in DEVS-MS is entirely composed of static data. Thereby, coupled model definitions can be entirely done in a declarative way, without any imperative constructs.

Code 95 shows the DEVS-MS specification of the Proxy model. First of all, some identifiers are declared, to denote ports and components of the model. Then, a template alias is used to define the actual coupled model. All models in the semantic pervasive dual cache model are defined as templates, parameterized with the type of values handled. This way, they can be reused with various types, without having to write a new model each time. However, this is just a design choice we made, and by no means a requirement of DEVS-MS, as shown in Code 94.

The coupled model specification maps directly to Figure 49. It includes the list of components, the list of couplings, a specification of the tie-breaking function that states that the slow cache has priority over the fast cache, and the declarations of the input and output ports. To make the Proxy model self-sufficient, we define a class whose sole purpose is to create and store the two components. Alternatively, we could directly use the `CoupledModel` specialization, but we would still need to create the components somewhere. Indeed, `CoupledModel` instances only reference their components (aggregation), they do not contain them (composition). This gives more flexibility to the user, notably when it comes to initialization: components can be created in any conceivable way before being passed to the coupled model. Here, we decided to embed them into a container class.

The reason we used an intermediary template alias (`proxy::base`) instead of directly using `CoupledModel` in the base-list is to avoid repeating the entire specification in the initialization list of the constructor.

```
DECLARE_PORT((semCache) (proxy), RemainderQuery)
DECLARE_PORT((semCache) (proxy), CompleteQuery)
DECLARE_PORT((semCache) (proxy), Response)

DECLARE_COMPONENT((semCache) (proxy), FastCache)
DECLARE_COMPONENT((semCache) (proxy), SlowCache)

namespace semCache {

namespace proxy {
    template <typename T>
    using base = devs::classic::CoupledModel<
        COMPONENTS(
            ((FastCache, SingleServerQueue<T>))
            ((SlowCache, SingleServerQueue<T>))
        ),
        COUPLINGS(
            (( (devs::This, RemainderQuery),   (FastCache,  singleServerQueue::In) ))
            (( (devs::This, CompleteQuery),    (SlowCache,  singleServerQueue::In) ))
            (( (FastCache,  singleServerQueue::Out),   (devs::This, Response) ))
            (( (SlowCache,  singleServerQueue::Out),   (devs::This, Response) ))
        ),
        SELECT(
            ( (SlowCache) )
            ( (FastCache) )
        ),
        PORTS(
            ((RemainderQuery, T))
            ((CompleteQuery, T))
        ),
        PORTS(
            ((Response, T))
        )
    >;
} //namespace proxy

template <typename T>
struct Proxy
  : proxy::base<T>
{
    Proxy()
      : proxy::base<T>(fastCache_, slowCache_),
        fastCache_(60),
        slowCache_(20)
    {}

private:
    SingleServerQueue<T> fastCache_;
    SingleServerQueue<T> slowCache_;
};

} //namespace semCache
```

Code 95. Proxy coupled model in DEVS-MS.

We will now present the results we obtained from testing DEVS-MS with our semantic pervasive dual cache model.

IV.3. Results

IV.3.1. Performance

To test the impact of DEVS-MS on performance, we implemented the model in two other DEVS simulators that use the same simulation algorithms. One was a Java library that was already in use in our team, and the other was a C++ library written specifically for the test. The latter is an exact transposition of DEVS-MS where all compile-time computations have been moved back to runtime. This way, the difference in performance between these two could only be attributed to the use of metaprogramming or not.

Additionally, to observe the overhead of using a generic simulation library instead of specialized software, we developed three different C++ programs whose sole purpose is to simulate this particular model. Each of these was implemented following the ideas presented in Chapter 6.II (flattening, direct connection, removing of all simulation processors, suppression of useless simulation operations, etc.). The only difference between these three implementations lied in the way events were scheduled.

In the first implementation (key + queue), a numerical key is assigned to each component. These keys are used in a queue sorted according to the time of next event of components. The second implementation (base class + queue) is rather similar except that instead of using keys to denote components, they all derive from a common base class and the queue stores pointers to components. In the last implementation (base class + scan), each component once again derive from a common base class, but is also responsible for computing its time of next event; then, at each simulation step, we scan all the components to find the one with the smallest time of next event. In all three implementations, tie-breaking between simultaneous internal events is performed implicitly, either by appropriately ordering the identifiers of the components, or by scanning them in a particular order. This is only possible because the select functions used in the cache model are very simple, i.e. they merely prioritize components. Had they been more complex, the three custom implementations would have needed to go through some convolutions to handle them correctly.

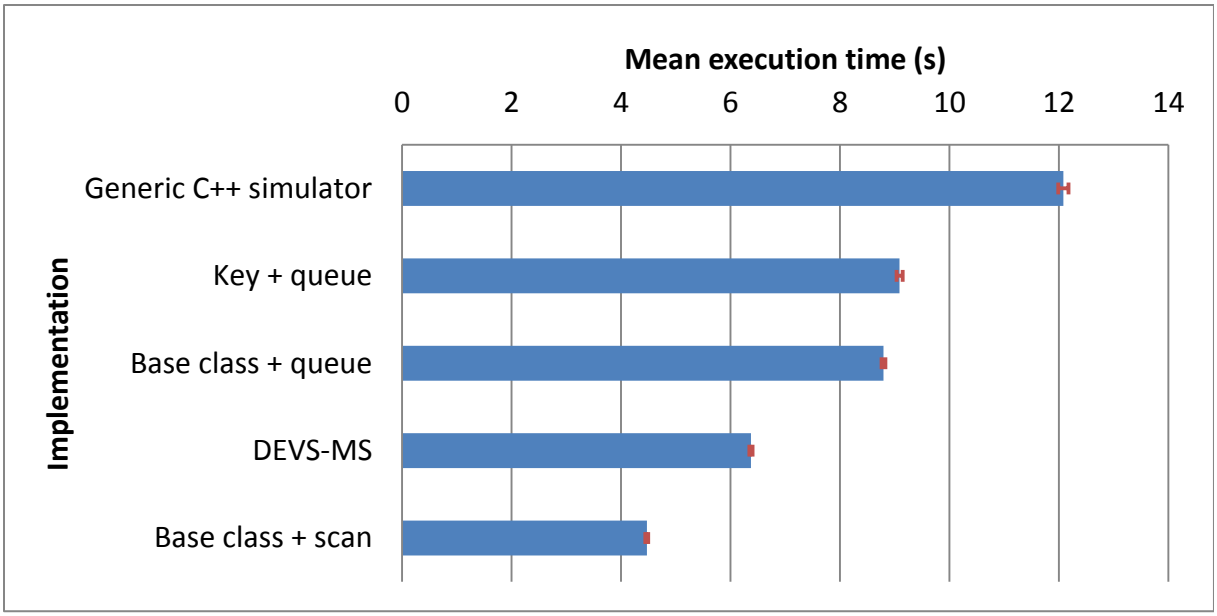


Figure 50. Mean execution time for various implementations of the cache model.

IV.3.1.1. Execution times

We measured the execution times of each implementation over a simulation run of 100 000 time units, representing the equivalent of approximately 21 millions activation messages, 11 millions input or output messages, 7 millions internal events and as much external events, and 12 thousands tie-breaks. All implementations gave the same simulation results. They were all compiled with GNU C++ 4.7 with the optimization level O3 enabled, and executed in the same environment, namely a Linux distribution (Ubuntu 10.10) running on a virtual machine (VirtualBox 4.1.10). To take into account the variability of the runtime context, we performed 40 replications of each simulation. The mean execution times obtained are given in Figure 50 and Table 3, along with the margin of error (95% confidence interval). We also provide the speedup of DEVS-MS over the other implementations ($\frac{T_{other}}{T_{DEVS-MS}}$), as well as the overhead of the various implementations ($\frac{T_{other}-T_{best}}{T_{best}}$)[43]. We voluntarily left aside the Java implementation due to the difference of programming language: slower execution time could be imputed to Java rather than to the simulation kernel. For the record, the mean execution time of this implementation was about 29.27 seconds.

[43] To compute the true overhead, we would need the theoretical time spent in transition functions. Since we do not have this time, we took the execution time of our fastest implementation, which should be pretty close.

	Generic C++ simulator	Key + queue	Base class + queue	DEVS-MS	Base class + scan
Mean execution time (s)	12.08 ± 0.06	9.09 ± 0.04	8.80 ± 0.03	6.38 ± 0.03	4.47 ± 0.02
Speedup	1.89	1.43	1.38	1.00	0.70
Overhead	170.17 %	103.27 %	96.73 %	42.60 %	0.00 %

Table 3. Comparison of the execution time of various implementations of the cache model.

These results show that DEVS-MS vastly outperforms the generic C++ simulator, being almost twice as fast. More surprisingly, it also outdoes the two implementations that use an event queue, even though they were manually crafted and optimized for the model at hand. This is arguably due to the event queue: with the sample model at hand, maintaining an ordered set of events is probably more time consuming than querying components at each iteration. Incidentally, the last implementation, which uses this scanning approach, is the fastest, in front of DEVS-MS. It is quite possible that simulating a model with different characteristics would show the opposite, for instance if there was a great number of components per coupled model.

The last implementation is supposed to be very close to the code we would like DEVS-MS to generate. We see two potential reasons for explaining why DEVS-MS lags a bit behind. The first one is that we optimized the manually crafted software using model characteristics that were not available to the simulator. For instance, all computations related to the elapsed time, including keeping track of the time of last event, were removed. Indeed, it was never used, not even by the single server queues (they directly store their time of next time event). Similarly, computations related to the determination of the time of next event were greatly simplified in the specialized implementation, which was not possible to do in a generic library like DEVS-MS.

Another thing that could explain the difference between DEVS-MS and the custom implementation is the numerous function calls performed by the former. Indeed, the use of TMP forced us to use many intermediate functions, often with long chains of delegation.

These functions are always very simple, making them very amenable to inlining, but it is possible that the optimizer decided to not inline some of them for some reason, incurring a small additional cost compared to a manually written implementation.

When considering the results presented here, one must keep in mind that for a given model, the simulation overhead is not function of the overall execution time, but rather of the number of simulation steps. Indeed, the important differences observed here are explained by the fact that the transition functions used in the model are very short. If we increased the amount of computations in these functions, the absolute overhead would stay the same, and thus the relative overhead would decrease. We can draw an analogy between simulation overhead and time "lost" at the airport: whether the flight taken lasts two hours or twelve hours, the amount of time spent at customs and at the boarding gate will stay the same. Consequently, all relative data provided should be used only as a baseline for comparing alternatives, not as an indication of the intrinsic value of a solution. For instance, comparing the overhead of DEVS-MS to the one of the generic simulator shows that DEVSMS overhead is almost four times smaller; this ratio should remain constant even when increasing the complexity of transition functions or the number of simulation steps.

Another thing to bear in mind is that the characteristics of the model also play an important part. For instance, a model with highly interconnected components and a high depth would favor DEVS-MS over the classical C++ simulator, because the additional computations incurred would be performed at compile-time by DEVS-MS. Similarly, complex tie-breaking functions would reduce the lead of the manually crafted implementation, while a high depth would probably be to its advantage. To test the impact of these various criteria, we plan to use a rigorous testbed, using a model generator such as DEVStone [Glinsky and Wainer 2005] to automatically create models with various characteristics. Unfortunately, such benchmark can only be used to test DEVS simulation libraries, not to compare them with manually written software (unless willing to spend days developing dozens or hundreds of useless programs). Nevertheless, it will allow us to compare the classical approach and the metaprogramming approach with more detail.

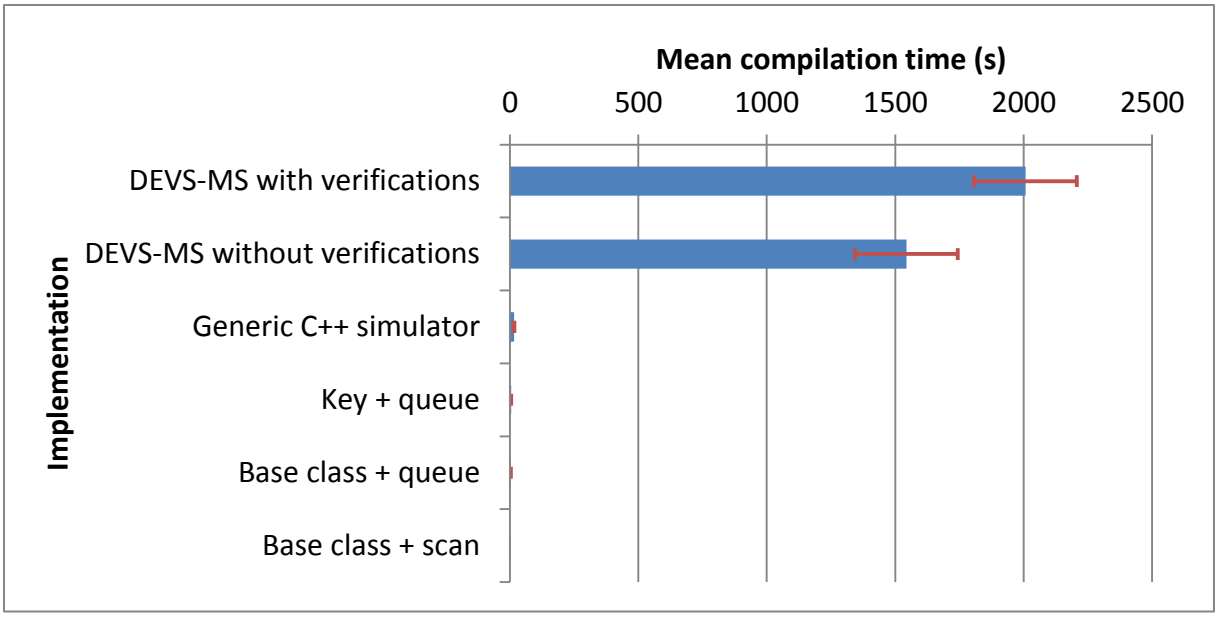


Figure 51. Mean compilation time of various implementations of the cache model.

	DEVS-MS with verifications	DEVS-MS without verifications	Generic C++ simulator	Key + queue	Base class + queue	Base class + scan
Mean compilation time (s)	2000 ± 200 (~33 min)	1500 ± 200 (~ 25 min)	16.1 ± 0.9	5.02 ± 0.04	3.18 ± 0.04	2.93 ± 0.04

Table 4. Mean compilation time of various implementations of the cache model.

IV.3.1.1. Compilation time

Moving computations from runtime to compile-time comes with a cost: compilation times drastically increase. Figure 51 and Table 4 give the mean compilation times for each implementation of the semantic pervasive dual cache model. We compiled DEVS-MS in two different ways: one where all compile-time verifications (static assertions) are performed, and one where they are disabled. Regarding the generic C++ simulator version, the library was compiled together with the model, not during a preliminary step.

All compilations were performed 5 times, to account for variability of the environment. Results are provided with the confidence interval at 95%.

The huge compilation times observed with DEVS-MS obviously come from the heavy use of template metaprogramming. Indeed, actual compilers are not really good at handling such numerous and deep template instantiations, resulting in very long compilation times and huge memory consumption. In fact, Microsoft Visual C++ compiler even failed to compile the application due to memory exhaustion. However, this situation should quickly improve as metaprogramming becomes ubiquitous in modern C++ libraries. Compiler vendors begin to implement alternative strategies for handling template instantiations, providing much better results.

In this regard, the recent compiler Clang[44] promises very important improvements, notably in terms of compilation times[45]. It is still in its early days: the last time we tried it, it supported only a small subset of C++, thereby failing to compile many Boost libraries and by extension DEVS-MS. However, it has greatly improved these last months, so we plan to make a new attempt in a close future. For the time being, the use of DEVS-MS is unfortunately limited to medium-sized models, until compilers integrate better support for C++ TMP.

IV.3.2. Model verification

In addition to the performance gain just described, DEVS-MS provides another very important improvement over classical simulation libraries, relating to model verification.

Indeed, by processing some of the model specification at compile-time, we are able to detect modeling errors very early in the development cycle, even before launching the simulation. Practically, this means that the library checks many requirements over the model during compilation, and that failures to meet these requirements are reported to the practitioner in the form of compiler errors.

The most important improvement is arguably static type checking. By typing ports and keeping track of the types of values throughout the simulator, we effectively manage to statically assert the compatibility of all output and input sets, meaning that it is impossible to connect two ports whose associated sets are not compatible without triggering a compiler error. The compatibility checks rely on the C++ type system, thereby providing support for rich features such as user-defined conversions, subtyping, etc.

[44] http://clang.llvm.org/ (last accessed 17/08/2012)
[45] http://lists.cs.uiuc.edu/pipermail/cfe-dev/2009-May/005267.html (last accessed 17/08/2012)

Many model verifications are also performed on the specification of couplings. DEVS-MS is able to statically assert that a model does not contain any direct feedback loop, i.e. a connection from a component to itself. Connecting a port to two input ports of the same component is also forbidden, as that would imply a simultaneous reception of two events by a single component, which is disallowed in CDEVS.

For every coupling, we also check the validity of the source and of the destination. To do so, we verify that each pair component/port is made of a component that is either the coupled model itself or one of its child components, and of a port that actually belongs to the component it is associated with. This check fits in a larger verification that asserts that all identifiers used throughout the model actually denote an entity: it is not possible to use a name that refers to nothing.

Regarding tie-breaking functions, the format we require for specifying them implicitly forces the selected component to be included in the set of imminent components: it is not possible to create a select function that would return a non-imminent component. However, with our approach, a practitioner could define selection rules with components that do not belong to the coupled model, so we added a static check to avoid such mistake.

Finally, by enforcing the use of `const` on the time advance function and the output function of atomic models, we make sure that the state of the model will not be accidentally modified by these functions. A practitioner can circumvent this restriction using `const_cast` or mutable data members, but he is protected against involuntary mistakes.

V. Conclusion

In this chapter, we presented DEVS-MS, a DEVS simulator that specializes itself for the model it is given. DEVS-MS can be seen as a simulator template from which an endless number of model-specific simulators can be created.

To achieve this, DEVS-MS uses a metaprogramming technique called C++ Template MetaProgramming (TMP), an approach that leverages the template mechanism to perform computations at compile-time and generate robust and efficient code. Basically, the DEVSMS library is parameterized with a model at compile-time, and generates an executable that is

entirely dedicated to simulating this model, resembling the program a developer would write.

Such specialization is achieved by using a classical simulation algorithm where most of the computations have been moved from runtime to compile-time. For example, iterating over components, routing events according to couplings and evaluating tie-breaking functions are simulation operations that are performed by the compiler itself, during template instantiation.

The result of this partial evaluation at compile-time is an executable that contains exclusively computations that are related to the behavior of atomic models and to the handling of simulation time. In this specialized simulator, components communicate directly, sending events to each other as if they were simply invoking methods. The sole responsibility of the simulation entities is the handling of time and the determination of the component to activate at each simulation step.

This specialization provides several advantages. Since many computations are performed at compile-time, the resulting simulation software executes faster than one using a classical simulation library. To verify this claim, we conducted some experiments using a sample model adapted from a previous work. The results of these tests showed that DEVS-MS had four times less overhead than a similar C++ simulator operating only at runtime. However, one of the applications we crafted to specifically simulate this particular model managed to outperform DEVS-MS, showing that a human developer could write more efficient software than that generated by our library. That being said, DEVS-MS has the advantage of being a generic simulation library capable of simulating any DEVS model: implementing our sample model in DEVS-MS was a matter of minutes, while developing and debugging the specific software for the model took several days.

The other important improvement of DEVS-MS over classical DEVS simulation libraries is static model verification. To be conformant with the DEVS formalism, a model must fulfill some requirements. Failing to meet these requirements may lead to bugs that are very hard to detect, so modeling errors should be caught as early as possible in the development cycle.

Since DEVS-MS manipulates model specifications at compile-time, it can perform several verifications before execution even begins. When it detects a design error in a model, it generates a comprehensive error message and stops compilation, allowing the practitioner to correct its mistake. DEVS-MS includes numerous verifications, but the most interesting one is probably the type checking of ports and events. By leveraging the C++ type system, our library is capable of statically asserting that all values received by a model will always be elements of its input set. In other words, it can detect couplings that connect ports with incompatible sets. The notion of compatibility enforced by DEVS-MS goes beyond set equality: two sets are also compatible if the source one is a subset of the destination one (subtype polymorphism), or if a conversion function from the source to the destination is defined.

Regrettably, the metaprogramming approach used in DEVS-MS is not without drawbacks. First of all, it requires the model structure to be known statically, meaning that it cannot be created or modified during execution (e.g. by reading a file, or by interacting with the user). This limitation makes it impossible to apply this approach to some DEVS variants, notably DS-DEVS, where the structure of the model varies throughout the simulation. Another negative aspect is the complexity of the implementation. Indeed, DEVS-MS uses some very advanced C++ features, making it harder to maintain than more straight-forward simulation libraries. However, this complexity is confined on the library side. Finally, an important impediment to the adoption of the library at the time of this writing is the current lack of proper support for C++ TMP by major compilers. Due to the way actual compilers handle template instantiation, using heavy template metaprogramming makes for huge compilation times and memory consumption, restricting the use of DEVS-MS to medium-sized models. However, the ever-increasing use of TMP in modern C++ libraries is forcing compiler vendors to improve this aspect in their compilers. Likewise, new compilers such as Clang use alternative strategies for handling templates to account for this particular use, resulting in much more reasonable compilation times.

The issue of compilation times is our priority regarding the future of DEVS-MS. To improve the situation, we plan to apply some refactoring to the library. First of all, we will make some small modifications of its design to facilitate comprehension and maintenance. Then, we will try to optimize it for compilation time. Indeed, some techniques can be used to reduce the load on the compiler, and we did not do much work in this direction yet.

We also intend to perform more thorough performance tests, using a model generator such as DEVStone [Glinsky and Wainer 2005] to observe the influence of model characteristics on simulation overhead. It could also be interesting to compare DEVS-MS to other simulators using different algorithms, and even to develop metaprogramming variants of these algorithms. Our roadmap also includes developing a Parallel-DEVS version.

We will conclude this chapter by emphasizing the benefits of metaprogramming. This programming technique allows developers to devise libraries that are very generic while still providing efficiency and robustness. For example, the DEVS simulation library we presented in this chapter is as generic as a classical one, i.e. it can simulate any Classic-DEVS model; however, the application it generates is faster and less likely to contain errors, since many of them have been caught during a preliminary stage. This ability to offer the best of both worlds is what makes metaprogramming increasingly used among library writers, whether they are developing a C++ library, a collection of LISP macros, or a Domain-Specific Language.

Chapter 7. General conclusion

I. Discussion

In this thesis, we introduced two complementary tools for DEVS M&S that relies on innovative approaches to provide features that are seldom found in existing works.

The first tool we developed is SimStudio, an Eclipse-based environment for DEVS modeling. The main characteristic of SimStudio is that it employs a Model-Driven Engineering (MDE) approach, using a platform-independent DEVS metamodel as a pivot for representing and manipulating DEVS models. We developed this metamodel to be as generic as possible, allowing the representation of any DEVS model without any dependence to a particular programming language or simulator.

To do so, we modeled both atomic and coupled DEVS specifications, in terms of both structure and behavior. The latter aspect was the most challenging: to enable the definition of transition functions without relying on a particular general-purpose programming language, we devised our own programming language by taking the common denominator between the most mainstream languages. However, such common denominator is too limited to be practical, notably because it prevents the use of libraries or tool-dependent

features. Consequently, we introduced the possibility to include platform-specific snippets, i.e. pieces of code with no predefined semantics. The meaning of these snippets is determined by mappings that associate them with actual code in various platforms.

From this metamodel, the MDE framework we used automatically generated several artifacts for creating, editing and manipulating DEVS models conforming to it. For instance, we automatically obtained a set of Java classes corresponding to the metamodel, a serializer and a parser for saving and loading models from XML files, and an Eclipse plugin defining a tree view for editing models.

In addition to these features provided out of the box by the MDE environment, we leveraged a validation framework to include many model verifications, allowing the practitioner to check the correctness of his models. For instance, our metamodel forbids creating models with incorrect couplings or with direct feedback loops. These verifications are performed directly during model design, increasing the early detection of errors and thereby diminishing the development time.

Finally, we developed a number of transformations for processing DEVS models and generating various artifacts. The most important feature is code generation: given a model specification conforming to our metamodel, SimStudio can generate the corresponding code for various existing DEVS simulators such as CD++ or PythonDEVS. This generation is fully automated, meaning that no external intervention is needed to obtain a fully compliant model specification for a particular tool. In addition to code, SimStudio can also generate other elements such as diagrams of coupled models (in SVG) and XHTML documentation.

The MDE approach adopted in SimStudio provides many benefits, not only for the practitioner, i.e. the person that will use the environment to create DEVS models, but also for the toolsmith, i.e. the person that will maintain and extend the tool. The first advantage is the abstraction from implementation details: when most tools require models to be defined in a general-purpose programming language, for instance by subclassing some particular classes from a library, SimStudio allows the user to specify his models at a higher

level of abstraction, manipulating entities that are closer to its domain, with no need for prior knowledge.

In addition to making model specifications easier to write and comprehend, this raise in abstraction also reduces the risk of introducing bugs during implementation. Since the generation of tool-specific code is fully automated, it is not possible to make mistakes when mapping an abstract specification into a representation more suitable to simulation (assuming the code generators are bug-free themselves). The diminution of errors is further decreased by the thorough model verification performed by SimStudio: given a model specification, the environment is capable of analyzing it to detect inconsistencies with the DEVS formalism, such as incompatible couplings, incorrect tie-breaking function or invalid structure. Upon detection of such errors, SimStudio alerts the practitioner with comprehensive error messages, allowing him to quickly identify and correct problems.

Another important benefit of using platform-independent representations of DEVS models along with automatic generation of code for various target platforms is that it becomes much easier to test or compare tools. From a single specification, one can generate several versions of his model for several target simulators, for instance to try out different algorithms or to leverage a particular feature. Transitioning to a new simulator boils down to choosing a different target for the generation, assuming the appropriate transformation is defined. Moreover, the collaboration between practitioners is greatly simplified: even if two teams use different simulation tools, they can exchange and reuse models from each other seamlessly.

In fact, all the features we just presented could be provided equally by any M&S environment, not necessarily relying on MDE.As a matter of fact, existing tools already implement some of them. However, a major advantage of SimStudio, ensuing from the use of MDE, is that it is very easy to develop new features based on the pivot metamodel.

Indeed, each feature we described has been either automatically generated by the MDE framework or defined in a simple and concise way thanks to several specialized Domain-Specific Languages. For instance, the code for generating XHTML documentation from a

given DEVS model is about 200 lines, and was developed in a few hours. Using a general-purpose programming language, this would probably have taken weeks, with thousands of lines of code and a huge amount of debugging. The same remark applies to other features: model invariants are specified in a simple language that resembles usual mathematical notations; text generators are defined with a straightforward template system; textual languages are defined with plain EBNF grammars, the supporting tooling (editors, parsers, etc.) being automatically generated by the framework; graphical editors are automatically generated from high-level descriptions of their diagramming elements; and so on.

By providing several high-level abstractions for manipulating models conforming to some metamodel, MDE allows toolsmiths to develop new features with much less friction than before. This makes SimStudio a very extensible environment for DEVS M&S, where new functionalities can be added seamlessly by relying on the core DEVS metamodel.

As of now, SimStudio does not include simulation features. Instead, it relies on external simulators by providing code generators for several existing tools. However, we identified some shortcomings in these tools that we deemed rectifiable using a software engineering technique called metaprogramming. Therefore, we developed a C++ simulation library for Classic-DEVS, which solves these defects and can serve as a target platform for SimStudio or as a standalone simulator.

This library, named DEVS-MetaSimulator (DEVS-MS), adopts an innovative approach to DEVS simulation: instead of providing a generic simulator that handles all models in the same way, it processes models at compile-time to generate simulators that are fully specialized for those. We explained in this thesis how genericity is valuable from the user perspective, but incurs undesirable characteristics in terms of implementation, such as hindered performance and weak error detection. These defects disappear when writing specific software for simulating a particular model, but at the cost of a cumbersome and time-consuming development. With DEVS-MS, we provide the best of both worlds by having a generic library that specializes itself for the model it is given, in effect generating an executable that is fully dedicated to the simulation of this particular model and closely resembles what an experienced developer could have developed manually.

Considering DEVS-MS as a simulator generator is only one way to look at it. The other is to view it as a two-stage simulator, where some part of the simulation is performed at compile-time and the other at runtime. Indeed, the generation of specialized simulators is actually obtained by making the compiler perform several simulation operations ahead of execution, thanks to the C++ template mechanism. This programming technique, called C++ Template MetaProgramming, consists in shifting data from runtime to compile-time, in order to make it amenable to manipulation by the compiler.

In practice, DEVS-MS precomputes most aspects of the simulation that do not directly relate to modifying the state of the model. For instance, all iterations over components are performed statically (at compile-time), and so is the evaluation of tie-breaking functions. More importantly, the exchange of events between components is also handled during compilation: determining the source port, querying couplings to determine influenced components and retrieving the corresponding processors are all operations that are done prior to execution. This optimization is rather important, because it removes the unnecessary exchange of messages up and down the hierarchy of processors: in the end, the components participating in the simulation communicate directly, without intermediaries. Nevertheless, the hierarchical organization of the simulator is not totally suppressed, as opposed to the flattening approach. This allows certain operations to be performed in a simple and efficient way, notably the computation of the time of next event.

In addition to simulation operations, DEVS-MS also performs model verification at compile-time. Indeed, since many characteristics of the model are available statically, the compiler is able to detect inconsistencies and stop compilation accordingly. We implemented many checks in DEVS-MS, but the most important one is arguably port compatibility. By relying on the C++ type system, the library provides a rich type checking feature that can detect incorrect couplings, i.e. couplings where two ports with incompatible types are connected. The verification goes beyond asserting that every connected ports must generate and accept the same set of values: since it leverages the C++ type system, the library allows more powerful relationships such as subtyping, implicit and user-defined conversions. For instance, it is possible to connect two ports of different types if the destination type is a supertype of the source one, or if it defines an appropriate conversion constructor. Similarly, implicit conversions between fundamental types are also supported. Thereby, DEVS-MS

effectively provides a rich type system for ports, which protects the practitioner from design errors while still being flexible and intuitive.

Moving all these computations from runtime to compile-time leads to several benefits. The first one is an increase of performance: most of the simulation overhead is removed from the final executable, thanks to the static precomputations performed by the compiler. As a consequence, the resulting specialized simulator runs faster than a generic one. We demonstrated this by conducting some experiments, using a sample model to measure the performance of various simulations of the same model. The results showed that DEVS-MS greatly outperforms a similar simulator operating only at runtime, but also all but one of several implementations manually crafted especially for the model at hand.

The second advantage relates to model verification. By performing many consistency checks at compile-time, DEVS-MS catches design mistakes at an early stage of the development of models, allowing the practitioner to correct them right away. Moreover, error detection is much more thorough than when done at runtime. The compiler processes the model in its entirety: if there is a mistake somewhere, it will catch it. On the other hand, if the verification was done at runtime, errors could go undetected for a long time if they are hidden in an execution path that is seldom taken. These two characteristics reduce the time needed to develop models by shrinking the time allotted to debugging. The simple fact that a model specification compiles means that it is free of several potential bugs.

However, these benefits do not come for free. Performing so many computations statically put a heavy burden on the compiler, especially as it relies on an unforeseen use of the template mechanism. As a consequence, the resources needed to compile models quickly rise, resulting in very long compile times and even occasional memory exhaustion. This problem is mainly a quality of implementation issue in actual C++ compilers. Since template metaprogramming tends to become more widespread, we can expect compiler vendors to improve their support for this programming technique.

A second limitation of our approach is that it requires an important part of models to be defined statically. For instance, the set of components contained in a coupled model must be

known at compile-time, meaning that it is not possible to construct it at runtime based on some dynamic parameters. This restriction can be partially overcome by carrying out the construction of the model in a previous phase, using some code generation tool to automatically produce the appropriate specification based on some inputs. Another interesting solution would be to modify DEVS-MS so that every aspect of a model could be specified either statically or dynamically. Depending on his needs, the practitioner could chose to fix some aspects while leaving others variable. The library would provide more or less optimization and model verification, depending on the amount of information available at compile-time. Such an approach would greatly increase the applicability of the library, but would probably be very hard to implement.

II. Future works

Even though the works presented in this thesis are quite advanced, there are still some aspects that need to be finalized or improved. Regarding DEVS-MS, we want to try optimizing it to reduce compilation times and memory consumption. Several techniques exist to alleviate the load on compilers, based on the algorithms they use to handle template instantiations. We also want to perform some tests with alternative compilers, notably Clang, which seems to provide much better performance than its counterparts when dealing with template metaprograms.

When it comes to SimStudio, the priority is to complete the development of the semi-generic language used to describe the behavior of atomic models. At the time of this writing, we already defined the grammar of the language, but we still need to define the metamodel for mappings between platform-dependent snippets and the corresponding implementations in target platforms. Once this is done, we will be able to develop the model transformations needed to generate tool-specific code from code written in the semi-generic language.

Subsequently, we will be able to integrate the graphical editor developed concurrently by another PhD student, which provides a way of defining both atomic and coupled DEVS models with diagrams, using the DEVS-Driven Modeling Language (DDML) [Ighoroje and Traoré 2011]. This editor uses its own metamodel that maps closely to DDML. Thus, its

integration will consist in defining model-to-model transformations to produce SimStudio models from DDML models. The transformation of coupled models should be pretty straightforward, but we anticipate that the transformation from state diagrams—used in DDML to specify the behavior of atomic models—to semi-generic code should be more involved.

Finally, we still need to tie together the features we developed in a cohesive package. Indeed, the tooling we developed around our metamodel (editor with model checking, transformations, textual editor for the semi-generic language, etc.) is fully functional, but it is not bundled in a full-fledged environment. The idiomatic way of building such environment when working with the Eclipse Modeling Project is to create an Eclipse plugin embedding the various elements, which can then be deployed into installed Eclipses or converted to a standalone application.

In a longer term perspective, we think that the work presented here opens the way to many interesting follow-ups. First, it could be used by the DEVS community as a basis for thinking about what a future DEVS standard could look like. We showed that it is possible to devise a platform-independent metamodel for DEVS, and that it provided many benefits. These benefits would be even higher if such metamodel was common to the entire community, i.e. if there was a standardized format for representing DEVS models. This would not only make models more easily exchanged and reused, but also encourage the development of DEVS tools as they would have more chances of being adopted.

Second, it could serve as a “laboratory” for experimenting with inter-formalism transformations. By making DEVS models amenable to automated manipulation, SimStudio provides an environment where theories can be put into practice. We aim at exploiting SimStudio to attempt to develop automatic transformations from various formalisms to DEVS. Our first try could be with Petri Nets, for which many metamodels are already available. Ultimately, the goal would be to have a metamodel for each of the most prominent formalisms along with a set of transformations between those metamodels, thereby enabling seamless multi-formalism M&S.

Finally, we would like to identify M&S patterns and include them in SimStudio, if possible. The idea is to pinpoint and formalize best M&S practices, like design patterns do in software

engineering. Practitioners often stumble upon issues that have already been properly solved by others. Instead of going through the trouble of solving them on their own, they could benefit from the experience of their predecessors by referring to compilations of tried and tested solutions for their problems. At first, we simply want to identify such solutions, describe them precisely and classify them. Eventually, the application of these patterns could be facilitated by providing some support for them in SimStudio, for instance through templates or wizards.

References

[Abiteboul et al. 1999] Serge Abiteboul, Peter Buneman, and Dan Suciu, *Data on the Web: From Relational to Semistructured Data and XML*, Morgan Kaufmann, 1999, 258 p.

[Abrahams and Gurtovoy 2004] Dave Abrahams and Aleksey Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, Addison—Wesley Professional, 2004, 400 p.

[Albert et al. 2008] Vincent Albert, Alexandre Nketsa, Mario Paludetto, Marc Courvoisier, "Requirements related to the validity of a simulation," European Simulation and Modelling Conference (ESM 2008), Le Havre (France), October 27–29, 2008, pp. 7-13.

[Alexandrescu 2001] Andrei Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley Professional, 2001, 352 p.

[Al-Zoubi and Wainer 2008] Khaldoon Al-Zoubi and Gabriel A. Wainer, "Interfacing and Coordination for a DEVS Simulation Protocol Standard," 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications, Vancouver, BC, DS-RT 2008, October 27-29, 2008. pp. 300-307.

[Al-Zoubi and Wainer 2009] Khaldoon Al-Zoubi and Gabriel A. Wainer, "Using REST Web Services Architecture for Distributed Simulation," PADS '09 Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation, New York, USA, June 22-25, 2009, pp. 114-121.

[Ashenden 2008] Peter J. Ashenden, *The Designer's Guide to VHDL*, 3nd Edition, Morgan Kaufmann, May 2008, 936 p.

[Balci 1995] Osman Balci, "Principles of Simulation Model Validation, Verification, and Testing," Winter Simulation Conference 1995, Arlington, VA, USA, December 3–6, 1995, pp. 147-154.

[Bézivin 2006] Jean Bézivin, "On the Unification Power of Models," Software and System Modeling, Springer, Vol. 4, No. 2, May 2005, pp. 171-188.

[Bézivin and Gerbé 2001] Jean Bézivin and Olivier Gerbé, “Towards a Precise Definition of the OMG/MDA Framework,” Proceedings of the 16th Annual International Conference on Automated Software Engineering (ASE 2001), Nantes, France, November 26–29, 2001, pp. 273-280.

[Bézivin et al. 2003] Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette and Jamal Eddine Rougui, “First experiments with the ATL model transformation language: Transforming XSLT into XQuery,” In OOPSLA 2003 Workshop, Anaheim, California, October 26-30, 2003.

[Bézivin et al. 2004] Jean Bézivin, Frédéric Jouault and Patick Valduriez, “On the Need for Megamodels,” Best Practices for Model-Driven Software Development Workshop (co-located with OOPSLA 2004 and GPCE 2004), Vancouver, Canada, October 25, 2004.

[Bolduc and Vangheluwe 2002] Jean-Sébastien Bolduc and Hans Vangheluwe, “A modeling and simulation package for classic hierarchical DEVS,” Internal document for the Modelling, Simulation and Design Lab (MSDL), School of Computer Science, McGill University, 2002.

[Boost.Preprocessor] Vesa Karvonen and Paul Mensonides, The Boost Preprocessor Metaprogramming library, www.boost.org/doc/libs/release/libs/preprocessor/doc/index.html Last accessed 30/04/2012.

[Brutzman and Zyda 2002] Don Brutzman and Michael Zyda, “Extensible Modeling and Simulation Framework (XMSF) - Challenges for Web-Based Modeling and Simulation,” Technical Challenges Workshop, Strategic Opportunities Symposium, MOVES Institute, Naval Postgraduate School, October 2002, 52 p.

[Chen and Vangheluwe 2010] Bin Chen and Hans Vangheluwe, “Symbolic flattening of DEVS models,” 2010 Summer Simulation Multiconference, Ottawa, ON, Canada, July 11–15, 2010, pp. 209-218.

[Chepovsky et al. 2003] Andrei M. Chepovsky, Andrei V. Klimov, Arkady V. Klimov, Yuri A. Klimov, Andrei S. Mischenko, Sergei A. Romanenko and Sergei Yu. Skorobogatov,

"Partial Evaluation for Common Intermediate Language," Perspectives of System Informatics, LNCS, Vol. 2890, 2003, pp. 171-177.

[Clang] Clang: a C language family frontend for LLVM, http://clang.llvm.org/ Last accessed 30/04/2012.

[Crosbie 2012] Roy Crosbie, "Real-Time Simulation using Hybrid Models," in *Real-time Simulation Technologies: Principles, Methodologies, and Applications*, Katalin Popovici and Pieter J. Mosterman (Editors), CRC Press, August 2012, pp 3-34.

[Damaševičius and Štuikys 2008] Robertas Damaševičius and Vytautas Štuikys, "Taxonomy of the Fundamental Concepts of Metaprogramming," Information Technology And Control, Kaunas, Technologija, 2008, Vol. 37, No. 2, pp. 124-132.

[Damus 2007] Christian W. Damus, "Implementing Model Integrity in EMF with MDT OCL," Eclipse corner article, http://www.eclipse.org/articles/article.php?file=Article-EMF-Codegen-with-OCL/index.html Last accessed 06/06/2012.

[De Lara and Vangheluwe 2002] Juan De Lara and Hans Vangheluwe, "AToM3: A Tool for Multi-formalism and Meta-modelling," In Fundamental Approaches to Software Engineering 5th International Conference FASE 2002 held as Part of the Joint European Conferences on Theory and Practice of Software ETAPS 2002, Grenoble, France, April 8-12, 2002, Springer Vol. 2306, pp. 174-188.

[DEVS Standardization Group] DEVS Standardization Group site: http://cell-devs.sce.carleton.ca/devsgroup/ Last accessed March 2012.

[DEVSJava 2004] ACIMS software site: http://www.acims.arizona.edu/SOFTWARE/software.shtml Last accessed March 2012.

[Diaw et al. 2010] Samba Diaw, Rédouane Lbath and Bernard Coulette, "État de l'art sur le développement logiciel basé sur les transformations de modèles," in Technique et Sciences Informatiques, Vol. 29, No 4-5, 2010, pp. 505-536.

[D'Orazio and Traoré 2009] Laurent D'Orazio and Mamadou K. Traoré, "Semantic Caching for Pervasive Grids," Thirteenth International Database Engineering & Applications Symposium, Cetraro, Calbria, Italy, September 16-18, 2009, pp. 227-233.

[E 2011] Weinan E, *Principles of multiscale modeling*, Cambridge University Press, 2011, 488 p.

[EMP logo 2006] Preliminary logo of the Eclipse Modeling Project https://bugs.eclipse.org/bugs/attachment.cgi?id=48508 Last accessed 25/04/2012.

[Favre 2004a] Jean-Marie Favre, "Foundations of Model (Driven) (Reverse) Engineering: Models - Episode I, Stories of the Fidus Papyrus and of the Solarus," In Proceedings of the Dagstuhl Seminar 04101 on Language Engineering for Model-Driven Software Development, Dagsthul, Germany, February 29-March 5, 2004.

[Favre 2004b] Jean-Marie Favre, "Foundations of the Meta-pyramids: Languages and Metamodels - Episode II, Story of Thotus the Baboon," In Proceedings of the Dagstuhl Seminar 04101 on Language Engineering for Model-Driven Software Development, Dagsthul, Germany, February 29-March 5, 2004.

[Favre et al. 2006] Jean-Marie Favre, Jacky Estublier and Mireille Blay-Fornarino, *L'ingénierie dirigée par les modèles : au-delà du MDA*, Hermes-Lavoisier, Cachan, France, February 2006, 236 p.

[Feng et al. 2008] Bo Feng, Qi Liu, Gabriel A. Wainer, "Parallel simulation of DEVS and Cell-DEVS models on Windows-based PC cluster systems," Proceedings of the 2008 Spring simulation multiconference, Ottawa, ON, Canada, April 14-17, 2008, pp. 439-446.

[Filippi et al. 2010] Jean-Baptiste Filippi, Teruhisa Komatsu and David R.C. Hill, "Environmental Models in DEVS: Different Approaches for Different applications," in *Discrete Event Simulation and Modeling: Theory and Applications*, Gabriel A. Wainer and Pieter J. Mosterman (Editors), Taylor and Francis Group, 2010, Chapter 14, pp. 357-386.

[Fishwick 2002] Paul A. Fishwick, “Using XML for Simulation Modeling,” Proceedings of the 2002 Winter Simulation Conference, San Diego, California, USA, December 8-11, 2002, Vol. 1, pp. 616-622.

[Foster and Kesselman, 1998] Ian Foster and Carl Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1998, 675 p.

[Fowler 2005] Martin Fowler, “Language Workbenches: The Killer-App for Domain-Specific Languages?,” http://www.martinfowler.com/articles/languageWorkbench.html, June 2005, Last accessed 20/04/2012.

[Fowler 2010] Martin Fowler, *Domain-Specific Languages*, Addison—Wesley Professional, September 2010 ,640 p.

[Glinsky and Wainer 2002] Ezequiel Glinsky and Gabriel A. Wainer, “Definition of Real-Time simulation in the CD++ toolkit,” Proceedings of the 2002 SCS Summer Computer Simulation Conference, San Diego, California, USA, July 14-18, 2002.

[Glinsky and Wainer 2005] Ezequiel Glinsky and Gabriel Wainer, “DEVSTONE: a Benchmarking Technique for Studying Performance of DEVS Modeling and Simulation Environments,” Proceedings of IEEE DS-RT, Montréal, QC, October 10-12, 2005, pp. 265-272.

[González et al. 2010] Andrés Senac González, Gregorio Martinez Perez, Diego Sevilla Ruiz, “EMF4CPP: a C++ Ecore Implementation,” DSDM 2010 - Desarrollo de Software Dirigido por Modelos, Jornadas de Ingenieria del Software y Bases de Datos (JISBD 2010), Valencia, Spain, September 2010, Vol. 4, No. 2, article 13.

[Gordon 1962] Geoffrey M. Gordon, “A General Purpose Systems Simulator,” IBM Systems Journal Vol. 1 No. 1, 1962, pp. 18-32.

[Greenfield and Short 2004] Jack Greenfield and Keith Short, *Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools*, John Wiley and Sons, August 2004, 500 p.

[Gregor et al. 2006] Douglas Gregor, Jaako Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis and Andrew Lumsdaine, “Concepts: linguistic support for generic programming in C++,” ACM SIGPLAN Notices - Proceedings of the 2006 OOPSLA Conference, Portland, Oregon, USA, October 22-26, 2006, Vol. 41, Issue 10, pp. 291-310.

[Gronback 2009] Richard C. Gronback, *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*, Addison—Wesley Professional, March 2009, 736 p.

[GSAT 2005] Groupe Simulation – Académie des Technologies, “Enquête sur les frontières de la simulation numérique en France. La situation en France et dans le monde. Diagnostic et propositions,” Report from the Académie des Technologies. May 2005.

[Hemel and Visser 2010] Zef Hemel and Eelco Visser, “PIL: A Platform-Independent Language for Retargetable DSLs,” Lecture Notes in Computer Science, 2010, Volume 5969/2010, pp. 224-243.

[Herrmann and Langhammer 2006] Christoph A. Herrmann and Tobias Langhammer, “Combining partial evaluation and staged interpretation in the implementation of domain-specific languages,” Science of Computer Programming, Vol. 62, No. 1, September 2006, pp. 47-65.

[Hill 1993] David R.C. Hill, *Analyse Orientée-Objets et Modélisation par Simulation*, Addison—Wesley, January 1993, 362 p.

[Hill 1996] David R.C. Hill, *Object-Oriented Analysis and Simulation*, Addison—Wesley Longman, November 1996, 320 p.

[Hill 2002a] David R.C. Hill, “Object-Oriented Modelling And Post-Genomic Biology Programming Analogies,” Proceedings of Artificial Intelligence, Simulation and Planning (AIS 2002), Lisbon, Portugal, April 7-10, 2002, pp. 329-334.

[Hill 2002b] David R.C. Hill, “URNG: A portable optimisation technique for every software application requiring pseudorandom numbers,” Simulation Modelling Practice and Theory, Vol. 11, 2002, pp. 643-654.

[Hill and Gourgand 1993] David R.C. Hill and Michel Gourgand, “A Multi-Domain Tool for Object-Oriented Simulation,” In TOOLS 10, Prentice Hall, Versailles, France, 1993, pp. 181-195.

[Hill and Roche 2002] David R.C. Hill and Alexandre Roche, “Benchmark of the unrolling of pseudorandom numbers generators,” 14th European Simulation Symposium, Dresden, Germany, October 23-26, 2002, pp. 119-129.

[Himmelspach 2007] Jan Himmelspach, *Konzeption, Realisierung und Verwendung eines allgemeinen Modellierungs, Simulations und Experimen tiersystems*, PhD thesis, University of Rostock, November 2007.

[Himmelspach and Uhrmacher 2006] Jan Himmelspach and Adelinde M. Uhrmacher, “Sequential processing of PDEVS models,” In: Agostino G. Bruzzone and Antoni Guasch and Miguel Angel Piera and Jerzy Rozenblit (ed.): Proceedings of I3M, LogiSim, Barcelona, Spain, October 4-6, 2006, pp. 239-244.

[Himmelspach and Uhrmacher 2007] Jan Himmelspach, Roland Ewald and Adelinde M. Uhrmacher, “Plug'n Simulate,” Simulation Symposium, ANSS '07, Norfolk, VA, USA, March 26-28, 2007, pp. 137-143.

[Ighoroje 2010] Ufuoma B. Ighoroje, *A Graphical Modeling Framework for Modeling and Simulation of Dynamic Systems*, Master (MSc) Thesis, African University of Science and Technology, 2010.

[Ighoroje and Traoré 2011] Ufuoma B. Ighoroje and Mamadou K. Traoré, “Eclipse-DDML: A Graphical Modeling Tool for Dynamic System Simulation,” Proceedings of the European Simulation and Modeling Conference, Guimaraes, Portugal, October 24-26, 2011, pp. 61-66.

[Ingalls et al. 1988] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph and Ken Doyle, “Fabrik: A Visual Programming Environment,” In ACM SIGPLAN Notices - Special issue: Object-Oriented Programming Systems, Languages and Applications (OOPSLA '88), Vol. 23, Issue 11, November 1988, pp. 176-190.

[Jafer 2011] Shafagh Jafer, *Parallel simulation techniques for large-scale discrete-event models*, Ph.D. Thesis, Carleton University, Canada, 2011, 141 p.

[Jafer and Wainer 2009] Shafagh Jafer and Gabriel A. Wainer, “Flattened Conservative Parallel Simulator for DEVS and CELL-DEVS,” International Conference on Computational Science and Engineering, Vancouver, Canada, August 29-31, 2009, pp. 443-448.

[Jammalamadaka et al. 2007] Rajanikanth Jammalamadaka, Bernard P. Zeigler, Ming Zhang, Mark E. Gettings and Mark Bultman, “Complex system simulation: DEVS implementation of the valley fever model,” In Proceedings of the 2007 Spring Simulation Multiconference, SpringSim 2007, Norfolk, Virginia, USA, March 25-29, 2007, Volume 2, pp. 316-319.

[Janoušek et al. 2006] Vladimír Janoušek, Petr Polášek, Pavel Slavícek, “Towards DEVS Meta Language,” ISC 2006 Proceedings, Samos, Greece, August 30-September 2, 2006, pp. 69-73.

[Jones 1996] Neil D. Jones, “An Introduction to Partial Evaluation,” ACM Computing Surveys, Vol. 28, No. 3, 1996, pp. 480-503.

[Jouault and Bézivin 2006] Frédéric Jouault and Jean Bézivin, “KM3: a DSL for Metamodel Specification,” In Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, LNCS 4037, Bologna, Italy, June 14-16, pp. 171-185.

[Jouault and Kurtev 2006] Frédéric Jouault and Ivan Kurtev, “On the architectural alignment of ATL and QVT,” Proceedings of the 2006 ACM symposium on Applied Computing, Dijon, France, April 23-27, 2006, pp. 1188-1195.

[Jouault et al. 2006] Frédéric Jouault, Jean Bézivin and Ivan Kurtev, “TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering,” In GPCE'06: Proceedings of the fifth international conference on Generative programming and Component Engineering, Portland, Oregon, USA, October 22-26, 2006, pp. 249-254.

[Keith at al. 2000] Keith H. Bennett, Paul Layzell, David Budgen, Pearl Brereton, Linda Macaulay and Malcom Munro, “Service-based software: the future for flexible software,” Proceedings of the Seventh Asia-Pacific Software Engineering Conference, Singapore, December 5-8, 2000, pp. 214-221.

[Kendall 1953] David G. Kendall, “Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain,” The Annals of Mathematical Statistics, Vol. 24, No. 3, September 1953, pp. 338-354.

[Kiczales et al. 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin, “Aspect-Oriented Programming,” Proceedings of the 1997 European Conference on Object-Oriented Programming, Jyväskylä, Finland, June 9-13, 1997, New York: Springer-Verlag, pp. 220-242.

[Kim and Kang 2004] Ki-Hyung Kim and Won-Seok Kang, “CORBA-based, Multi-threaded Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-hierarchical One,” In Proceedings of the International Conference on Computational Science and Its Applications (ICCSA 2004), Assisi, Italy, May 14-17, 2004 LNCS 3046: pp. 167-176.

[Kim et al. 2000] Kihyung Kim, WonSeok Kang, Bong Sagong and Hyungon Seo, “Efficient distributed simulation of hierarchical devs models: Transforming model structure into a non-hierarchical one,” In Proceedings of the 33rd Annual Simulation Symposium (SS 2000), Washington, DC, USA, April 16-22, 2000, pp. 227.

[Kim et al. 2009] Sungung Kim, Hessam S. Sarjoughian and Vignesh Elamvazhuthi, “DEVS-Suite: a Simulator Supporting Visual Experimentation Design and Behavior Monitoring,” In Proceedings of the 2009 Spring Simulation Multiconference, San Diego, CA, USA, March 22-27, 2009, Article No. 161.

[Kleppe et al. 2003] Anneke Kleppe, Jos Warmer and Wim Bast, *MDA Explained: The Model Driven Architecture™: Practice and Promise*, Addison-Wesley Professional, May 2003, 192 p.

[Kuhl et al. 1999] Frederick Kuhl, Richard Weatherly and Judith Dahmann, *Creating computer simulation systems: an introduction to the high level architecture*, Prentice Hall, October 1999, 224 p.

[L'Ecuyer et al. 2002] Pierre L'Ecuyer, Lakhdar Meliani and Jean Vaucher, “SSJ: A Framework for Stochastic Simulation in Java,” In Proceedings of the 2002 Winter SimulationConference, San Diego, California, USA, December 8-11, 2002, Vol. 1, pp. 234-242.

[Lengauer 2003] Christian Lengauer, “Program Optimization in the Domain of High-Performance Parallelism,” Domain-specific program generation. International seminar, Dagstuhl Castle , Germany, March 23-28, 2003, LNCS Vol. 3016, pp. 73-91.

[Levine et al. 1992] John Levine, Tony J.R. Mason and Doug Brown, *Lex & Yacc*, 2nd edition, O’Reilly & Associates, October 1992, 388 p.

[Levytskyy et al. 2003] Andriy Levytskyy, Eugène J.H. Kerckhoffs, Ernesto Posse and Hans Vangheluwe, “Creating DEVS components with the meta-modelling tool AToM3,” In Alexander Verbraeck and Vlatka Hlupic, editors, 15th European Simulation Symposium (ESS), Society for Modeling and Simulation International (SCS), Delft, The Netherlands, October 26-29, 2003, pp. 97-103.

[Liskov and Zilles 1974] Barbara Liskov and Stephen N. Zilles, “Programming with abstract data types,” ACM SIGPLAN Notices, Vol. 9, No .4, April 1974, pp. 50-59.

[Mellor and Balcer 2002] Stephen J. Mellor and Marc J. Balcer, *Executable UML: A foundation for model-driven architectures*, Addison-Wesley Professional, May 2002, 416 p.

[Meyer 1988] Bertrand Meyer, “Genericity Versus Inheritance,” Journal of Pascal, Ada, & Modula-2, Vol. 7, No. 2, March/April 1988, pp. 13-30.

[Minsky 1965] Marvin L. Minsky, “Matter, Minds and Models,” International Federation of Information Processing Congress, Vol. 1, 1965, pp. 45-49.

[Missaoui et al. 2008] Mohieddine Missaoui, David R.C. Hill and Pierre Peyret, “A Comparison of Algorithms for a Complete Backtranslation of Oligopeptides,” Special issue, IJCBDD,

International Journal of Computational Biology and Drug Design, 2008, Vol. 1, No. 1, pp. 26-38.

[Mittal et al. 2007] Saurabh Mittal, José-Luis Risco-Martín and Bernard P. Zeigler, “DEVSML: automating DEVS execution over SOA towards transparent simulators,” Proceedings of SpringSim 2007, Norfolk, VA, USA, March 25-29, 2007, Vol. 2, pp. 287-295.

[Mittal et al. 2009] Saurabh Mittal, José L. Risco-Martín and Bernard P. Zeigler, “DEVS/SOA: A Cross-Platform Framework for Net-centric Modeling and Simulation in DEVS Unified Process,” SIMULATION, July 2009, Vol. 85, No. 7, pp. 419-450.

[MOF 2002] OMG, Meta Object Facility (MOF) Specification v.1.4, April 2002, http://www.omg.org/spec/MOF/1.4/ (last accessed 19/04/2012)

[Møller and Schwartzbach 2006] Anders Møller and Michael I. Schwartzbach, *An Introduction to XML and Web Technologies*, Addison-Wesley, January 2006, 568 p.

[Müller 2010] Jean-Pierre Müller, “A Framework for Integrated Modeling using a Knowledge-Driven Approach,” In Proceedings of the 2010 International Congress on Environmental Modelling and Software (iEMSs 2010), Ottawa, ON, Canada, July 5-8, 2010, Article S.21.08.

[Muller et al. 2005] Pierre-Alain Muller, Franck Fleurey and Jean-Marc Jézéquel, “Weaving Executability into Object-Oriented Meta-Languages,” In S. Kent L. Briand, editors, Proceedings of MODELS/UML'2005, LNCS 3713, Montego Bay, Jamaica, October 2-7, 2005, pp. 264-278.

[Muzy and Hill 2011] Alexandre Muzy and David R.C. Hill, “What Is New With The Activity World View In Modeling And Simulation? Using Activity As A Unifying Guide For Modeling And Simulation,” IEEE/SMC – ACM/SIGSIM- Proceedings of the 2011 Winter Simulation Conference S. Jain, R.R. Creasey, J. Himmelspach, K.P. White, and M. Fu, eds. Invited Paper, 13 p.

[Muzy and Nutaro 2005] Alexandre Muzy and James J. Nutaro, “Algorithms for efficient implementation of the DEVS and DS-DEVS abstract simulators,” In Open International

Conference on Modeling and Simulation (OICMS'05), Clermont-Ferrand, France, June 12-15, 2005, pp. 273-279.

[Newcomer and Lomow 2004] Eric Newcomer and Greg Lomow, *Understanding Soa With Web Services*, Addison-Wesley Professional, December 2004, 480 p.

[Niebler 2007] Eric Niebler, "Proto: A Compiler Construction Toolkit for DSELs," Proceedings of the 2007 Symposium on Library-Centric Software Design, Montreal, Canada, October 21, 2007, pp. 42-51.

[Ninios et al. 1995] Panagiotis Ninios, Kiriakos Vlahos and Derek W. Bunn, "Industry simulation: System modelling with an object oriented/DEVS technology," European Journal of Operational Research, Volume 81, Issue 3, March 1995, pp. 521-534.

[NSF 2006] National Science Foundation, *Simulation-based engineering science. Revolutionizing engineering science through simulation*, NSF Report, May 2006.

[Nutaro 2010] James J. Nutaro, *Building Software for Simulation: Theory and Algorithms, with Applications in C++*, Wiley, December 2010, 347 p.

[O'Reilly and Nordlund 1989] Jean J. O'Reilly and Kristen C. Nordlund, "Introduction to SLAM II and SLAMSYSTEM," Proceedings of the 21st Winter Simulation Conference, Washington, D.C., USA, December 4-6, 1989, pp. 178-183.

[Pedgen et al. 1995] Claude D. Pegden, Robert E. Shannon and Randall P. Sadowski, *Introduction to Simulation Using SIMAN*, 2nd edition, McGraw-Hill Companies, 1995, 600 p.

[PITAC 2005] President's Information Technology Advisory Committee, *Computational science: ensuring America's competitiveness*, Report to the President, 2005.

[Poslad 2009] Stefan Poslad, *Ubiquitous Computing: Smart Devices, Environments and Interactions*, Wiley, May 2009, 502 pages.

[Posse and Bolduc 2003] Ernesto Posse and Jean-Sébastien Bolduc, "Generation of DEVS modelling and simulation environments," In A. Bruzzone and Mhamed Itmi, editors,

Summer Computer Simulation Conference, Student Workshop, Society for Computer Simulation International (SCS), Montréal, Canada, July 20-24, 2003, pp. S139-S146.

[Potier 1983] Dominique Potier, *New User's Introduction to QNAP2*, INRIA technical report No. 40, October 1983.

[Quesnel 2006] Gauthier Quesnel, *Approche formelle et opérationnelle de la multi-modélisation et de la simulation des systèmes complexes*, Ph.D. Thesis, Université du Littoral de la Côte d'Opale, 2006.

[Quesnel et al. 2009] Gauthier Quesnel, Raphaël Duboz and Eric Ramat, "The Virtual Laboratory Environment -- An operational framework for multi-modelling, simulation and analysis of complex dynamical systems," Simulation Modelling Practice and Theory, Vol. 17, April 2009, pp. 461-653.

[Rhöl and Uhrmacher 2005] Mathias Rhöl and Adelinde .M. Uhrmacher, "Flexible Integration of XML into Modeling and Simulation Systems," Proceedings of the 2005 Winter Simulation Conference, Orlando, FL, USA, December 4-7, 2005, pp. 1813-1820.

[Rhöl et al. 2008] Mathias Röhl and Adelinde M. Uhrmacher, "Definition and analysis of composition structures for discrete-event models," Proceedings of the 2008 Winter Simulation Conference, Miami, FL, USA, December 7-10, 2008, pp. 942-950.

[Risco-Martín et al. 2007] José-Luis Risco-Martín, Saurabh Mittal, Miguel Ángel López-Peña and Jesús Manuel de la Cruz, "A W3C XML schema for DEVS scenarios," Proceedings of SpringSim 2007, Norfolk, VA, USA, March 25-29, 2007, Vol.2, pp. 279-286.

[Saidani et al. 2009] Tarik Saidani, Joël Falcou, Claude Tadonki, Lionel Lacassagne and Daniel Etiemble, "Algorithmic Skeletons within an Embedded Domain Specific Language for the CELL Processor," Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, Raleigh, USA, September 12-16, 2009, pp. 67-76.

[Schmidt 2006] Douglas C. Schmidt, "Model-Driven Engineering – Guest Editor's Introduction," IEEE Computer, Vol. 39, No. 2, February 2006, pp. 25-31.

[Seck et al. 2004] Mamadou Seck, Claudia Frydman and Norbert Giambiasi, “Using DEVS for modeling and simulation of human behaviour,” In Proceedings of the 13th international conference on AI, Simulation, and Planning in High Autonomy Systems, Jeju Island, Korea, October 4-6, 2004, pp. 692-698.

[Seo 2009] Chungman Seo, *Interoperability between DEVS Simulators using Service Oriented Architecture and DEVS Namespace*, Ph.D. thesis, Electrical and Computer Engineering Dept., University of Arizona, Spring 2009.

[Seo et al. 2004] Chungman Seo, Sunwoo Park, Byounguk Kim, Saehoon Cheon and Bernard P. Zeigler, “Implementation of Distributed High-performance DEVS Simulation Framework in the Grid Computing Environment,” In Proceedings of the 2004 Advanced Simulation Technologies Conference – High-Performance Computing Symposium (ASTC’04), Arlington, VA, USA, April 18-22, 2004.

[Shang and Wainer 2005] Hui Shang and Gabriel A. Wainer, “A model of virus spreading using Cell-DEVS,” In Proceedings of the 5th international conference on Computational Science, Atlanta, USA, May 22-25, 2005, Vol. 2, pp. 373-377.

[Sheard 2001] Tim Sheard, “Accomplishments and research challenges in meta‑programming,” Proceedings of the 2nd International Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG’2001), Florence, Italy, September 6, LNCS, Vol. 2196, pp. 2-44.

[Shlaer and Mellor 1991] Sally Shlaer and Stephen J. Mellor, *Object Lifecycles: Modeling the World in States*, Prentice Hall, September 1991, 251 p.

[SVG] World Wide Web Consortium, Scalable Vector Graphics (SVG) 1.1 specification (Second Edition), http://www.w3.org/TR/SVG11/ Last accessed 18/06/2012

[Taha 1999] Walid Taha, *Multi‑Stage Programming: Its Theory and Applications*, PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

[Taha and Sheard 1997] Walid Taha and Tim Sheard, “Multi-Stage Programming with Explicit Annotations,” Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and

Semantic Based Program Manipulations PEPM'97, Amsterdam, The Netherlands, June 12-13, 1997, pp. 203-217.

[Touraille et al. 2009] Luc Touraille, Mamadou K. Traoré and David R. C. Hill, “A mark-up language for the storage, retrieval, sharing and interoperability of DEVS models,” Proceedings of the 2009 Spring Simulation Multiconference, article No. 163, 6 p.

[Touraille et al. 2011] Luc Touraille, Mamadou K. Traoré and David R.C. Hill, “A model-driven software environment for modeling, simulation and analysis of complex systems,” In Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium, Boston, MA, USA, April 4-7, 2011, pp. 229-237.

[Touraille et al. 2012] Luc Touraille , Jonathan Caux and David Hill, “Optimizing Discrete Modeling and Simulation for Real Time Constraints,” In Katalin Popovici and Pieter J. Mosterman (editors), *Real-time Simulation Technologies: Principles, Methodologies, and Applications*. Boca Raton, FL, USA: CRC Press, August 2012, pp. 97-122.

[Traoré 2008] Mamadou K. Traoré, “SimStudio: a next generation modeling and simulation framework,” Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems, Marseille, France, March 3-7, 2008, 6 p.

[Traoré 2009] Mamadou K. Traoré, “A Graphical Notation for DEVS,” HPCS: a joint DEVS and HPC Symposium. San Diego, CA, March 22-27, 2009, 7 p.

[Truyen 2006] Frank Truyen, *The Fast Guide to Model Driven Architecture: The Basics of Model Driven Architecture, Practice*. Available at: http://corba.com/mda/mda_files/Cephas_MDA_Fast_Guide.pdf Last accessed 19/04/2012

[Vangheluwe 2000] Hans L. M. Vangheluwe, “DEVS as a Common Denominator for Multi-formalism Hybrid Systems Modelling,” IEEE International Symposium on Computer-Aided Control System Design, Anchorage, Alaska, USA, September 25-27, 2000, pp. 129-134.

[Vangheluwe et al. 2002] Hans L. M. Vangheluwe, Juan De Lara and Pieter J. Mosterman, “An Introduction to Multi-Paradigm Modelling and Simulation,” In Fernando Barros and Norbert Giambiasi, editors, Proceedings of the AIS'2002 Conference (AI, Simulation and Planning in High Autonomy Systems), Lisboa, Portugal, April 7-10, 2002, pp. 9-20.

[Veldhuizen 1998] Todd L. Veldhuizen, “Arrays in Blitz++,” Proceedings of the 2nd International Scientific Computing in Object Oriented Parallel Environments (ISCOPE'98), Santa Fe, USA, December 8-11, 1998, LNCS, Vol. 1505, pp. 223-230.

[Veldhuizen 2003] Todd L. Veldhuizen, “C++ Templates are Turing Complete,” Technical report, Indiana University, 2003.

[Veldhuizen and Gannon 1998] Todd L. Veldhuizen and Dennis Gannon, “Active Libraries: Rethinking the Roles of Compilers and Libraries,” Proceedings of the 1998 SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing, New York, USA, October 21-23, 1998, pp. 286-295.

[Venhola and Wainer 2006] Wilson Venhola and Gabriel Wainer, “DEVSView: A tool for visualizing CD++ simulation models,” In Proceedings of SpringSim 2006 (DEVS Symposium), Huntsville, AL, USA, April 2-6, 2006, 8 p.

[Wachsmuth 2007] Guido Wachsmuth, “Metamodel Adaptation and Model Co-adaption,” ECOOP’07, Berlin, Germany, July 30-August 3, 2007, LNCS Vol. 4609, pp. 600-624.

[Wainer 2006] Gabriel A. Wainer, “Applying Cell-DEVS Methodology for Modeling the Environment,” SIMULATION: Transactions of the Society for Modeling and Simulation International, Vol. 82, No. 10, October 2006, pp. 635-660.

[Wainer 2009] Gabriel A. Wainer, *Discrete-Event Modeling and Simulation: a Practitioner’s approach*, CRC Press. Taylor and Francis, April 2009, 520 p.

[Wainer and Castro 2011] Gabriel A. Wainer and Rodrigo Castro, “DEMES: a Discrete-Event methodology for Modeling and simulation of Embedded Systems,” Modeling and Simulation Magazine, Society for Modeling and Simulation International, No. 2,, April 2011

[Wainer and Mosterman 2010] Gabriel A. Wainer and Pieter J. Mosterman. *Discrete-Event Modeling and Simulation: Theory and Applications*, Taylor & Françis, December 2010, 534 p.

[Wainer et al. 2010] Gabriel A. Wainer, Khaldoon Al-Zoubi, David R.C. Hill, Saurabh Mittal, José-Luis Risco-Martín, Hessam Sarjoughian, Luc Touraille, Mamadou K. Taoré, Bernard P. Zeigler, "An Introduction to DEVS Standardization," in *Discrete-Event Modeling and Simulation: Theory and Application*s, Gabriel A. Wainer and Pieter Mosterman (editors), Taylor & Françis, December 2010, Chapter 16, pp.393-425.

[Walmsley 2001] Priscilla Walmsley, *Definitive XML Schema*, Prentice Hall, December 2001, 560 p.

[Wang and Lu 2002] Yung-Hsin Wang, Yao-Chung Lu, "An XML-based DEVS Modeling Tool to Enhance Simulation Interoperability, " Proceeding of the 14th European Simulation Symposium, Dresden, Germany, October 23-26, 2002, pp. 406-412.

[Warmer and Kleppe 2003] Jos Warmer and Anneke Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*, 2nd edition, Addison-Wesley Professional, September 2003, 240 p.

[Zeigler 1976] Bernard P. Zeigler, *Theory of Modeling and Simulation*, New York: John Wiley, 1976.

[Zeigler 1998] Bernard P. Zeigler, *DEVS theory of quantized systems*, Advanced simulation technology thrust DARPA contract, June 1998, 63 p.

[Zeigler and Louri 1993] Bernard P. Zeigler and Ahmed Louri, "A Simulation Environment for Intelligent Machine Architectures," Journal of Parallel and Distributed Computing Volume 18, Issue 1, May 1993, pp. 77-88.

[Zeigler and Sarjoughian 2003] Bernard P. Zeigler and Hessam Sarjoughian, "Introduction to DEVS modeling and simulation with Java: Developing component-based simulation models," http://www.acims.arizona.edu/SOFTWARE/devsjava_licensed/CBMSManuscript.zip

[Zeigler et al. 2000] Bernard P. Zeigler, Tag Gon Kim and Herbert Praehofer, *Theory of Modeling and Simulation*, Academic Press, Inc., Orlando, FL, 2000.

Appendices

Appendix A – DEVS metamodel in OCLinEcore

The following is the full definition of our DEVS metamodel. It is written in a domain-specific language, OCLinEcore, which allows developing Ecore models efficiently thanks to a clear and concise syntax.

```
import ecore : 'http://www.eclipse.org/emf/2002/Ecore#/';

package DEVS : devs = 'http://www.isima.fr/limos/devs/1.1'
{
    class DocumentedElement { abstract }
    {
       attribute _'documentation' : String[?];
    }
    class Port extends DocumentedElement { abstract }
    {
        attribute name : String[1];
        property type : ecore::EClassifier[1];
        property model#ports : Model[1];
    }
    class Model extends DocumentedElement { abstract }
    {
        invariant PortNamesUniqueness:
            ports->forAll(p1, p2 : Port | p1 <> p2 implies p1.name <>
p2.name);
        attribute name : String[1];
        property ports#model : Port[*] { !ordered,composes };
    }
    class Component extends DocumentedElement
    {
        attribute name : String[1];
        property type : Model[1];
        property containingModel#components : CoupledModel[1];
    }
    class Coupling { abstract };
    class SelectMapping
    {
        invariant ActivableComponentIsInTheImminentSet:
            componentSet->exists(c | c = priorComponent);
        property componentSet : Component[+] { !ordered };
        property priorComponent : Component[1] { !ordered };
    }
    class CoupledModel extends Model
    {
        invariant ComponentNamesUniqueness:
            components->forAll(c1, c2 : Component | c1 <> c2 implies
c1.name <> c2.name);
        invariant ComponentsInSelectMappingsBelongsToSelf:
            select->forAll(s : SelectMapping | components-
>includesAll(s.componentSet));
        property components#containingModel : Component[+] {
!ordered,composes };
        property couplings : Coupling[*] { !ordered,composes };
```

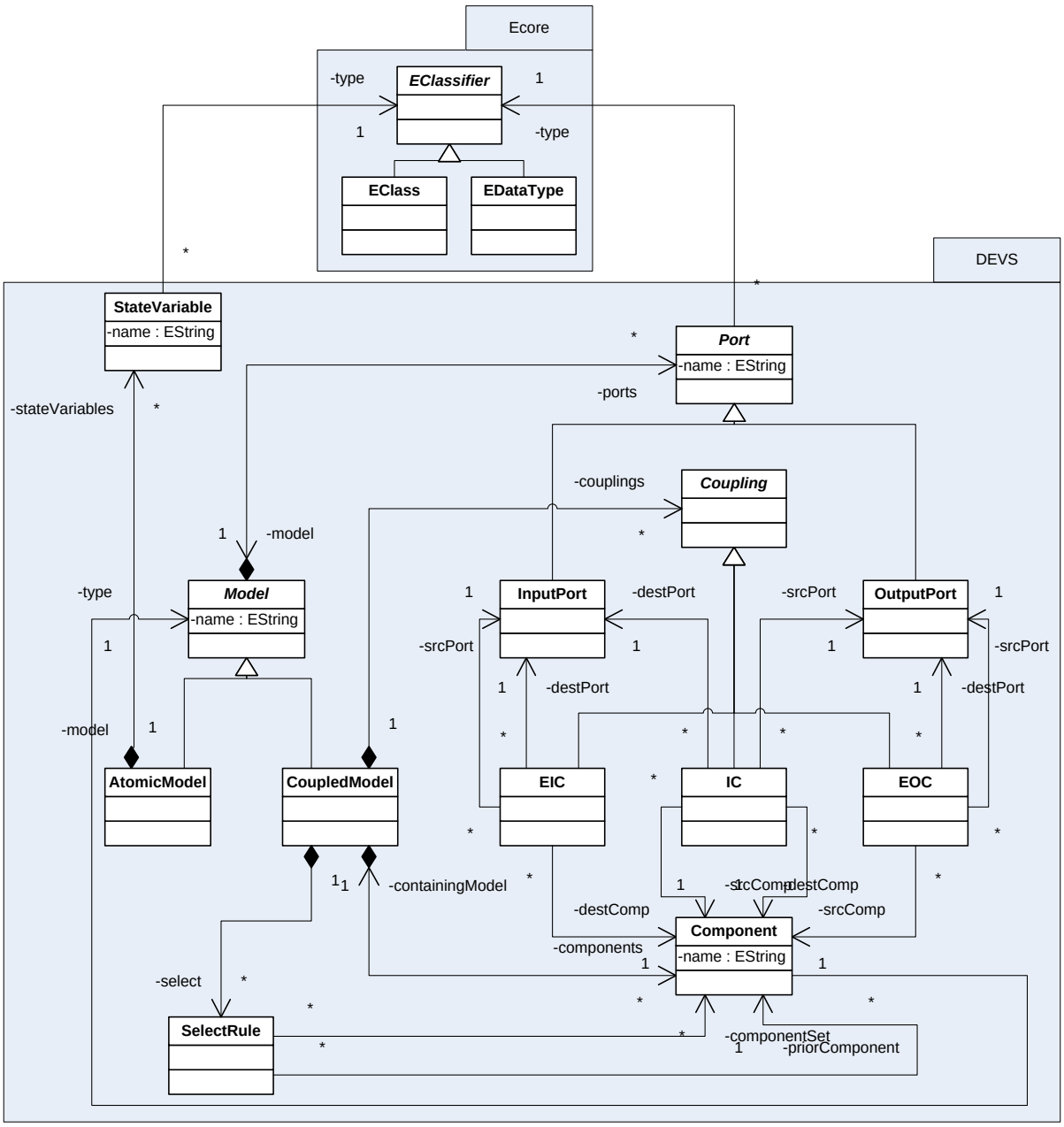
```
        property select : SelectMapping[+] { composes };
    }
    class AtomicModel extends Model
    {
        invariant StateVariableNamesUniqueness:
            stateVariables->forAll(sv1, sv2 : StateVariable | sv1 <> sv2
implies sv1.name <> sv2.name);
        attribute delta_int : String[1];
        property stateVariables : StateVariable[*] { !ordered,composes };
        attribute delta_ext : String[1];
        attribute lambda : String[1];
        attribute ta : String[1];
    }
    class StateVariable extends DocumentedElement
    {
        attribute name : String[1];
        property type : ecore::EClassifier[1];
    }
    class EIC extends Coupling
    {
        invariant PortBelongsToComponent:
            destComponent.type = destPort.model;
        invariant PortCompatibility:
            (srcPort.type.oclIsKindOf(ecore::EDataType) and srcPort.type =
destPort.type) or
            let destPortClass : ecore::EClass =
destPort.type.oclAsType(ecore::EClass) in

destPortClass.isSuperTypeOf(srcPort.type.oclAsType(ecore::EClass));
        invariant SourcePortFromSelf:
            srcPort.model.oclIsKindOf(CoupledModel) and
            let model : CoupledModel =
srcPort.model.oclAsType(CoupledModel) in
            model.couplings->exists(c | c = self);
        property destComponent : Component[1];
        property srcPort : InputPort[1];
        property destPort : InputPort[1];
    }
    class IC extends Coupling
    {
        invariant PortsBelongToComponents:
            srcComponent.type = srcPort.model and
            destComponent.type = destPort.model;
        invariant PortCompatibility:
            (srcPort.type.oclIsKindOf(ecore::EDataType) and srcPort.type =
destPort.type) or
            let destPortClass : ecore::EClass =
destPort.type.oclAsType(ecore::EClass) in

destPortClass.isSuperTypeOf(srcPort.type.oclAsType(ecore::EClass));
        invariant NoDirectFeedbackLoop:
            srcComponent <> destComponent;
        property srcComponent : Component[1];
        property destComponent : Component[1];
        property srcPort : OutputPort[1];
        property destPort : InputPort[1];
    }
    class EOC extends Coupling
    {
        invariant PortBelongsToComponent:
            srcComponent.type = srcPort.model;
```

```
        invariant PortCompatibility:
            (srcPort.type.oclIsKindOf(ecore::EDataType) and srcPort.type =
destPort.type) or
            let destPortClass : ecore::EClass =
destPort.type.oclAsType(ecore::EClass) in

destPortClass.isSuperTypeOf(srcPort.type.oclAsType(ecore::EClass));
        invariant DestPortToSelf:
            destPort.model.oclIsKindOf(CoupledModel) and
            let model : CoupledModel =
destPort.model.oclAsType(CoupledModel) in
            model.couplings->exists(c | c = self);
        property srcComponent : Component[1];
        property srcPort : OutputPort[1];
        property destPort : OutputPort[1];
    }
    class InputPort extends Port;
    class OutputPort extends Port;
}
```

Appendix B – Diagram of the DEVS metamodel

This diagram shows the relation between the various elements of our DEVS metamodel. It merges the diagrams presented in Chapter 5.II.2.1.