

To Masha and Helena.

Promotors: Prof. dr. ir. Peter Vanrolleghem
Department of Applied Mathematics, Biometrics and Process Control (BIOMATH)
Ghent University, Belgium

Prof. dr. Peter Fritzson
Programming Environments Laboratory (PELAB)
Linköping University, Sweden

Dean: Prof. dr. ir. Herman Van Langenhove

Rector: Prof. dr. Paul Van Cauwenberge

Filip CLAEYS

**A Generic Software Framework for Modelling and
Virtual Experimentation with Complex
Environmental Systems**

Thesis submitted in fulfillment of the requirements
for the degree of Doctor (PhD) in Applied Biological Sciences:
Environmental Technology

Dutch translation of the title:

Een Generisch Software-raamwerk voor Modelleren en Virtueel Experimenteren met Complexe Milieusystemen

Please refer to this work as follows:

Filip H.A. Claeys. *A generic software framework for modelling and virtual experimentation with complex biological systems*. PhD thesis. Dept. of Applied Mathematics, Biometrics and Process Control, Ghent University, Belgium, January 2008, pp. 303.

ISBN-number: 978-90-5989-223-1

The author and the promotor give the authorization to consult and to copy parts of this work for personal use only. Every other use is subject to the copyright laws. Permission to reproduce any material contained in this work should be obtained from the author.

Acknowledgements

“Unorthodox” is probably the word that springs to mind when looking at the multi-disciplinary nature of this dissertation, and the entire process that lead to it. Indeed, writing a PhD dissertation at the age of 40, while being employed as CTO of a startup company, and while experiencing an eventful private life, is not your every day situation, at least not in Belgium. To be honest, there have been times during the write-up process that I was ready to give up, but thanks to the support of my wife Masha, and - in her own way - also our little girl Helena, I managed to pull through.

As usual for any PhD, there are many people to thank. However, since the work that is described in this dissertation has been carried out over a longer period of time, this is even more so in this particular case. This dissertation would never have seen the light of day without the vision and endless enthusiasm of my promotors: profs. Peter Vanrolleghem and Peter Fritzson. I first started working with prof. Peter Vanrolleghem back in 1995 and have enjoyed our collaboration until the present day. Peter is a man of many talents, the most important for me personally being his ability and willingness to think beyond the boundaries of his own domain of expertise. I am also grateful to Peter Fritzson for making time for reviewing my dissertation during his sabbatical overseas and during the busy period of the creation of the Open Source Modelica Consortium.

As mentioned before, the work discussed in this dissertation was carried out over a longer period of time. During the first period, which lasted from 1995 to 1998, I still had a lot to learn and was therefore happy to be able to enjoy the mentorship of an extremely bright and witty person: Hans Vangheluwe. Hans introduced me to the beauty of multi-disciplinary research and let me assist him in laying the foundations for the current WEST modelling and virtual experimentation software. During that first period, I also enjoyed working with many other people, software developers and modellers alike. Some names that spring to mind are (in no particular order) Bhama Sridharan, Jonathan Levine, Sebastiaan Kops, Henk Vanhooren, Jurgan Meirlaen, Diederik Rousseau, Diedert Debusscher, Frederik Decouttere, Britta Petersen, Dirk Stevens and Bart Vanderhaegen.

During the second period, which lasted from 2003 until the time of writing, being a bit wiser, I enjoyed passing on some of the knowledge I had gained over the years to another “generation” of young researchers. Some of the people I am grateful to for their contribution to this second period are (again in no particular order) Lorenzo Benedetti, Dirk De Pauw, Ingmar Nopens, Petra Claeys, Karel Van Laer, Webbey De Keyser, Gürkan Sin, Michael Rademaker and Cyril Garneau. Especially Lorenzo and Dirk must be acknowledged for the substantial impact they have had on the current state of the Tornado framework.

Evidently, I also want to thank MOSTforWATER and its CEO Dirk Van der Stede for partially sponsoring my research. Other colleagues at MOSTforWATER, both experts in their own particular field, who have had a substantial impact on my work are Youri Amerlinck and Stefan De Grande. Enrico Remigi and Denis Vanneste only joined the company recently, but I also enjoyed working with them thus far.

Tornado and WEST in their various incarnations have received sponsoring from the IWT, Ghent University and Aquafin. I therefore wish to thank these organizations for their continued support and their belief in the potential of the software.

Finally, I am grateful to my parents. They have always supported me during my long lasting career as a student, and have never too much questioned my often unorthodox career moves ...

Filip Claeys
Melle, January 8th, 2008

Contents

1	Introduction and Objectives	15
I	Concept Exploration and Requirements	21
2	State-of-the-Art	22
2.1	Modelling and Virtual Experimentation Concepts	22
2.1.1	General	22
2.1.2	Equation-based Mathematical Model Representation	25
2.1.3	Coupled Model Representation	27
2.1.4	Petersen Matrix Representation	27
2.2	Complex Environmental Systems - Case: Water Quality Management	28
2.3	Existing Generic Frameworks	31
2.3.1	Introduction	31
2.3.2	ACSL	32
2.3.3	MATLAB/Simulink	32
2.3.4	Mathematica	33
2.3.5	Mathcad	34
2.3.6	LabVIEW	35
2.3.7	Maple	36
2.3.8	Modelica	37
2.4	Existing Water Quality Tools	38
2.4.1	Aquasim	38
2.4.2	BioWin	39
2.4.3	GPS-X	40
2.4.4	Simba	40
2.4.5	STOAT	41
2.4.6	WEST	41
3	Problem Statement	43
3.1	Complex Modelling	43
3.2	Complex Virtual Experimentation	46
3.3	Deployment and Integration Flexibility	47
3.4	Modern Architectural Standards	47
3.5	Conclusion	48

4	Requirements	49
4.1	Introduction	49
4.2	Complex Modelling	49
4.3	Complex Virtual Experimentation	52
4.4	Deployment and Integration Flexibility	53
4.5	Modern Architectural Standards	53
II	Design and Implementation	54
5	Design Principles	55
5.1	Introduction	55
5.2	Overall Architecture	55
5.3	Entities	55
5.4	Properties	57
5.5	Exceptions	58
5.6	Events	58
5.7	Design Patterns	59
5.7.1	Singleton	59
5.7.2	Factory	59
5.7.3	Facade	59
5.7.4	Proxy	59
5.8	Thread-safety	60
5.9	Persistency	60
5.10	Naming	60
6	Materials and Methods	63
6.1	Introduction	63
6.2	Object-oriented Programming	68
6.3	Advanced C++	69
6.3.1	Coding Style	70
6.3.2	Standard Library	70
6.3.3	Namespaces	72
6.3.4	Exceptions	72
6.3.5	Streaming Operators	73
6.3.6	Interfaces	74
6.4	XML	75
6.5	Third-party Software Components	76
6.6	Development Environment	76
7	Common Library	78
7.1	Introduction	78
7.2	Encryption	78

7.3	Dynamically-loaded Libraries	79
7.4	Exceptions	79
7.5	Strings	79
7.6	Properties	80
7.7	Manager	81
7.8	Mathematics	81
7.8.1	Aggregation Functions	81
7.8.2	Sampling from Statistical Distributions	84
7.8.3	Interpolation	86
7.9	Platform	88
7.10	Vectors and Matrices	89
8	Experimentation Environment	91
8.1	Introduction	91
8.2	Hierarchical Virtual Experimentation	91
8.2.1	Basic Concepts	91
8.2.2	Input Providers	96
8.2.3	Output Acceptors	98
8.2.4	General Form of a Virtual Experiment	98
8.3	Virtual Experiment Types	99
8.3.1	ExpSimul: Dynamic Simulation	100
8.3.2	ExpSSRoot and ExpSSOptim: Steady-state Analysis	100
8.3.3	ExpObjEval: Objective Evaluation	104
8.3.4	ExpScen: Scenario Analysis	110
8.3.5	ExpMC: Monte Carlo Analysis	117
8.3.6	ExpSens: Sensitivity Analysis	120
8.3.7	ExpOptim and ExpCI: Optimization and Confidence Information	126
8.3.8	ExpStats: Statistical Analysis	129
8.3.9	ExpEnsemble: Ensemble Simulation	130
8.3.10	ExpSeq: Sequential Execution	132
8.3.11	ExpScenOptim / ExpMCOptim: Optimization Scenario / Monte Carlo Analysis	133
8.3.12	ExpScenSSRoot: Steady-state Analysis with Root Finder Scenario Analysis	133
8.4	Generalized Framework for Abstraction and Dynamic Loading of Numerical Solvers	135
8.4.1	Introduction	135
8.4.2	Solver Availability	136
8.4.3	Embedding Solvers into Quality Software: Integration Issues	136
8.4.4	The Tornado Solver Framework	138
8.5	Managing Graphical Input Provider and Output Acceptor Representations	142
8.6	Internationalization and Localization	142
8.7	Improving Simulation Speed through a priori Exploration of the Solver Setting Space	145
8.7.1	Description of the Procedure	145
8.7.2	Application to the Lux Case	147

9	Modelling Environment	151
9.1	Introduction	151
9.2	Modelling Languages	151
9.2.1	MSL	151
9.2.2	Modelica	157
9.3	Executable Models	161
9.3.1	Symbolic Information	162
9.3.2	Computational Information	164
9.4	Model Compilers	169
9.4.1	MSL: Introduction	169
9.4.2	MSL: Front-end	170
9.4.3	MSL: Back-end	172
9.4.4	Modelica: Front-end	173
9.4.5	Modelica: Back-end	174
9.5	Layouts	183
9.6	ModelLib: Model Library	183
9.7	IconLib: Icon Library	185
III	Deployment	189
10	Application Programming Interfaces	190
10.1	Introduction	190
10.2	Comprehensive API's	191
10.2.1	C++	191
10.2.2	.NET	196
10.3	Restricted API's	204
10.3.1	C	204
10.3.2	JNI	206
10.3.3	MEX	207
10.3.4	OpenMI	209
10.3.5	CMSLU	209
10.3.6	CLIPS	210
11	Command-line User Interface	212
11.1	Introduction	212
11.2	Components of the CUI Suite	213
11.3	Flow of Data	216
11.4	Usage	216
12	Graphical User Interfaces	219
12.1	Introduction	219
12.2	Tornado-I	219

12.2.1	WEST++	219
12.2.2	WEST-3	221
12.2.3	EAST	222
12.3	Tornado-II	222
12.3.1	MORE	222
12.3.2	WEST-4	224
13	Remote Execution	229
13.1	Introduction	229
13.2	Sockets	230
13.3	CORBA	231
13.4	DCOM	232
13.5	OPC	232
13.6	Java RMI	233
13.7	SOAP	233
13.8	.NET Remoting	234
13.9	ASP.NET	237
13.10	Conclusion	238
14	Distributed Execution	239
14.1	Introduction	239
14.2	Gridification of Virtual Experiments	240
14.3	Typhoon	244
14.4	gLite	248
14.5	Adoption of Distributed Execution in the CD4WC Project	248
14.6	Adaptive Scheduling	250
IV	Future Perspectives and Conclusions	253
15	Future Perspectives	254
15.1	Introduction	254
15.2	Modelling Environment	254
15.2.1	Acausal Systems and Simplification of Equations	254
15.2.2	Hybrid Systems	255
15.2.3	Partial Differential Equations	255
15.3	Experimentation Environment	256
15.3.1	New Virtual Experiment Types	256
15.3.2	Intelligent Solver Configuration	258
15.4	Distributed Computing	258
15.4.1	Job Length Estimation	258
15.4.2	Checkpointing	259
15.5	Virtual Engineering	259

15.5.1	Version management	260
15.5.2	Collaborative Development	261
15.5.3	Workflow management	261
15.5.4	Conclusion	262
16	Conclusions	263
16.1	Overview	263
16.2	Complex Modelling	264
16.3	Complex Virtual Experimentation	265
16.4	Deployment and Integration Flexibility	266
16.5	Modern Architectural Standards	267
16.6	Overall Conclusion	267
V	Appendices	277
A	Integration Solvers	278
A.1	Introduction	278
A.2	Simple Methods	279
A.2.1	AE	279
A.2.2	ODE: Explicit Single-Step Methods	280
A.2.3	ODE: Explicit Multi-Step Methods	281
A.2.4	ODE: Explicit Predictor-Corrector Methods	281
A.3	Complex Methods	281
A.3.1	ODE	281
A.3.2	DAE	283
A.3.3	HybridDAE	284
B	Root-finding Solvers	285
B.1	Introduction	285
B.2	Broyden	285
B.3	Hybrid	285
C	Optimization Solvers	286
C.1	Introduction	286
C.2	Praxis	286
C.3	Simplex	286
C.4	GA	287
C.5	SA	287
D	Monte Carlo Solvers	288
D.1	Introduction	288
D.2	CVT	288
D.3	IHS	289

D.4 LHS	289
D.5 PR	289
E Features of Domain-Specific Water Quality Tools	290
F List of Abbreviations	293
G Summary	297
H Samenvatting	300

List of Figures

2.1	Modelling and Virtual Experimentation Concepts	23
2.2	Evolution of the Level of Integration in Water Quality Management	29
2.3	Timeline and Dependencies of WEST and Related Software	42
3.1	Evolution of Model Complexity in Water Quality Management (Linear Scale)	44
3.2	Evolution of Model Complexity in Water Quality Management (Logarithmic Scale)	44
3.3	Moore's Law (Logarithmic Scale)	45
5.1	Overall Architecture of the Tornado Kernel	56
5.2	Entities of the Tornado Kernel	57
6.1	Props XML Schema Definition	75
7.1	Common Library: Exception Classes	80
7.2	Common Library: Properties Class	82
7.3	Common Library: Manager Class	83
8.1	Experiment Inheritance Hierarchy	92
8.2	Experiment Composition Hierarchy	93
8.3	Experiment Composition Hierarchy and Number of Runs	94
8.4	General Form of a Virtual Experiment	99
8.5	Exp XML Schema Definition	101
8.6	ExpSimul ER Diagram	102
8.7	ExpSimul XML Schema Definition	103
8.8	ExpSSRoot ER Diagram	104
8.9	ExpSSRoot XML Schema Definition	104
8.10	ExpSSOptim ER Diagram	104
8.11	ExpSSOptim XML Schema Definition	105
8.12	ExpObjEval ER Diagram	105
8.13	ExpObjEval XML Schema Definition	106
8.14	ExpScen ER Diagram	111
8.15	ExpScen XML Schema Definition	112
8.16	ExpScen Solver Procedures	119
8.17	ExpMC ER Diagram	120
8.18	ExpMC XML Schema Definition	120
8.19	ExpSens ER Diagram	123

8.20	ExpSens XML Schema Definition	124
8.21	ExpOptim ER Diagram	127
8.22	ExpOptim XML Schema Definition	127
8.23	ExpCI ER Diagram	128
8.24	ExpCI XML Schema Definition	128
8.25	ExpStats ER Diagram	129
8.26	ExpStats XML Schema Definition	130
8.27	ExpEnsemble ER Diagram	131
8.28	ExpEnsemble XML Schema Definition	132
8.29	ExpSeq ER Diagram	134
8.30	ExpSeq XML Schema Definition	134
8.31	ExpScenSSRoot ER Diagram	135
8.32	ExpScenSSRoot XML Schema Definition	135
8.33	Solver Plug-ins	138
8.34	Solver Inheritance Hierarchy Excerpt	139
8.35	Controls ER Diagram	143
8.36	Controls XML Schema Definition	143
8.37	Plot ER Diagram	144
8.38	Plot XML Schema Definition	144
8.39	Solver Setting Scenario Analysis Procedure	148
8.40	Number of <i>ComputeState</i> Evaluations with Respect to Absolute and Relative Tolerance for the 0-1-0 Pattern	150
8.41	TIC with Respect to Absolute and Relative Tolerance for the 0-1-0 Pattern	150
9.1	Flow of the Executable Model Generation Process	162
9.2	Compute Event Routines	165
9.3	Executable Model ER Diagram	166
9.4	MSL Model Compiler	170
9.5	Relationship between <i>tmsl</i> , <i>omc</i> and <i>mof2t</i>	173
9.6	Relationship between <i>omc</i> and <i>mof2t</i>	175
9.7	Bounds Violation Handling Mechanism implemented in <i>mof2t</i>	181
9.8	Layout ER Diagram	183
9.9	Layout XML Schema Definition	184
9.10	ModelLib ER Diagram	184
9.11	ModelLib XML Schema Definition	184
9.12	IconLib ER Diagram	185
9.13	IconLib XML Schema Definition	185
10.1	Tornado Application Programming Interfaces	191
10.2	The .NET Framework	197
10.3	Correspondence between Calls, Callbacks (Events) and Exceptions in .NET and C++	198
10.4	Building and Deploying Applications using the Tornado JNI API	207

11.1 Tornado CUI Suite Data Flow	218
12.1 WEST++ Macro-Modules	221
12.2 WEST++ GUI Applied to the BSM1 Case	222
12.3 WEST-3 GUI Applied to the BSM1 Case	223
12.4 WEST-3 Model Library Browser and Model Editor	224
12.5 WEST-3 Petersen Matrix Editor	225
12.6 MORE GUI Applied to the OUR Case	226
12.7 WEST-4 Project Explorer and Properties Sheet	227
12.8 WEST-4 Applied to the BSM1 Case	228
14.1 Flow of Execution of the ExpOptim and ExpScen Experiment Types	241
14.2 Typhoon Job XML Schema Definition	242
14.3 Modes of Operation for ExpScen/ExpMC	242
14.4 Grid Node Types	243
14.5 Relationship between Distributed Execution and Modes of Operation	243
14.6 High-level Typhoon Architecture	244
14.7 Detailed Typhoon Architecture	246
14.8 Embedding of Resources in Typhoon Job Descriptions	247
14.9 File Resource Identifiers in Typhoon Job Descriptions	247
14.10 HTTP Resource Identifiers in Typhoon Job Descriptions	247
14.11 Conceptual Diagram of the DSIDE Simulator	252

List of Tables

2.1	Petersen Matrix Representation of a Process Model	28
2.2	Bacterial Growth in an Aerobic Environment	28
2.3	Standard WWTP Unit Process Models	29
2.4	Complexity of a Number of Selected Water Quality Models	30
3.1	Comparison of Modelling and Virtual Experimentation Tools	46
5.1	Tornado's Coherent Naming Scheme	62
6.1	Comparison of Programming Languages	64
6.2	Timing Results of the ToMMTi Micro-benchmark for Algorithmic and Trigonometric Code	65
6.3	Timing Results of a Simulation Benchmark Program	68
6.4	Naming Conventions for Identifiers in C++ Code	70
7.1	Property Data Members	81
7.2	Cubic Spline Symbol Descriptions	88
7.3	Index Checking Strategies for the <i>Common::Vector*</i> Data Type	90
8.1	Input Generator Symbol Descriptions	97
8.2	ExpObjEval Symbol Descriptions	107
8.3	ExpScen Symbol Descriptions	112
8.4	ExpMC Symbol Descriptions	121
8.5	ExpSens Symbol Descriptions	125
8.6	ExpEnsemble Symbol Descriptions	132
8.7	Popular Numerical Solver Repositories	136
8.8	Arguments of the <i>SolveInteg::Configure()</i> Method	139
8.9	Properties of the CVODE Integration Solver	141
8.10	Special Objectives that are available for Scenario Analysis	146
8.11	Comparison between Standard and Solver Setting Scenario Analysis	147
8.12	Solver Settings for the Lux Case	148
8.13	Properties of <i>AbsoluteTolerance</i> and <i>RelativeTolerance</i> Scenario Analysis Variables	149
8.14	Total Number of Failures, Average Number of State Evaluations and Average TIC Value per Pattern	149
9.1	Operators in MSL	154
9.2	MSL Model Libraries	155

9.3	Modelica Model Libraries	160
9.4	Performance of C Compilers for the Galindo_CL Model (Windows platform)	168
9.5	<i>mof2t</i> AST Node Types	186
9.6	<i>mof2t</i> Functions	187
9.7	Code Instrumentations	187
9.8	Results of the Application of <i>mof2t</i> to Various WWTP Models	188
10.1	Mapping between C++ and .NET Data Types in the Tornado .NET API	198
11.1	Tornado CUI Suite	215
13.1	Relevant Remote Execution Technologies and their Application to Tornado	238
14.1	Design Cases studied in the CD4WC Project	249
14.2	Upgrade Cases studied in the CD4WC Project	249
A.1	Integrator Solver Symbol Descriptions	279
E.1	Features of Domain-Specific Water Quality Tools (Part 1)	291
E.2	Features of Domain-Specific Water Quality Tools (Part 2)	292

List of Algorithms

1	Generation of a Random Value from a Multi-modal Distribution	87
2	Computation of Cubic Spline Coefficients	89
3	Interpolation using Cubic Spline Coefficients	89
4	Computation of the Number of Lower Bound Violations	109
5	Computation of the Percentage of Time in Violation of Lower Bound	110
6	Generation of an Exhaustive List of Variable Value Vector Index Combinations	115
7	Generation of Sequentially-ordered Value Vectors	115
8	Generation of Randomly-ordered Value Vectors	116
9	Generation of Fixed Value Vectors	116
10	Generation of Grid-ordered Value Vectors	117
11	Generation of Cross-ordered Value Vectors	118
12	Unit Conversion Selection	145

Listings

6.1	Forward Euler Integration of a Small ODE System in C#	66
6.2	Example of Strings in C++	70
6.3	Example of Smart Pointers in C++	71
6.4	Example of Vectors in C++	71
6.5	Example of Dictionaries in C++	72
6.6	Example of Namespaces in C++	72
6.7	Example of Exceptions in C++	73
6.8	Example of Streaming Operators in C++	73
6.9	Example of Interfaces in C++	74
6.10	Props XML Schema Definition	75
9.1	MSL Description of a Fat Crystallization Model	155
9.2	Modelica Description of a Fat Crystallization Model	159
9.3	BNF Description of the <i>mof2t</i> Flat Modelica Grammar	176
10.1	Tornado C++ API (Excerpt)	192
10.2	Execution of an Experiment through the Tornado C++ API	192
10.3	Creation and Execution of an ExpOptim Experiment through the Tornado C++ API . . .	194
10.4	Execution of an Experiment through the Tornado .NET API (C#)	198
10.5	Creation and Execution of an ExpOptim Experiment through the Tornado .NET API (C#)	199
10.6	Creation and Execution of an ExpOptim Experiment through the Tornado .NET API (VB.NET)	201
10.7	Execution of an ExpSimul Experiment in a Separate Thread through the Tornado .NET API	203
10.8	Tornado C API	204
10.9	Execution of an Experiment through the Tornado C API	205
10.10	Tornado JNI API	206
10.11	Execution of an Experiment through the Tornado JNI API	207
10.12	Examples of the Application of the Tornado MEX API	208
10.13	Example of the Application of the Tornado C MSLU API	210
10.14	Example of the Application of the Tornado CLIPS API	211
11.1	On-line Help for the <i>mof2t</i> Command	216
11.2	On-line Help for the <i>tmain</i> Command	217
13.1	.NET Remoting Tornado Client	234
13.2	.NET Remoting Tornado Client Configuration File	236
13.3	.NET Remoting Tornado Server	236
13.4	.NET Remoting Tornado Server Configuration File	237

1

Introduction and Objectives

Computer-based modelling and virtual experimentation (*i.e.*, any procedure in which the evaluation of a model is required, such as simulation and optimization) has proven to be a powerful mechanism for solving problems in many areas, including environmental science. Since the late 1960's, a *plethora* of software systems have been developed that allow for modelling, simulation, and to a lesser extent, also other types of virtual experimentation. However, given the persistent desire to model more complex systems (enhanced by a growing insight into these systems), and the trend towards more complex computational procedures based on model evaluation, it may be required to re-evaluate and improve existing software frameworks, or even to suggest new frameworks. Moreover, recent developments in information technology have caused a growing trend towards flexible deployment and integration of software components. In order to be able to handle the need for malleable deployment and integration of modelling and virtual experimentation systems with other software components, re-evaluation of existing frameworks and/or development of new frameworks may also be required.

When inspecting sub-domains of environmental science and technology in the broadest sense, it can be noted that the need for computer-based modelling and virtual experimentation varies. In some disciplines complex models that require efficient execution mechanisms are ubiquitous, whereas other domains have much lower demands. Also, some disciplines are more mature than others, in the sense that standardized methodologies, procedures, components, *etc.* exist that are known by the majority of scientists active within that discipline, hence creating a *lingua franca*. Unfortunately, other domains are still lacking this level of maturity and have a long way towards consolidation of ideas and procedures.

One particular domain of environmental science that has in recent years attracted the interest of researchers in the field of software frameworks, is water quality management (which, amongst other, includes the study of water quality in treatment plants, sewer networks and river systems). This domain can be considered mature since the inception of standards such as the Activated Sludge Model (ASM) series (Henze et al., 2000) and the River Water Quality Model (RWQM) series (Reichert et al., 2001). Moreover, the complexity of the most advanced integrated models that are currently studied in this domain is such that mainstream software systems simply cannot handle them anymore. New or modified systems that take these changing requirements into account are therefore in order.

The work that is presented in this dissertation concerns the design and implementation of an advanced framework for modelling and virtual experimentation with complex environmental systems, which attempts to accomplish four major goals:

- Deliver support for **complex modelling**
- Deliver support for **complex virtual experimentation**
- Offer a wide variety of **deployment and integration** options
- Comply with **modern architectural standards**

As most systems that are studied in environmental science are continuous, focus is on complex continuous system modelling. The sources of model complexity are mainly the number of phenomena studied, and the detail at which they are studied. The sources of complexity of virtual experimentation are the number of model evaluations that are required to perform the experiment, and the computational load that is a result hereof.

The framework that is presented is generic in nature, but the design and implementation process has been strongly guided by demands coming from the field of water quality modelling. Although this fact has not impeded the genericity of the framework design, it has had an effect on the priorities set to implementation tasks. Certain features that may be indispensable in other fields, but are not strictly necessary in the field under consideration, will therefore be tackled in future work.

The work that has led to this dissertation had been carried out over a longer period of time, *i.e.*, mainly from 1995 until 1998 and from 2003 until 2007. The WEST (Vanhooren et al., 2003) modelling and virtual experimentation product that is currently commercially available (and is mainly used for modelling and simulation of wastewater treatment plants) is an emanation of the work that was done during the first period (*i.e.*, 1995-1998). The work done during the second period (2003-2007), has resulted into a software framework named “Tornado” and should eventually find its way to WEST-4, which is a major rewrite of the former WEST-3 product. Although the framework that was developed during 1995-1998 did not receive an official name, it will be referred to it as “Tornado-I” during the remainder of the text. The framework that was developed during 2003-2007 will be referred to as “Tornado-II” in case there would be a potential for confusion, otherwise “Tornado” will be used. In any case, the bulk of this document concerns “Tornado-II”. It is only to explain certain design or implementation decisions that “Tornado-I” will be involved in the discussion. For, the fact that work has been done during two periods has allowed for evaluation and re-consideration of a number of decisions taken during the first period.

The dissertation is divided into four parts, each of which are further subdivided into a number of chapters:

- The **Concept Exploration and Requirements** part gives a review of the current state-of-the-art with respect to modelling and virtual experimentation solutions. Also, it is stated why currently none of these solutions, no matter how powerful, are a perfect match for the current demands imposed by complex environmental (*i.e.*, water quality management) problems. Finally, a detailed list of requirements is established for a system that fulfills the four goals that were stated above, in the context of complex environmental problems.
- The **Design and Implementation** part discusses the design and the implementation of the framework that was developed. It first describes the design principles that were adopted and the development tools that were used. Subsequently, the implementation of the framework is extensively discussed. Mixed with this discussion is an overview of the features of the framework that was developed and results from the application of the framework to a number of relevant cases.
- The **Deployment** part focuses on the wide range of possible deployment options for the framework that was developed. Command-line user interfaces, graphical user interfaces, application programming interfaces, distributed execution and remote execution are discussed.

-
- Finally, the **Future Perspectives and Conclusions** part makes a round-up of the relevance of the framework with respect to the requirements that were put forward. Also, an initial *impetus* is given to several potential directions of future research.

The work that is presented in this dissertation can be considered multi-disciplinary. As mentioned before, the domain of water quality management has strongly guided the process and is assumed to be the main area of application of the framework that was developed. However, during the design and implementation process itself, focus was mainly on a number of sub-domains of computer science and mathematics, *i.e.*, software engineering, algorithms and complex datatypes, numerical analysis, compiler technology, computer algebra and distributed computing. For the remainder of this document, it is assumed that the reader at least has a basic understanding of computer science concepts. Although nearly all computer science concepts and methodologies that are adopted in the framework are shortly explained in the text, a clear understanding of many of the rationales that have led to design decisions require a theoretical and preferably also a practical background in computer science.

The work presented in this document is multi-facetted. It is difficult to single out one aspect to serve as most important contribution. It is believed that several contributions are of equal importance. These include the following:

- Design and implementation of a working **advanced modelling and virtual experimentation framework** (Claeys et al., 2006b) that allows to solve problems (mainly in the water quality domain) that hitherto could not be solved in a timely and/or convenient manner. The framework is based on high-level, declarative, object-oriented modelling and the use of model compilers to generate efficient executable code, which is hence used during the execution of virtual experiments. The development of the Tornado framework was mainly done in C++, adhering to modern design and implementation principles.
- Development of an extensible framework for **hierarchical virtual experimentation**, *i.e.*, for hierarchically structuring highly-configurable virtual experiment types. These virtual experiments allow for dynamic simulation, steady-state analysis, optimization, sensitivity analysis, scenario analysis and uncertainty analysis. In total, 15 different types of virtual experiments have been implemented. The hierarchical virtual experimentation framework is discussed in Chapter 8.
- Development of a **generalized framework for abstraction and dynamic loading of numerical solvers** (Claeys et al., 2006f), *i.e.*, run-time loading of numerical algorithms for tasks such as integration, optimization, solving sets of non-linear equations, Latin Hypercube Sampling (LHS), *etc.* These numerical solvers are at the heart of many virtual experiment types. Through the generalized framework, more than 40 solver plug-ins have been developed for the Tornado kernel. The solver plug-in framework is discussed in Chapter 8.
- Development of a mechanism for *a priori* **exploration of the solver setting space** (Claeys et al., 2006e). The mechanism allows for investigating the sensitivity of a model to solver settings using the same techniques that are normally used for determining the sensitivity of the model to changes to parameter values and/or dynamic input. The solver setting exploration framework is discussed in Chapter 8.
- Generation of **efficient and safe executable model code** through a model compiler back-end that can be used from two high-level modelling languages: MSL and Modelica (Claeys et al., 2006d, 2007a). The model compiler is capable of generating information in several formats, including an executable model format that was especially designed for use with Tornado. The generation of efficient and safe executable model code is discussed in Chapter 9.

- Definition and implementation of an extensive set of **programming interfaces** that allow for the use of the Tornado software kernel from custom applications. Interfaces for .NET languages such as C# and VB.NET, as well as for C, C++, Java, OpenMI and CLIPS have been developed. Chapter 10 presents a discussion of these developments.
- Development of a transparent framework for **distributed execution** of complex virtual experiments (Claeys et al., 2006c). This framework allows for the execution of virtual experiments on large-scale grid computing infrastructures (*e.g.*, the European EGEE grid), as well as on light-weight cluster middlewares such as Typhoon (Claeys et al., 2006a), which was developed in conjunction with the Tornado framework. Transparent distributed execution in the scope of Tornado is discussed in Chapter 14.

The work discussed is related to several other projects that are focused on the development of modelling and virtual experimentation frameworks. One of the most relevant frameworks in this respect is Modelica (Fritzson, 2004). The fact that the Tornado and Modelica frameworks are related is not surprising, since the Modelica language and MSL (the high-level modelling language that historically was the first to be adopted in the Tornado framework) have a common origin. This common origin is the 1993 ESPRIT Basic Research Working Group 8467 (Vangheluwe et al., 1996), which has investigated the future of modelling and simulation in Europe.

Other frameworks that are related to Tornado are TIME (Rahman et al., 2003), OMS (David et al., 2002), JAMS (Kralish and Krause, 2006) and ModCom (Hillyer et al., 2003). In these cases, the relationship is less technical than in the case of Modelica, but rather based on the fact that all are focused on the domain of environmental modelling. One may wonder why time and effort is spent on the concurrent development of multiple decision-support frameworks in the environmental domain. In this dissertation it is discussed why the development of Tornado is considered necessary. A more general discussion also involving frameworks such as TIME, OMC, JAMS and ModCom can be found in (Rizzoli et al., 2008).

Although the work discussed in this dissertation is multi-faceted and encompasses several sub-domains, it is **not** focused on the following:

- Development and/or improvement of numerical algorithms: Numerical algorithms adopted in Tornado have been obtained from various external sources (literature, on-line code repositories, *etc.*) and are mostly used in their original form. No changes to the mathematics of the externally acquired solver algorithms have been made, and no new algorithms have been developed.
- Development of virtual experimentation methodologies: The methodologies behind the virtual experiment types that have been implemented in Tornado are either universally known, or have been developed in related work. No new methodologies have been developed during the course of this work.

The work discussed in this dissertation is strongly related to other research that has been conducted at the Department of applied Mathematics, Biometrics and Process Control of Ghent University. The PhD dissertation entitled “*Multi-formalism modelling and simulation*” (Vangheluwe, 2000) has led to the design of the MSL language and the development of the first MSL model compiler. Also, the notion of virtual experiments as self-contained entities was a result of this work.

In the scope of the dissertation “*Optimal experimental design for calibration of bioprocess models: a validated software toolbox*” (De Pauw, 2005), the Tornado-I framework was extended in order to validate a number of concepts and methodologies related to optimal experimental design, applied to the calibration of bioprocess models. Some of the ideas resulting from this work have been picked up and fit into the frame of Tornado-II.

A great number of practical modelling and simulation cases have been successfully carried out with the aid of the Tornado kernel in its various incarnations. The following is an overview of the most relevant PhD studies that have benefited from Tornado (in chronological order)¹:

- *“Calibration, identifiability and optimal experimental design of activated sludge models”* (Petersen, 2000)
- *“Modelling for optimisation of biofilm wastewater treatment processes: a complexity compromise”* (Vanhooren, 2001)
- *“Immission based real-time control of the integrated urban wastewater system”* (Meirlaen, 2002)
- *“Dynamic integrated modelling of basic water quality and fate and effect of organic contaminants in rivers”* (Deksissa, 2004)
- *“Systematic calibration of activated sludge models”* (Sin, 2004)
- *“Coding error isolation in computational simulation models - With application to wastewater treatment systems”* (Graham, 2005)
- *“Modelling the activated sludge flocculation process: a population balance approach”* (Nopens, 2005)
- *“Performance of constructed treatment wetlands: model-based evaluation and impact of operation and maintenance”* (Rousseau, 2005)
- *“Modelling, simulation and optimization of autotrophic nitrogen removal processes”* (Van Hulle, 2005)
- *“Modelling and monitoring the anaerobic digestion process in view of optimisation and smooth operation of WWTP’s”* (Zaher, 2005)
- *“Probabilistic design and upgrade of wastewater treatment plants in the EU Water Framework Directive context”* (Benedetti, 2006)
- *“Monitoring and modelling the dynamic fate and behaviour of pesticides in river systems at catchment scale”* (Holvoet, 2006)
- *“Modelling methane oxidation in landfill cover soils using stable isotopes”* (Mahieu, 2006)
- *“Mathematical modelling of waste sludge digestion reactors”* (de Gracia, 2007)
- *“Predicting effect of chemicals on freshwater ecosystems: model development, validation and application”* (De Laender, 2007)
- *“A new methodology for the integrated modelling of WWTP’s”* (Grau, 2007)
- *“Characterisation and modelling of soluble microbial products in membrane bioreactors”* (Jiang, 2007)
- *“Modelling the sewer-treatment-urban river system in view of the EU Water Framework Directive”* (Solvi, 2007)

¹Most of the PhD’s mentioned in this list can be freely downloaded from the BIOMATH website - <http://biomath.ugent.be/publications/download>

All of these PhD studies have lead to several publications in international journals and/or conference proceedings; the availability of information on practical Tornado-related modelling and simulation studies in literature is therefore extensive. This dissertation consequently focuses entirely on the design and implementation of the Tornado kernel, rather than its application to practical cases.

Most information in this dissertation relates to Tornado-II v0.32, which was released on January 15th, 2008 and consists of approximately 470,000 lines of source code.

Contributions

In this document, various incarnations of the Tornado framework and its related software components are discussed. Several people, including the author, have provided contributions to these components in terms of source code. In order to clarify the efforts realized by the author with respect to the efforts realized by others, an overview of the most relevant source code contributions to Tornado and its related software components is given below:

Kernels

The Tornado-I kernel's modelling and experimentation environments were mainly developed by the author, except for one major component: the MSL language and its model compiler. The MSL language was designed by Hans Vangheluwe in the scope of his PhD (Vangheluwe, 2000). The model compiler was developed by Bhama Sridharan and Hans Vangheluwe (both BIOMATH, UGent). A minor contribution to the Tornado-I kernel was also provided by Sebastiaan Kops (BIOMATH, UGent) through the development of a number of wrappers for numerical algorithms taken from (Press et al., 1992).

The Tornado-II kernel's modelling environment (including the MOF2T model compiler and an improved version of the MSL model compiler), its experimentation environment and all API's were also mainly developed by the author. Exceptions are the ExpCI experiment type for which the code was largely taken from (De Pauw, 2005), and the ExpSens experiment type that was re-developed by the author based on ideas from (De Pauw, 2005). The GA and SA optimization solvers were developed by Dirk De Pauw (BIOMATH, UGent), and the Dormand-Prince integration solvers were implemented by Cyril Garneau (modelEAU, ULaval, Quebec). Minor contributions were also realized by Karel Van Laer, Petra Claeys and Michael Rademaker (all BIOMATH, UGent), who respectively provided help with the development of the JNI and OpenMI Tornado API's, the generation of XML representations of equations in the MSL compiler, and the Common Library.

The Typhoon kernel was initially developed by Maria Chtepen (INTEC, UGent) in the scope of her Master's thesis (Chtepen, 2004) under the supervision of the author, and was later stabilized and improved further by the author. The gridification of Tornado experiments and the integration of Tornado with gLite was realized by the author.

Applications

The WEST-1 application was initially developed by Bart De Rauw (BIOMATH, UGent) and Dirk Stevens (HEMMIS N.V., Kortrijk). Minor contributions were also provided by Jonathan Levine (BIOMATH, UGent) and the author. The WEST-2 and WEST-3 applications were developed by Stefan De Grande (formerly HEMMIS N.V., now MOSTforWATER N.V., Kortrijk). At the moment of writing, the WEST-4 application is being developed by Stefan De Grande as well.

As discussed in Chapter 12, also a number of research applications were developed on top of the Tornado-I and Tornado-II kernels. The WEST++ application was initially developed by the author. Afterwards, a module for sensitivity analysis was added by Dirk De Pauw (BIOMATH, UGent). The EAST (De Pauw, 2005) and MORE (Sin et al., 2007a,b) applications were fully developed by Dirk De Pauw.

Part I

Concept Exploration and Requirements

2

State-of-the-Art

2.1 Modelling and Virtual Experimentation Concepts

2.1.1 General

In most projects studying ill-defined systems (*i.e.*, systems with respect to which some *prima facie* appropriate theoretical description is not suitable, because it treats the system as being more well-defined than it really is (Boden, 1984)), researchers have very different scientific backgrounds. Particularly in the field of environmental sciences, this is very apparent. Biologists and ecologists need to collaborate with *e.g.*, mathematicians, statisticians and computer scientists. This diversity has obvious advantages, but may also lead to problems because of misunderstandings due to the use of different terminology. This section therefore focuses on a number of terms and concepts that are of importance in the scope of the work described in this dissertation. Most definitions were taken from (Zeigler, 1976), (Kops et al., 1999) and (Fritzson, 2004). Another overview of terminology and methodology, specifically with respect to water quality management, can be found in (Carstensen et al., 1997).

- An **object** is some real-world entity. It can exhibit widely varying behavior depending on the context in which it is studied, as well as the aspects of its behavior that are under study.
- A **base model** is a hypothetical, abstract representation of the object's properties (in particular, its behavior), which is valid in all possible contexts, and describes all the object's facets. A base model is hypothetical as, in practice, we will never be able to construct/represent such a holistic model.
- A **system** is a well-defined object in the real world under specific conditions, only considering specific aspects of its structure and behavior.
- A **model** is an abstract representation of a system that allows for investigation of the properties of the system and, in some cases, prediction of future outcomes.
- An **experiment** is a test under controlled conditions that is made on a system to demonstrate a known truth, examine the validity of a hypothesis, or determine the efficacy or outcome of something previously untried.

- A **virtual experiment** is the virtual-world counterpart of a real-world experiment. It is therefore not applied to a real-world system, but to a model. The most obvious type of virtual experiment is a dynamic simulation. However, in our interpretation, virtual experiments are defined as any procedure in which the evaluation of a model is required. There are several possible motivations for performing virtual experiments rather than real-world experiments. Mostly these motivations are related to the following:
 - Real-world experiments may be **expensive** in terms of financial cost, *e.g.*, investigating ship durability by building ships and letting them collide is a very expensive method of gathering information.
 - Real-world experiments may be **dangerous**, *e.g.*, training nuclear plant operators in handling dangerous situations by letting the nuclear reactor enter hazardous states is not advisable.
 - The real-world system needed for the experiment may **not yet exist** because it is still to be designed and/or manufactured.
 - The **time scale** of real-world systems may be inappropriate for observation. Changes may either occur too fast or too slowly to be monitored.
 - Variables of real-world systems may be **inaccessible**.
- The **experimental frame** describes the context in which a system is studied. Typically, it will describe input providers and output acceptors.
- **Input providers** are generators that describe the stimuli applied to the system.
- **Output acceptors** are transducers that describe the transformations that are applied to the system outputs for meaningful interpretation.

Figure 2.1 gives an overview of the relationships between the concepts described above.

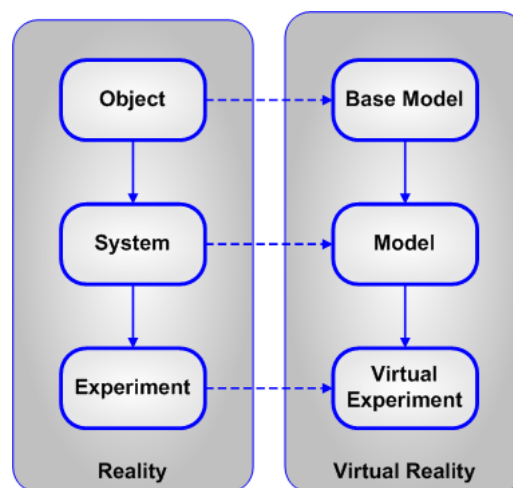


Figure 2.1: Modelling and Virtual Experimentation Concepts

Different types of models are distinguished, depending on how the model is represented:

- A **mental** model is an explanation in someone's thought process for how something works in the real world. It is an internal symbol or representation of external reality, hypothesized to play a major role in cognition and decision-making. Once formed, mental models may replace carefully considered analysis as a means of conserving time and energy. A simple example is the mental

model of a wild animal as dangerous: upon encountering a raccoon or a snake, one who holds this model will likely retreat from the animal as if by reflex.

- A **verbal** model is expressed in words. Expert systems¹ are a technology for formalizing verbal models.
- A **physical** model is a physical object that mimics some properties of a real-world system, to help answer questions about that system.
- A **mathematical** model is a description of a system where the relationships between variables of the system are expressed in mathematical form. Variables can be measurable quantities such as size, weight, temperature, *etc.* Sometimes the term **physical modelling** is used also for building computerized mathematical models of physical systems if the structuring and synthesis process is the same as when building real physical models. A better term however is **mechanistic modelling** (also known as **first principles modelling**).

A *plethora* of modelling formalisms have been developed over the years, each aimed at representing a particular type of system in the most convenient and concise manner. Typically, systems are classified on the basis of the way the concept of time is looked at:

- A **continuous system** (or continuous-time system) is a system whose inputs, outputs and state are capable of changing at every instant of time.
- A **discrete system** (or discrete-time system) is a system whose inputs, outputs and state only change at discrete (equidistant or non-equidistant) time points.
- A **hybrid system** is a system that has continuous as well as discrete components, *i.e.*, it has phenomena that are capable of leading to changes at every instant of time, as well as phenomena that only lead to changes at discrete time points.

For each of these three categories, various modelling formalisms exist. In the case of discrete systems, one for example distinguishes Finite State Automata (FSA), Petri Nets, Statecharts and event-based formalisms. Event-based formalisms can further be sub-divided according to the way events are grouped. In event-scheduling, events are described individually. In activity-scanning, start and stop events of activities are described jointly. Finally, process-interaction describes entire processes that can give rise to multiple types of events.

In continuous system modelling, formalisms are often based on mathematical equations, hence the term equation-based modelling. However, other continuous system modelling formalisms exist such as Transfer Functions, Bond Graphs and Forrester System Dynamics. It must be noted though that many of these (*e.g.*, the two last examples) are mostly graphical formalisms and that the semantics of the systems they represent can also be expressed as equations.

In environmental science, continuous aspects of systems are usually studied, and for complex systems traditional, equation-based approaches are typically most convenient. Consequently, focus of this dissertation is on continuous systems, modelled on the basis of equation-based approaches.

The following section presents some terminology and formal representations that are useful when discussing equation-based continuous system modelling.

¹http://en.wikipedia.org/wiki/Expert_system

2.1.2 Equation-based Mathematical Model Representation

Let x be an unknown function $x : \mathbb{R} \rightarrow \mathbb{R}$ in t with $\frac{d^i x}{dt^i}$ the i -th derivative of x , then (2.1) is called an **Ordinary Differential Equation** (ODE) of order n .

$$\frac{d^n x}{dt^n} = f\left(\frac{d^{n-1} x}{dt^{n-1}}, \frac{d^{n-2} x}{dt^{n-2}}, \dots, x, t\right) \quad (2.1)$$

For vector-valued functions $X : \mathbb{R} \rightarrow \mathbb{R}^m$, we call F a system of ordinary differential equations of dimension m as in (2.2).

$$\frac{d^n X}{dt^n} = F\left(\frac{d^{n-1} X}{dt^{n-1}}, \frac{d^{n-2} X}{dt^{n-2}}, \dots, X, t\right) \quad (2.2)$$

When a differential equation of order n has a form as represented in (2.3), it is called an **implicit** differential equation, whereas (2.2) is an **explicit** differential equation.

$$0 = F\left(\frac{d^n X}{dt^n}, \frac{d^{n-1} X}{dt^{n-1}}, \dots, X, t\right) \quad (2.3)$$

The value of X_i at $t = t_0$, with t_0 the lower bound of the time interval under consideration, is called **initial value** or initial condition. The value of X_i at $t = t_f$, with t_f the upper bound of the time interval, is called **final value**.

Any ODE of order n can be easily converted to a set of n first order ODE's through the introduction of auxiliary variables X_0, \dots, X_n as in (2.4).

$$\begin{cases} X_0 = X \\ X_1 = \frac{dX_0}{dt} \\ X_2 = \frac{dX_1}{dt} \\ \vdots \\ X_n = \frac{dX_{n-1}}{dt} = F(X_{n-1}, X_{n-2}, \dots, X_1, X_0, t) \end{cases} \quad (2.4)$$

In the field of modelling and simulation, differential equations are often written in the form (2.5) where X are referred to as **state variables** or **derived variables**, Θ as **parameters**, U as **input variables** and W as **worker variables**. (2.5) is then referred to as **state equations**. By definition, parameters do not change over time. Input variables are time-dependent and are supplied by the experimental frame that surrounds the model. Finally, worker variables are the left-hand sides of algebraic equations as in (2.6) that can be used to simplify the representation of differential equations or to express algebraic constraints.

$$\frac{dX}{dt} = F_X(X, \Theta, U, W, t) \quad (2.5)$$

$$W = F_W(X, \Theta, U, t) \quad (2.6)$$

Next to state and worker equations, one also distinguishes **output equations**, as in (2.7). Output equations are algebraic equations but, in contrast to worker equations, do not contribute to the state of a system. The variable at the left-hand side of an output equation is an **output variable**.

$$Y = F_Y(X, \Theta, U, W, t) \quad (2.7)$$

State, worker and output equations are all time-dependent. It is however customary to also distinguish **initial** and **final equations** (also known as terminal equations). These are algebraic equations that are independent of time, in the sense that initial equations are only relevant at $t = t_0$ whereas final equations are only relevant at $t = t_f$, as shown in (2.8) and (2.9).

$$\begin{aligned}\Theta &= F_{0,\Theta}(\hat{\Theta}, U(t_0)) \\ W(t_0) &= F_{0,W}(\Theta, U(t_0)) \\ X(t_0) &= F_{0,X}(\Theta, U(t_0), W(t_0))\end{aligned}\tag{2.8}$$

$$Y(t_f) = F_{f,Y}(X(t_f), \Theta, U(t_f), W(t_f))\tag{2.9}$$

As an example, consider the two-step model for fat crystallization in chocolate due to (Foubert et al., 2005) that is given in (2.10). In this model, h_1 and h_2 are state variables, r_1 and r_2 are worker variables and f_1 and f_2 are output variables. The set of parameters consists of a_1 , a_2 , b , K , n and $tind$. The initial value of h_2 is computed through equation (2.11). Since this model is presented here solely for the purpose of illustrating various types of equations, the biological meaning of the various parameters and variables is not given.

$$\begin{cases} \frac{dh_1}{dt} = r_1 - r_2 \\ \frac{dh_2}{dt} = r_2 \\ r_1 = -bh_1 \\ r_2 = K(h_2^n - h_2) \\ f_1 = a_1 - a_1h_1 \\ f_2 = a_2 - a_2h_2 \end{cases}\tag{2.10}$$

$$h_2(t_0) = \left(1 + \frac{0.99^{1-n} - 1}{e^{(n-1) \times K \times tind}}\right)^{\frac{1}{1-n}}\tag{2.11}$$

ODE's are not the only type of differential equations. Several other types can be distinguished. A short overview of the most important types follows:

- An **ordinary differential equation** (ODE) is a differential equation in which the unknown variable is a function of a single independent variable.
- A **partial differential equation** (PDE) is a differential equation in which the unknown function is a function of multiple independent variables and their partial derivatives.
- A **delay differential equation** (DDE) is a differential equation in which the derivative of the unknown function at a certain time is given in terms of the values of the function at previous times.
- A **stochastic differential equation** (SDE) is a differential equation in which one or more of the terms is a stochastic process, thus resulting in a solution which is itself a stochastic process.
- A **differential algebraic equation** (DAE) is a differential equation comprising differential and algebraic terms, given in implicit form.

Each of the above categories can be subdivided into **linear** and **non-linear** differential equations. A differential equation is linear if the dependent variables and all their derivatives appear to the power of one, and there are no products or functions of the dependent variables. Otherwise the differential equation is non-linear.

Another term that is often used in the field of modelling and simulation is **variability**. An item is said to be of variability 0 in case it is always constant, *i.e.*, its value never changes. An item is of variability

1 in case it is constant during simulations but can receive a different value in between simulations. Parameters are by definition of variability 1. Items for which values are free to change during and in between simulations are of variability 2, except in case of a hybrid language, where discrete-time variability is of level 2 and continuous-time variability is of level 3.

2.1.3 Coupled Model Representation

A common means to tackle complexity is to decompose a problem **top-down** into smaller sub-problems. Conversely, complex solutions may be built **bottom-up** by combining primitive sub-problem building blocks. Both approaches are instances of **compositional modelling**: the connection (or rather, composition) of interacting **component** models to build new models. In case the components only interact via their interfaces, and do not influence each other's internal workings in any other way, the compositional modelling approach is called **modular**, otherwise it is called **non-modular**.

Models that are constructed from *sub-models* in a mechanistic way through composition are called **coupled models**. As any other model, coupled models have their own set of inputs and outputs (and possibly also parameters). The graph that is a result of the inter sub-model coupling, and the coupling between sub-models and interfaces of the coupled model, is typically represented in a graphical way. This graph is usually called connection graph, configuration or **layout**. The latter is the term that will be used throughout the sequel. Sub-models are also sometimes referred to as nodes (since they are constituents of a graph) or **unit processes**.

2.1.4 Petersen Matrix Representation

A clear presentation of dynamic biochemical processes is very important to facilitate users of a software program to obtain an overview of the interactions between the components of a system. The method of presentation used in tools such as Aquasim and WEST was made popular for technical biochemical systems by the report of the IAWQ task group on mathematical modelling for design and operation of biological wastewater treatment (Henze et al., 1986). It is based on work on chemical reaction engineering (Petersen, 1965), and is therefore referred to as Petersen matrix.

Dynamic processes describe transformations by their contribution to the temporal rate of change of (dynamic) state variables. Usually, a biological or chemical process transforms several substances in fixed stoichiometric proportions. Therefore, it is advantageous to separate a common factor as a process rate, and to describe a process by this rate and by stoichiometric coefficients for all substances involved in the process. The contribution of a process to the temporal change of the concentration of a substance is then given as the product of the common process rate and the substance-specific stoichiometric coefficient. This decomposition of process rates into a common process rate and individual stoichiometric coefficients is not unique; to make it unique, one of the stoichiometric coefficients is therefore usually set to unity. With this concept, the total transformation rate of a substance s_j is given by

$$r_j = \sum_{i=1}^n \nu_{i,j} \rho_i \quad j = 1, \dots, m$$

where r_j is the total transformation rate of the substance s_j , $\nu_{i,j}$ is the stoichiometric coefficient of the substance s_j for the process p_i and ρ_i is the rate of the process p_i . A clear presentation of the process model is given by writing the stoichiometric matrix $(\nu_{i,j})$, supplemented by the process rates ρ_i in an additional column. This results in a process matrix as shown in Table 2.1. The non-zero elements of a row of such a matrix show which substances are affected by a given process, whereas the nonzero elements of a column indicate which processes have an influence on a given substance. It is a useful convention to use positive process rates. In this case the signs of the stoichiometric coefficients indicate consumption (-) or production (+) of the corresponding substance.

Table 2.1: Petersen Matrix Representation of a Process Model

Process	Stoichiometry				Kinetics
	s_1	s_2	\dots	s_m	
p_1	$\nu_{1,1}$	$\nu_{1,1}$	\dots	$\nu_{1,m}$	ρ_1
p_2	$\nu_{2,1}$	$\nu_{2,1}$	\dots	$\nu_{2,m}$	ρ_1
\vdots	\vdots	\vdots	\ddots	\vdots	\vdots
p_n	$\nu_{n,1}$	$\nu_{n,1}$	\dots	$\nu_{n,m}$	ρ_n

Table 2.2: Bacterial Growth in an Aerobic Environment

Process	Stoichiometry			Kinetics
	Biomass X_B	Substrate S_S	Oxygen S_O	
Growth	1	$-\frac{1}{Y}$	$-\frac{1-Y}{Y}$	$\frac{\hat{\mu}S_S}{K_S+S_S}X_B$
Decay	-1		-1	bX_B

An example is given in Table 2.2. From this table follows that the total transformation rate for biomass X_B is given by (2.12), the total transformation rate for soluble substrate S_S is given by (2.13) and the total transformation rate for oxygen S_O is given by (2.14).

$$r_{X_B} = \frac{\hat{\mu}S_S}{K_S + S_S}X_B - bX_B \quad (2.12)$$

$$r_{S_S} = -\frac{1}{Y} \frac{\hat{\mu}S_S}{K_S + S_S}X_B \quad (2.13)$$

$$r_{S_O} = -\frac{1-Y}{Y} \frac{\hat{\mu}S_S}{K_S + S_S}X_B - bX_B \quad (2.14)$$

2.2 Complex Environmental Systems - Case: Water Quality Management

In water quality research, the biological and/or chemical quality of water in rivers, sewers and wastewater treatment plants (WWTP) is studied. Research in this domain is facilitated by a number of models that have received a formal or *de facto* standardization status. Most notable are River Water Quality Model No.1 (RWQM1) (Reichert et al., 2001), the Activated Sludge Model (ASM) series (Henze et al., 2000), and Anaerobic Digestion Model No.1 (ADM1) (Batstone et al., 2002).

Because of the large number of universally accepted methodologies and standardized models, the water quality domain can be considered mature, in contrast to other sub-domains of environmental science that have not yet reached this level of standardization.

The current maturity of the water quality domain is the result of an evolution that is depicted in Figure 2.2. At first, the situation could be characterized as chaotic: no clear distinction was made between sub-domains and no standardized unit process models existed. During this era, the further advancement of research was inhibited by the lack of a *lingua franca*. Afterwards, separate research communities started to form around sewer networks, wastewater treatment plants and river systems. As a result of the work done by these communities, models for unit processes were standardized. Subsequently, focus shifted to the use of these unit processes for the comprehensive modelling of treatment plants, sewer

Table 2.3: Standard WWTP Unit Process Models

Model	Year	Purpose	#Processes	#Components	#Parameters
'ASM0'	1987	C	2	3	4
ASM1	1987	C, N	9	13	19
ASM2	1994	C, N, P	20	19	65
ASM2d	1999	C, N, P	22	19	67
ASM3	1999	C, N	13	13	36
ASM3P	2001	C, N, P	24	17	71
ADM1	2002	X	28	36	96
ASDM	2007	C, N, P, X	51	62	470

networks and river systems. Finally, the boundaries between sub-domains were broken down and large-scale integrated models encompassing treatment plants, sewer networks and river systems were built.

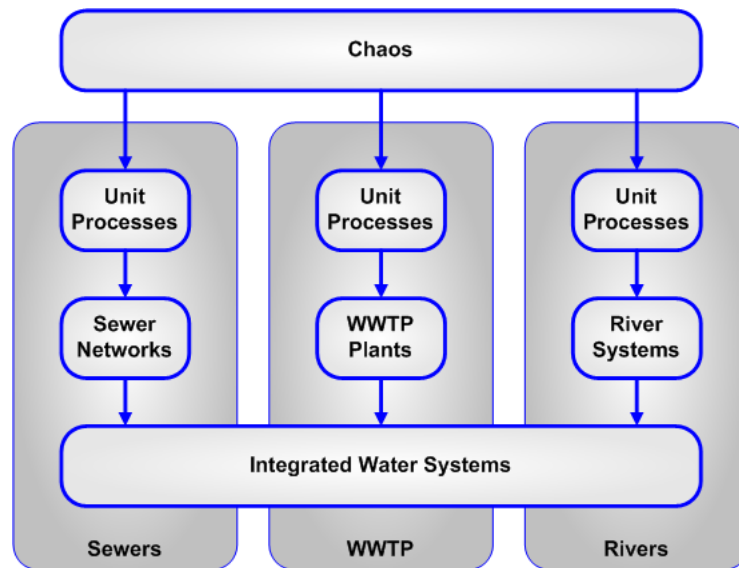


Figure 2.2: Evolution of the Level of Integration in Water Quality Management

Table 2.3 gives an overview of the standard unit process models that are in use today to describe activated sludge and sludge digestion processes. For each model, the year of standardization, its purpose (C = COD removal; N = nitrogen removal; P = phosphorus removal; X = sludge digestion), the number of processes, the number of components and the number of parameters are given. All models except for the first and the last are standardized by the IWA (International Water Association). The first model is only given from the point of view of completeness; it is not used in practice. The last model is a proprietary model that is implemented in the BioWin program (to be discussed later in this chapter).

Water quality models typically consist of large sets of non-linear Ordinary Differential Equations (ODE) and non-linear Algebraic Equations (AE). These equations are mostly relatively well-behaved, although discontinuities do occur. The complexity of water quality models is therefore not in the nature of the equations, but in the sheer number. In WWTP, smaller coupled models such as variants of the well-known Benchmark Simulation Model No.1 (BSM1) (Copp, 2002) consist of approximately 150 derived variables. Larger systems have up to 1,000 derived variables and over 10,000 (partly coupled) parameters. On a typical workstation, a simulation run usually lasts minutes to hours.

Table 2.4 gives an overview of the complexities of a number of typical coupled water quality models,

Table 2.4: Complexity of a Number of Selected Water Quality Models

Model	#Parameters	#Derived Variables	#Algebraic Variables
ASU	652	20	620
BSM1_OL	1,009	178	1,162
BSM1_CL	1,052	180	1,321
Galindo_OL	5,641	270	3,352
Galindo_CL	5,740	275	3,700
Orbal	6,588	260	3,325
Nil	8,484	256	5,541
Lux	11,806	414	4,408
VLWWTP	10,555	579	7,178
IUWS	16,167	1,663	18,353

which were taken from the Tornado testbench. The Tornado testbench is an extensive collection of models and virtual experiments that was established over a longer period of time and that is used to verify the behavior of the Tornado kernel, and to investigate the sources of complexity in water quality management and other domains. For each selected case, Table 2.4 lists the number of parameters, the number of derived variables and the number of algebraic variables. It must be noted that many parameters in these models are actually coupled (approximately 5 to 10%), the true number of degrees of freedom with regard to parameters is therefore less than the figures in the table indicate. The number of derived variables corresponds to the number of ODE's; the number of algebraic variables corresponds to the sum of the number of algebraic worker equations (which influence the state of the system) and the number of algebraic output equations (which have no effect on the state of the system).

The models that are given in the table are the following:

- **ASU**: In this simple model, a WWTP with alternating aeration (30 minute switch) is studied. The activated sludge basin was modelled with one completely mixed reactor.
- **BSM1_OL** (Copp, 2002): The Benchmark Simulation Model No.1 is a simulation environment defining a plant layout, a simulation model, influent loads, test procedures and evaluation criteria. For each of these items, compromises were pursued to combine plainness with realism and accepted standards. Once the user has validated the simulation code, any control strategy can be applied and the performance can be evaluated according to certain criteria. The plant layout is modelled using 5 activated sludge tanks in series, 2 of which are anoxic, and a 10-layer settling tank. It represents a 100,000 PE (Person Equivalent) nitrogen removal WWTP.
- **BSM1_CL** (Copp, 2002): In this case, a basic control strategy is proposed to test the benchmark: its aim is to control the dissolved oxygen level in the final compartment of the reactor by manipulation of the oxygen transfer coefficient, and to control the nitrate level in the last anoxic compartment by manipulation of the internal recycle flow rate.
- **Galindo_OL** (Ayessa et al., 2006): The Galindo WWTP (Bilbao, Spain) is designed for carbon and nitrogen removal and handles a daily average wastewater flow of 345,600 m³/day. The WWTP is divided into six identical lines and the configuration in each of them can be operated as one of two alternative configurations: Regeneration-Denitrification-Nitrification (RDN) and Denitrification-Regeneration-Denitrification-Nitrification (DRDN). In the open loop configuration, the Galindo WWTP was modelled without any control strategy. In the model, the aeration in the aerated tanks, the flow rate going to the regeneration zone, the internal recycle, the waste flow rate and the temperature are implemented as parameters that have to be set by the user.

- **Galindo_CL** (Ayesa et al., 2006): In the closed loop version of the Galindo model, influent data for a whole year are used, as it is important to consider the temperature variations for such a long period. Next to the temperature, also the internal and external recycles and the waste flow rate vary dynamically. Although the waste flow rate reaches very low values during some periods, the TSS in the DN does not increase very much. The reason is that the Galindo WWTP is oversized and with low TSS values the effluent quality is still guaranteed.
- **Orbal** (Insel et al., 2003): This model of an Orbal plant (located in Athens, GA, USA) treating 50,000 PE and achieving biological nutrient removal, was modelled using nitrate, oxygen, ammonium, nitrogen and phosphate measurements for calibration. An Orbal plant is a type of extended aeration activated sludge plant, which claims to achieve simultaneous nitrification and de-nitrification in a single reactor, offering reduced costs. Generally, three or four channels are recommended for design.
- **Nil** (De Schepper et al., 2006): This model was used to simulate the concentrations of the pesticides diuron and chloridazon in the river Nil, a small stream flowing in a rural area in Belgium. In order to do so, the existing RWQM1 model was modified with processes determining the fate of non-volatile pesticides in the water phase and sediments. The exchange of pesticides between the water column and the sediment is described by diffusion, sedimentation and resuspension. Burial of sediments is also included.
- **Lux** (Solvi et al., 2006): For this model, the RWQM1 model was simplified (no grazers and no pH calculation) to model the nutrients and oxygen dynamics of a stretch of the river Sure in Luxembourg and of two tributaries, the Alzette and the Attert.
- **VLWWTP**: This model represents a very large wastewater treatment plant. It was dimensioned for 220,000 PE but is actually used for 280,000 PE. It consists of 4 streets with 4 tanks each (1 anaerobic, 1 anoxic and 2 aerobic), and has substantial facilities for primary and secondary clarification.
- **IUWS** (Integrated Urban Water System) (Solvi et al., 2005, 2006): This is a model that integrates the Lux model, the Bleesbruck WWTP (modelled with ASM2d) and its draining sewer system (modelled with a modified implementation of the KOSIM model), including CSO's (Combined Sewer Overflows and structures). It was used to evaluate the effect on river water quality of measures taken in the catchment, sewer, WWTP and river itself.

2.3 Existing Generic Frameworks

2.3.1 Introduction

Since the advent of digital computers in the 1960's, there has been a continuous string of developments in the area of software tools for modelling and simulation. In fact, the variety of tools available at this moment is such that a rigorous classification is nearly impossible. Indeed, a *plethora* of environments, frameworks, applications and libraries exist, with overlapping features and functionalities, but typically with a different focus in terms of performance, flexibility and intended audience.

One possible classification of software tools for modelling and simulation is the one that uses genericity as a criterion. In many disciplines, one will find that both generic tools as well as specialized, domain-specific tools are used to solve modelling and simulation problems. The field of water quality management is no exception to this rule. This section therefore contains an overview of a number of tools that currently are (or could be) used in this field. The generic tools that will be discussed are ACSL, MATLAB/Simulink, Mathematica, Mathcad, Maple, LabVIEW and Modelica. From these,

MATLAB/Simulink is without a doubt the most popular amongst researchers and practitioners in the field of water quality management. Although all of the tools that are discussed have very different origins, they have converged over the years. As a result each of them is currently able to perform continuous system modelling and simulation on the basis of differential equations.

Next to the afore-mentioned generic tools, a number of domain-specific tools for water quality management will also be discussed: Aquasim, BioWin, GPS-X, Simba, STOAT and WEST. It is unclear which of these is currently the most popular overall. BioWin for instance is widely used amongst practitioners in North-America, but does not have a strong following in Europe. WEST is popular amongst researchers world-wide, but is only scarcely used amongst practitioners.

2.3.2 ACSL

The Advanced Continuous Simulation Language (usually abbreviated as ACSL and pronounced as “axle”) (Mitchell and Gauthier, 1976) is a computer language designed for modelling and evaluating the performance of continuous systems described by time-dependent, non-linear differential equations. It is a dialect of the Continuous System Simulation Language (CSSL), which was originally designed by the Simulations Council Inc. (SCI) in 1967 in an attempt to unify a number of other languages that existed at the time. Other versions of CSSL include HYTRAN, SL-I, S/360 and CSMP. ACSL however is the only CSSL dialect that has been able to maintain a substantial following until now.

ACSL is an equation-oriented language consisting of a set of arithmetic operators, standardized functions, a set of special ACSL statements, and macro capability. The macro capability allows for extension of the set of special ACSL statements. ACSL models are translated to executable code in general-purpose languages such as FORTRAN or C. The translators are usually fairly simple since there is a strict relationship between ACSL statements and general-purpose language code fragments. An important feature of ACSL however is its sorting of the continuous model equations, in contrast to models in general-purpose languages where program execution depends on the order of statements.

Typical areas in which ACSL is adopted include control system design, aerospace simulation, chemical process dynamics, power plant dynamics, plant and animal growth, toxicology models, vehicle handling, microprocessor controllers, and robotics.

Although ACSL is largely a language from the past, there are still commercial ACSL implementations that are believed to receive active development and support. Examples include the acslX² family of products (acslXtreme, acslXpress and OPTIMUM) by AEgis Technologies Group, Inc.³ (Huntsville, AL, USA) and MMS (Modular Modelling System), by nHance Technologies, Inc.⁴ (Lynchburg, VA, USA)

2.3.3 MATLAB/Simulink

MATLAB is a numerical computing environment and programming language. Created by The MathWorks⁵ (Natick, MA, USA), MATLAB allows easy matrix manipulation, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs in other languages. Although it specializes in numerical computing, an optional toolbox interfaces with the Maple symbolic engine, allowing it to be part of a full computer algebra system. As of 2004, MATLAB has been in use by more than one million people in industry and academia.

Short for MATRix LABoratory, MATLAB was invented in the late 1970's by Cleve Moler, then chairman of the computer science department at the University of New Mexico. It was designed to give

²<http://www.acslx.com>

³<http://www.aegistg.com>

⁴<http://www.nhancetech.com>

⁵<http://www.mathworks.com>

students access to LINPACK⁶ and EISPACK⁷ without having to learn FORTRAN. It soon spread to other universities and found a strong audience within the applied mathematics community. Jack Little, an engineer, was exposed to it during a visit Moler made to Stanford University in 1983. Recognizing its commercial potential, he joined with Moler and Steve Bangert. They rewrote MATLAB in C and founded The MathWorks in 1984 to continue its development. MATLAB was first adopted by control design engineers, but quickly spread to many other domains. MATLAB is structured around the M language, which is a scripted procedural language. The M language can be used interactively from a command shell, or sequences of statements can be saved in a script file.

MATLAB is a proprietary product of The MathWorks, so users are subject to vendor lock-in. However, some other tools are partially compatible (such as GNU Octave⁸) or provide a simple migration path (such as Scilab⁹). The M language has a mixed heritage and as a consequence has a syntax that is unconventional in several ways (see for instance the MATLAB Wikipedia entry¹⁰ for a list of language quirks). Actually, from a programming language point of view, M is by no means the most advanced or most sound language, however it has become a *de facto* standard for scientific and technical computing, and is generally appreciated by its many users.

The MATLAB environment can be expanded with a vast number of toolboxes. One of these toolboxes is Simulink. It has a long history and is nearly always used in conjunction with MATLAB when modelling and simulation of non-trivial ODE's and AE's are concerned (it is also possible to model and simulate these systems in M directly, but as M is a scripted language, performance is prohibitively low for all but trivial systems).

Simulink uses the Causal Block Diagram (CBD) approach for modelling. A causal block diagram is a graph made up of connected operation blocks. The connections stand for signals, which are propagated from one block to the next. Blocks can be purely algebraic or may involve some notion of time such as delay, integration and derivation. In order to allow for an algorithm based on time slicing to solve a CBD, the order in which blocks need to be computed can be determined by abstracting the diagram to a dependency graph. The dependencies between signals on either side of a block that denotes some form of time delay do not show up in a dependency graph as these blocks relate signal values at different time instants. If a dependency graph does not contain dependency cycles, a simple topological sort¹¹ will give an order in which the blocks need to be evaluated to give correct simulation results. Note that there may be different equivalent (from the point of view of the correctness of simulation results) topological sort orderings corresponding to different orders in which neighbors of a node are visited.

For large-scale modelling efforts, the CBD approach becomes impractical, both from the point of view of readability and maintainability, as from the point of view of performance. One therefore typically relies on external code in a conventional programming language (with C linkage) that is compiled separately and used in a Simulink diagram as a single block. Although this approach allows for solving the performance issue to a large extent, it also provides for even less readability and maintainability, since part of the model is represented as a Simulink diagram, while other parts of the model are coded in a conventional programming language.

2.3.4 Mathematica

Mathematica is a generic computing environment, organizing many algorithmic, visualization, and user interface capabilities within a document-like user interface paradigm. It was originally conceived by

⁶<http://www.netlib.org/linpack>

⁷<http://www.netlib.org/eispack>

⁸<http://www.octave.org>

⁹<http://www.scilab.org>

¹⁰<http://en.wikipedia.org/wiki/MATLAB>

¹¹http://en.wikipedia.org/wiki/Topological_sorting

Stephen Wolfram, and developed by a team of mathematicians and programmers that he assembled and led. Mathematica is currently commercially available through Wolfram Research¹² (Long Hanborough, UK).

Since version 1.0 in 1988, Mathematica has steadily expanded into more and more generic computational capabilities. Besides addressing nearly every field of mathematics, it provides a multi-faceted language for data integration and cross-platform support for a wide range of tasks such as giving computationally interactive presentations, graphics editing, and symbolic user interface construction.

The default Mathematica front-end features extensive layout and graphical capabilities. It allows for the formatting of mathematical expressions, performs pretty-printing (automatic indentation) and provides Mathematica documents in a form called a notebook. In a notebook, user input (both text and Mathematica input) as well as results sent by the kernel (including graphics, sound and interactive interfaces) are placed in a hierarchy of cells which also allows for outlining and sectioning of a document. All notebook contents can be placed in-line within text regions or within input. Starting with version 3.0 of the software, notebooks are represented as expressions that can be created, modified or analyzed by the kernel.

Mathematica Player is a front-end provided by the manufacturer free of charge, which can read notebooks and perform most of the other functions of the licensed front-end. It includes a Mathematica kernel so that calculations can be updated in response to interactive elements. It does not, however, allow new documents to be created.

Communication with other applications occurs through a protocol called MathLink. It allows not only for communication between the Mathematica kernel and front-end, but also provides a general interface between the kernel and arbitrary applications. Wolfram Research freely distributes a developer kit for linking applications written in the C programming language to the Mathematica kernel through MathLink. Two other components of Mathematica, whose underlying protocol is MathLink, allow developers to establish communication between the kernel and a Java or .NET program: J/Link and .NET/Link.

Using J/Link, a Java program can ask Mathematica to perform computations; also, a Mathematica program can load any Java class, manipulate Java objects and perform method calls, making it possible, for instance, to build Java graphical user interfaces from Mathematica. Similarly, a .NET software can invoke the kernel to perform calculations and send results back, and Mathematica developers can easily have access to .NET's functionality. Communication with SQL databases is achieved through built-in support for JDBC (Java Data Base Connectivity). Lastly, Mathematica can install web services from a WSDL (Web Service Description Language) description.

In the first versions of Mathematica, performance for large numerical computations lagged behind specialized software, but recent versions saw great improvement in this area with the introduction of packed arrays in version 4 (1999) and sparse matrices in version 5 (2003).

Version 5.2 (2005) added automatic multi-threading when computations are performed on modern multi-core computers. This release included CPU-specific optimized libraries. In addition Mathematica is supported by third-party specialist acceleration hardware. In 2002 gridMathematica was introduced to allow user-level parallel programming on heterogeneous clusters and multiprocessor systems and supports grid technology such as the Microsoft Compute Cluster Server.

2.3.5 Mathcad

Mathcad (originally spelled as MathCAD) is desktop software for performing and documenting engineering and scientific calculations. First introduced in 1986 on DOS, it was the first to introduce live editing of typeset mathematical notation, combined with its automatic computation. It was also the first to automatically compute and check consistency of engineering units such as the International System

¹²<http://www.wolfram.com>

of units. Both are important contributions to software technology. Mathcad today includes some of the capabilities of a computer algebra system somewhat similar to Mathematica or Maple. Mathcad nonetheless remains oriented towards ease of use and numerical engineering applications. Mathcad was conceived and originally written by Allen Razdow (of MIT), co-founder of Mathsoft¹³ which is now part of Parametric Technology Corporation¹⁴ (Needham, MA, USA).

Mathcad paved the way for a variety of desktop mathematical tools. Noted software metrician Capers Jones' study of 500 programming languages and programmable tools in the 1990's found Mathcad and Microsoft Excel to be number one and two in productivity (function-points per line of code).

Among the capabilities of Mathcad are:

- Solving differential equations, with several possible numerical methods
- Graphing functions in two or three dimensions
- The use of the Greek alphabet (upper and lower case) in both text and equations
- Symbolic calculations
- Vector and matrix operations
- Symbolically solving systems of equations
- Curve-fitting
- Finding roots of polynomials and functions
- Statistical functions and probability distributions
- Finding eigenvalues and eigenvectors
- Calculations in which units are bound to quantities

Although this program is mostly oriented to the non-programming users, it is also used in more complex projects to visualize results of mathematical modelling using distributed computing and traditional programming languages. It is often used in large engineering projects where traceability and standards compliance are of importance.

2.3.6 LabVIEW

LabVIEW (short for Laboratory Virtual Instrumentation Engineering Workbench) is a platform and development environment for a visual programming language from National Instruments¹⁵ (Austin, TX, USA). Originally released for the Apple Macintosh in 1986, LabVIEW is commonly used for data acquisition, instrument control, and industrial automation on a variety of platforms including Microsoft Windows, various flavors of UNIX, Linux, and Mac OS.

The programming language used in LabVIEW is named G. It is a dataflow programming language where execution is determined by the structure of a graphical block diagram on which the programmer connects different function nodes by drawing wires. These wires propagate variables and any node can execute as soon as all its input data become available. Since this might be the case for multiple nodes simultaneously, G is inherently capable of parallel execution. Multi-processing and multi-threading

¹³<http://www.mathsoft.com>

¹⁴<http://www.ptc.com>

¹⁵<http://www.ni.com>

hardware is automatically exploited by the built-in scheduler, which multiplexes multiple OS threads over the nodes ready for execution.

LabVIEW ties the creation of user interfaces (called front panels) into the development cycle. LabVIEW programs/subroutines are called virtual instruments (VI's). Each VI has three components: a block diagram, a front panel and a connector pane. The latter may represent the VI as a sub-VI in block diagrams of calling VI's. Controls and indicators on the front panel allow an operator to input data into or extract data from a running virtual instrument. However, the front panel can also serve as a programmatic interface. Thus a virtual instrument can either be run as a program, with the front panel serving as a user interface, or, when dropped as a node onto the block diagram, the front panel defines the inputs and outputs for the given node through the connector pane. This implies each VI can be easily tested before being embedded as a subroutine into a larger program.

The graphical approach also allows non-programmers to build programs by simply dragging and dropping virtual representations of the lab equipment with which they are already familiar. The LabVIEW programming environment, with the included examples and the documentation, makes it simpler to create small applications. This is a benefit on one side, but there is also a certain danger of underestimating the expertise needed for good quality G programming. For complex algorithms or large-scale code it is important that the programmer possesses an extensive knowledge of the special LabVIEW syntax and the topology of its memory management. The most advanced LabVIEW development systems offer the possibility of building stand-alone applications. Furthermore, it is possible to create distributed applications which communicate by a client/server scheme, and thus is easier to implement due to the inherently parallel nature of G code.

2.3.7 Maple

Maple (Hutton, 1995) is a general-purpose mathematics software package that is specifically renown for its computer algebra (symbolic manipulation) capabilities. It was first developed in 1981 by the Symbolic Computation Group¹⁶ at the University of Waterloo in Ontario, Canada. Since 1988, it has been developed and sold commercially by Maplesoft¹⁷.

Maple combines a programming language with an interface that allows users to enter mathematics in traditional mathematical notation. Most of the mathematical functionality of Maple is written in the Maple language, which is interpreted by the Maple kernel. The Maple kernel is written in C. The Maple programming language is an interpreted, dynamically typed programming language. As is usual with computer algebra systems, symbolic expressions are stored in memory as directed acyclic graphs.

In 1989, the first graphical user interface for Maple was developed. Prior versions of Maple included only a command-line interface with two-dimensional output. X11 and Microsoft Windows versions of the new interface followed in 1990. In 2003 the current standard interface was introduced with Maple 9. This interface is primarily written in Java, however, portions such as the rules for typesetting mathematical formulae are written in the Maple language. The new interface is widely derided for being slow, and for this reason Maplesoft continues to include the previous "classic" interface while working to improve the performance and features of the new interface. In 2005 Maple 10 introduced a new "document mode", as part of the standard interface. The main feature of this mode is that math is entered using two-dimensional input, appearing similar to a formula in a book.

¹⁶<http://www.scg.uwaterloo.ca>

¹⁷<http://www.maplesoft.com>

2.3.8 Modelica

Modelica¹⁸ is similar to ACSL in the sense that it is a language definition rather than a specific tool or product. The language definition is currently at version 3.0 and has incrementally evolved from its initial version, which was an indirect consequence of the 1993 ESPRIT Basic Research Working Group on “Simulation for the Future: new Concepts, Tools and Applications” (Vangheluwe et al., 1996). In fact, the white paper produced by this working group stimulated the start (in 1996) of the design of a unified modelling language, which was eventually called Modelica.

Modelica is a declarative, object-oriented, equation-based language that unifies a number of other languages that existed at the time of its inception. It can be considered as one of the most powerful languages for declarative physical system modelling available to date. In Chapter 9, the Modelica language is discussed in further detail.

Several tools currently exist that, to a certain extent, support the Modelica language (in fact, there is no tool that supports the latest language definition in every detail). Below is an overview of the most important tools with respect to Modelica that are currently available:

- **Dymola:** Dymola from Dynasim AB¹⁹ (Lund, Sweden) has a Modelica translator which is able to perform all necessary symbolic transformations for large systems (> 100,000 equations) as well as for real-time applications. A graphical editor for model editing and browsing, as well as a simulation environment are included. Convenient interfaces to MATLAB and the popular block diagram simulator Simulink exist. For example, a Modelica model can be transformed into a Simulink S-function C MEX-file, which can be simulated in Simulink as an input/output block. Dymola typically applied to automotive systems, mechatronics and power systems.
- **MathModelica System Designer** from MathCore²⁰ (Linköping, Sweden) provides a Modelica modelling and simulation environment. It has an interactive graphical environment for easy model composition and an environment for simulation management. The professional edition, MathModelica System Designer Professional, includes a Mathematica link, offering facilities for mathematical computation, plotting, calculation, natural display of mathematical formulae, inline documentation, *etc.* within the powerful Mathematica notebook environment. MathCore also offers a basic graphical modelling environment, MathModelica Lite, for the Modelica open source compiler.
- **MOSILAB** from the Fraunhofer-Gesellschaft²¹ (Germany) is a newly developed Modelica simulator for complex technical systems. One innovative feature of MOSILAB is the mapping of state-dependent changes of the model structure during the simulation experiment. This enables, for example, simulation experiments with models of variable modelling depth or varying model physics.
- **SimulationX** from ITI²² (Dresden, Germany) is a graphically-interactive modelling and simulation tool. It provides ready-to-use model libraries, *e.g.*, for mechanics, multi-body systems, power transmission, hydraulics, pneumatics, thermodynamics and electric drives. A comprehensive API supports the integration into any CAE (Computer Aided Engineering), CFD (Computational Fluid Dynamics), CAD (Computer Aided Design) or database tool environment. A model can be exported as a Simulink S-function or C code for HiL (Hardware in the Loop) and RCP (Real-time Control Program) applications.

¹⁸<http://www.modelica.org>

¹⁹<http://www.dynasim.se>

²⁰<http://www.mathcore.com>

²¹<http://www.fraunhofer.de>

²²<http://www.iti.de>

- **OpenModelica** is an open source project at Linköping University (Sweden). The goal of the project is to create a complete Modelica modelling, compilation and simulation environment based on free software, distributed in source code form and intended for research purposes. OpenModelica consists of the following components:
 - OpenModelica Compiler (OMC): Command-line model compiler that converts high-level Modelica code to executable model code
 - OpenModelica Shell (OMShell): Shell that allows for interactively running Modelica scripting commands
 - OpenModelica Notebook (OMNotebook): Viewer and editor for Mathematica-like notebooks
 - OpenModelica Eclipse Plug-in (MDT): Plug-in for the Eclipse²³ framework that provides for a Modelica model editor with syntax highlighting and debugging facilities
 - OpenModelica Development Environment (OMDev): Environment for further development of the OpenModelica framework, including a compiler for the meta-Modelica language (Pop and Fritzson, 2006)
- **Modelicac** is a compiler for a subset of the Modelica language. Modelicac is included into the Scilab²⁴ distribution (although it is an independent executable) and is used by Scicos²⁵ (Scilab's block-oriented editor and simulator) to handle hybrid model simulations.

2.4 Existing Water Quality Tools

2.4.1 Aquasim

Aquasim²⁶ (Reichert, 1994; Reichert et al., 1995) was developed by Peter Reichert at EAWAG, *i.e.*, the Swiss Federal Institute of Aquatic Science and Technology²⁷ in the early 1990's. It is mainly intended for academic use and has not been updated since 2001. Implementation was done in C++. Three versions of the program are available: the window interface version has a graphical user interface, the character interface version can be run on a primitive teletype terminal, and the batch version is designed for long calculations to be submitted as batch jobs. The character interface and batch versions are available for several platforms, *i.e.*, SUN Solaris, IBM AIX, HP-UX, VMS, DEC Unix, Linux and Microsoft Windows. The window interface version is only available for Microsoft Windows as it is based on Microsoft's MFC framework.

Aquasim does not allow for layouts to be created in a graphical way, nor does it use a high-level modelling language to define atomic models. Instead, Aquasim allows its users to define the spatial configuration of the system under study as a set of compartments, which can be connected to each other by links. The available compartment types include mixed reactors, biofilm reactors (consisting of a biofilm and a bulk fluid phase), advective-diffusive reactors (plug-flow reactors with or without dispersion), saturated soil columns (with sorption and pore volume exchange), river sections (describing water flow and substance transport and transformation in open channels) and lakes (describing stratification and substance transport and transformation in the water column of the lake and in adjacent sediment layers). Compartments can be connected by two types of links. Advective links represent water flow and advective substance transport between compartments, including bifurcations and junctions. Diffusive links

²³<http://www.eclipse.org>

²⁴<http://www.scilab.org>

²⁵<http://www.scicos.org>

²⁶<http://www.aquasim.eawag.ch>

²⁷<http://www.eawag.ch>

represent boundary layers or membranes, which can be penetrated selectively by certain substances. The user of the program is free in specifying any set of state variables and transformation processes to be active within the compartments. For the model as defined by the user, the program is able to perform simulations, sensitivity analyses, uncertainty analyses (using linear error propagation) and parameter estimations using measured data.

Aquasim uses the Petersen matrix notation to describe processes and supports a non-standard set of variable types:

- *State Variables* represent concentrations or other properties to be determined by a model according to user-selected transport and user-defined transformation processes
- *Program Variables* make quantities such as time, space coordinates, discharge, *etc.* that are used for model formulation, available as variables.
- *Constant Variables* describe single measured quantities that can also be used as parameters for sensitivity analysis or parameter estimation.
- *Real List Variables* are used to provide measured data or to formulate dependencies on other variables with the aid of interpolated data pairs.
- *Variable List Variables* are used to interpolate between other variables at given values of an arbitrary argument (*e.g.*, for multi-dimensional interpolation).
- *Formula Variables* allow the user to build new variables as algebraic expressions of other variables.
- *Probe Variables* make the values of other variables evaluated at a given location in a compartment globally available.

Recently, UNCSIM (Reichert, 2006) was developed by the same author as a companion tool for Aquasim. It implements various systems analysis methods and is based on a standardized flat file based interface.

2.4.2 BioWin

BioWin is a Microsoft Windows-based simulator (W2K, XP, Vista) used in the analysis and design of wastewater treatment plants. It is developed by EnviroSim²⁸ (Flamborough, ON, Canada) and has especially received appraisal from process engineers.

BioWin is a tool that tries to provide for increased efficiency during the plant design process by very closely following the line of thought of process engineers in its user interface. It focuses on treatment plants only, and does not have integrated modelling ambitions. As most other water quality tools, BioWin allows for coupled models to be built on the basis of model libraries containing unit processes. These unit processes can only be entered through a Petersen matrix editor, as BioWin does not use a high-level modelling language for the representation of its models. Next to standard ASM1, ASM2d and ASM3 models, BioWin also provides for its own activated sludge model (the so-called ASDM - Activated Sludge Digestion Model) which is more elaborate and includes new research results obtained by the authors of the product. However, the exact nature of the ASDM is not clear, as a detailed description of this model is not provided. In any case, the number of state variables that can be used in BioWin is fixed and is determined by the ASDM. Other activated sludge models (such as the ASM series) have to adapt to this limitation. Another limitation of the BioWin platform is that it does not offer elaborate support for modelling control strategies.

²⁸<http://www.envirosim.com>

2.4.3 GPS-X

GPS-X (GPS stands for General Purpose Simulator) from Hydromantis, Inc.²⁹ (Hamilton, ON, Canada) was first released in 1991 and comes in four different configurations: Entry, Advanced, Professional and Enterprise. Recent versions are implemented in Java, except for the modelling and simulation engine, which is based on ACSL. In spite of the use of Java, GPS-X only seems to be available for the Microsoft Windows platform (W2K, XP and Vista).

GPS-X supports dynamic and steady-state simulation, sensitivity analysis, optimization, scenario analysis, and Monte Carlo analysis, all of which are implemented through the ACSL engine. Available integration algorithms are Euler, 2nd and 4th order Runge-Kutta, Gear's Method, Adams-Moulton, 2nd and 5th order Runge-Kutta-Fehlberg and a DAE solver (DASSL). For optimization, the maximum likelihood algorithm (MLE) is used.

The product comes with a large number of pre-defined models, libraries and layouts but also allows for creating custom items of these types. Amongst the models that are provided are ubiquitous standard models such as ASM1, ASM2d, ASM3 and ADM1, and non-standard product-specific models such as Mantis (modified version of ASM1) and two-step Mantis (Mantis with two-step nitrification). Dynamic model input can be provided through input files or through graphical input control widgets (such as sliders and switches). Simulated output data can be visualized through graphical plot widgets or saved in output files. Also, control strategies can be added to plant layouts using typical On-Off, P, PI and PID controllers.

Additional functionalities include automated conversion of units (based on metric, imperial and custom unit systems) and generation of reports including graphical layout representations, parameter values, initial conditions, and output data represented in graphs and/or tables. Also, a number of spreadsheet-based applications are available that allow for influent characterization and Petersen matrix editing.

GPS-X is a complete product in terms of functionality and has a partially open modelling environment (stoichiometry and kinetics can be modified through a Petersen matrix representation, both other model aspects can only be modified by altering the underlying ACSL code, which is only feasible for experts). However, it relies on an obsolete modelling language (ACSL) and has a closed and non-transparent virtual experimentation environment. Focus is on treatment plants only, the product does not have integrated modelling ambitions.

2.4.4 Simba

Simba was first developed in 1994 at the Institut für Automation und Kommunikation e. V. Magdeburg³⁰ and was later commercialized by its spin-off company IFAK System GmbH³¹. The product comes in two configurations (Light and Standard) and is available for Microsoft Windows (9x, NT, W2K and XP). Simba is entirely based on MATLAB/Simulink and can basically be considered as a toolbox containing Simulink model libraries and some specialized user interface elements. In addition to well-known WWTP models based on ASM1, ASM2d, ASM3 or ADM1, Simba also includes models for river and sewer modelling.

Additional features of the software are the ability to exchange models through the SBML standard representation, to retrieve and store time series from databases, and to interact with Microsoft Excel. Next to Simba Standard and Simba Light, the Simba suite also contains tools that allow for on-line monitoring and control.

Simba focuses on integrated modelling and offers enormous flexibility thanks to the fact that it is based on MATLAB/Simulink. However, it does not include a high-level modelling language. In practice

²⁹<http://www.hydromantis.com>

³⁰<http://www.ifak.eu>

³¹<http://www.ifak-system.com>

it is mainly adopted for academic problems.

2.4.5 STOAT

STOAT (Sewage Treatment Operation Analysis over Time) is a program that performs dynamic simulation of wastewater treatment facilities. It has been under development at WRc plc³² (Swindon, Wiltshire, UK) for the past 10 years and during its development, each process model was validated against performance data from sewage treatment plants. Extensive modelling work has subsequently been undertaken to confirm the validity of individual process models.

STOAT is available for Windows and allows users to quickly build a plant model and enter the required data using simple dialog boxes. Results can be displayed while a simulation is taking place and are also stored as data files for subsequent analysis. STOAT is built around a graphical editor for coupled models but does not allow for atomic models to be built from within the program. Instead, these must be programmed manually using a general-purpose language and added to STOAT as a dynamically-linked library. Since the program does not allow for atomic models to be constructed, no Petersen matrix editor is available either.

Experiment types that are implemented include dynamic simulation, sensitivity analysis, parameter estimation and scenario analysis. For integration, several Runge-Kutta based solvers are available, as well as Gear-based methods, one Adams-method and some experimental codes.

Implementations of the standard ASM1, ASM2, ASM2d and ASM3 models are available. An implementation of ADM1 is under construction. RWQM1 is not available. Other models that are available include BOD-based activated sludge models, the Mosey model for anaerobic digestion and biofilm models based on the Wanner/Gujer model.

STOAT has a COM/DCOM interface and has recently also been made compliant with the OpenMI protocol. The STOAT GUI is implemented in Microsoft Visual Basic, while its numerical kernel was programmed in FORTRAN.

According to its developers, STOAT is a program, rather than a program-writing program. Its design permits users to always look at results, without needing a valid license, to ease distribution to clients. The focus is on providing a graphical environment for modelling with a large library of wastewater treatment models, rather than an environment for developing mathematical models.

2.4.6 WEST

WEST (World-wide Engine for Simulation, Training and Automation) is a modelling and virtual experimentation tool that is distributed by MOSTforWATER NV³³ (Kortrijk, Belgium). The tool is generic in nature, but is marketed specifically as an environmental modelling and virtual experimentation tool, and is hence nearly exclusively used in this domain (*e.g.*, for wastewater treatment plants, rivers and sewer catchments). Potential users of WEST include researchers, training centers, consultants, design engineers and plant operators. However, the product finds its largest following in the first category.

WEST-1 was released in 1993 for the IBM AIX platform. It was based on a rigid spreadsheet-based model description format and only had dynamic simulation experiments to offer. With WEST-2, the Tornado-I kernel (developed at the Department of Applied Mathematics, Biometrics and Process Control of Ghent University) was introduced. With it came the shift to the Microsoft Windows platform, the adoption of the MSL modelling language and the introduction of optimization as an experiment type. MSL (Vanhooren et al., 2003) is an object-oriented, equation-based modelling language that has allowed for the development of an elaborate water quality library, which is distributed with WEST as of version 2,

³²<http://www.wrcplc.co.uk>

³³<http://www.mostforwater.com>

and includes models based on ASM1, ASM2, ASM2d, ASM3, ADM1 and RWQM1, and a large number of sensor and controller models. WEST-3 saw the introduction of additional experiment types, *i.e.*, for sensitivity analysis and scenario analysis. Also, the user interface was improved: a Petersen matrix editor, a model library browser, and a textual model editor with syntax coloring were added. WEST-3 also includes an externally callable COM-based application programming interface that allows for most of the WEST functionality to be used from custom applications. More specifically, WEST-3 has been successfully integrated into synoptic panels, training simulators and SCADA (Supervisory Control And Data Acquisition) systems. An interesting feature of WEST is the fact that textual model descriptions can be encrypted, which allows for models to be used and distributed without revealing the internals of the model. In 2008, WEST-4 will be released. It will be the first commercial product that is fully based on the Tornado-II framework.

WEST has an open architecture that is generally appreciated by its customers. Its modelling environment, which is based on a high-level modelling language and a model compiler that generates executable model code, allows for high manageability and efficiency while constructing complex coupled models. WEST is used for integrated modelling and has served as a platform for the development of large-scale models covering treatment plants, sewer networks and river systems.

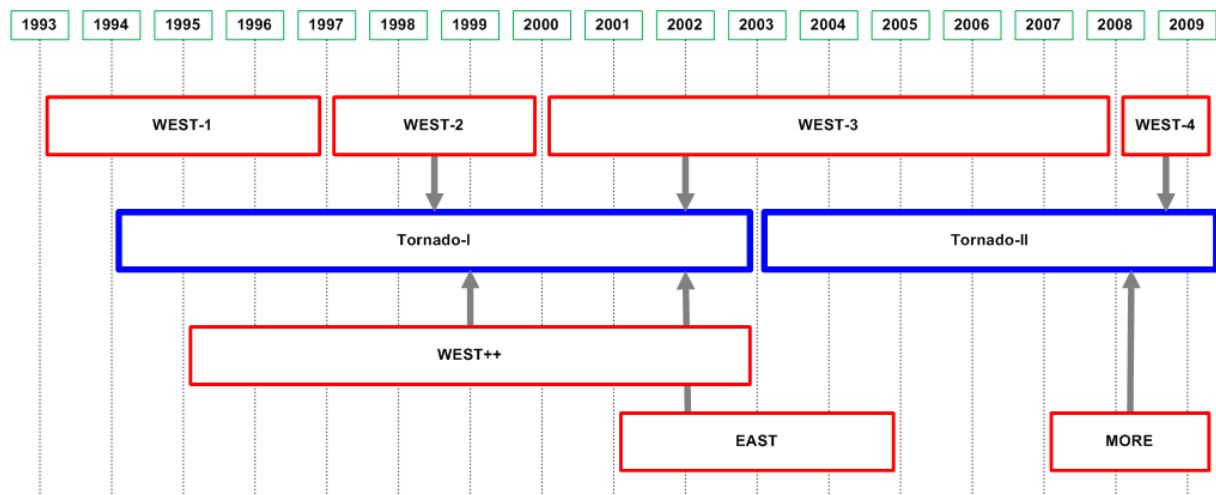


Figure 2.3: Timeline and Dependencies of WEST and Related Software

Figure 2.3 shows a timeline for the development of WEST, and the relationship between the various WEST versions and the Tornado kernel. In the figure are also a number of Tornado-related research applications that will be briefly discussed in Chapter 12.

3

Problem Statement

3.1 Complex Modelling

When investigating the evolution of the complexity of models in recent years, one must conclude that the speed at which complexity is growing is accelerating. Especially in the domain of water quality management, this tendency is very apparent.

In order to illustrate the evolution of complexity over the past ten years in water quality management, several metrics could be used. In the following examples, a metric that is related to the internal representation of models within model compilers is adopted, more specifically the number of nodes pertaining to the representation of a model as an abstract syntax tree (AST). It will be explained in Chapter 9 what exactly this measure means. At this point however, it suffices to remember that the more complex the model, the more nodes the abstract syntax tree holds.

Ten years ago, one was mainly concerned with modelling individual unit processes, such as the activated sludge process. In 1996 for instance, a typical modelling study would have focused on one activated sludge tank. The number of AST nodes for such a model, represented in a high-level modelling language, is approximately 10,000. In 1999, focus had shifted towards modelling entire treatment plants, including several unit processes. The Benchmark Simulation Model No.1 (BSM1) (Copp, 2002) is an example of a treatment plant including five activated sludge tanks in series, and a clarifier. The number of AST nodes for BSM1 is approximately 60,000. In 2002, real-world plants, several times bigger than BSM1, were being modelled. The model of the Galindo plant (Ayesa et al., 2006) in Bilbao for instance, has about 250,000 nodes in its AST representation. In 2004, integrated modelling (combined modelling of rivers, sewers and treatment plants) had become of interest to many researchers. The integrated model of a river/sewer system in Luxembourg (Solvi et al., 2006) had no less than 850,000 AST nodes. Finally, in 2007, researchers at CEIT in San Sebastian (Spain) launched a new modelling approach aimed at plant-wide modelling (PWM) (Grau et al., 2007a,b). The PWM model of the Galindo plant (including many more processes than the original model from 2002) requires a dazzling number of AST nodes to represent, *i.e.*, approximately 2,300,000. It is difficult to imagine where this trend will ultimately lead to, but by plotting (*cf.* Figure 3.1) the number of AST nodes of the above examples versus time, it seems logical to assume that the rise in complexity is far from over.

When the evolution of complexity is plotted against a logarithmic scale (*cf.* Figure 3.2), the trend line is linear and has a slope of approximately 0.17. Interestingly, in 1965 Gordon Moore predicted that

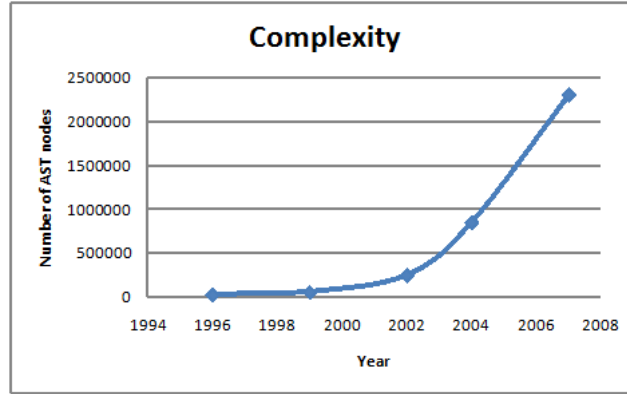


Figure 3.1: Evolution of Model Complexity in Water Quality Management (Linear Scale)

the number of transistors on an integrated circuit would double approximately every two years. So far, this prediction, which is popularly known as Moore's Law¹, has proven to be accurate. When plotting (against a logarithmic scale) the number of transistors on one integrated circuit from the 1970's until now, a linear trend line is discovered (*cf.* Figure 3.3). Also in this case the slope of the trend line is close to 0.17. We can therefore conclude that the complexity of water quality models evolves at the same pace as the complexity of the hardware (integrated circuits) on which these models are implemented.

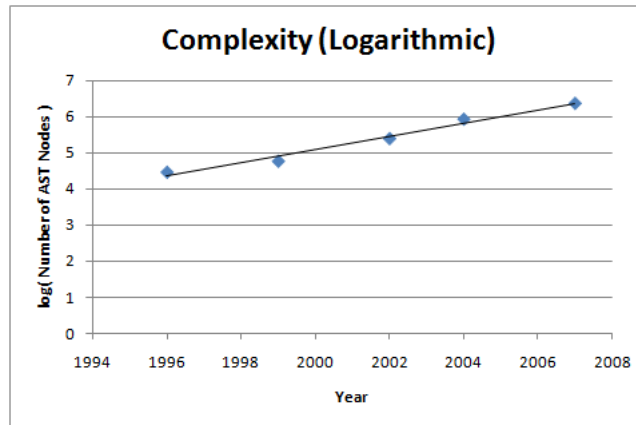


Figure 3.2: Evolution of Model Complexity in Water Quality Management (Logarithmic Scale)

Astonishingly, in his 2006 article "Activated sludge modelling: past, present and future" (Gujer, 2006), Willy Gujer comes to the exact same conclusion, however based on a different metric. In the Gujer metric, the complexity of activated sludge models is defined by the following equation:

$$C = n_i \times n_j \times n_x \quad (3.1)$$

In this equation C is the measure of complexity, n_i is the number of different compounds (state variables) considered, n_j is the number of transformation processes in the biokinetic model and n_x is the spatial resolution (the number of spaces for which information is provided). Applied to Gujer's own case studies published during the past 25 years, this metric results in a trend line with almost exactly the same slope as Moore's Law and the metric based on the number of AST nodes.

¹<http://www.intel.com/technology/mooreslaw>

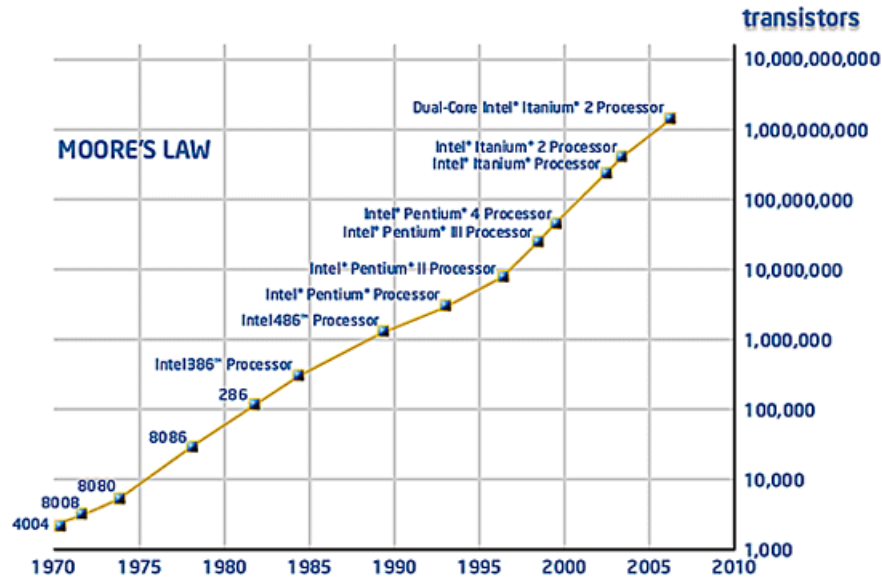


Figure 3.3: Moore's Law (Logarithmic Scale)

In order to be able to handle the growing complexity depicted above, two issues are critical: the ability to conveniently *represent* complex models, and the ability to efficiently *execute* them. When evaluating the state-of-the-art tools that are currently available, none of them seems to be entirely ready for the future challenges that are brought by the desire to perform complex modelling of environmental systems. In fact, what is needed to tackle the complexity issue is a tool that combines powerful model representation capabilities with the generation of efficient low-level executable model code. Powerful representation capabilities include the use of high-level (declarative, object-oriented) textual modelling languages, graphical representations of layouts, and domain-specific representations such as the Petersen matrix.

Table 3.1 gives a coarse comparison of the tools discussed in Chapter 2. From this table, one can conclude that each tool has its own advantages and disadvantages. The only tool that, as a result of the work done in the scope of Tornado-I, currently implements all features listed in the table, would be WEST. However, although all features mentioned in the table are present in the version of WEST that is currently available, this version is still not powerful enough to handle the latest generation of large-scale systems, because of a number of deficiencies in the Tornado-I kernel. The most important modelling-related problems with this kernel are the following:

- **Excessively long model compilation times:** For large models ($> 300,000$ nodes), the time required to generate an executable model is excessively long and ranges from 10's of minutes to hours on an average workstation.
- **Excessive executable model sizes:** Again for large models, the size of the executable model code that is generated may be larger than the size that is accepted by general-purpose language compilers.
- **Insufficient support for non-ODE equations:** Models not only become more complex in terms of their size, they also increasingly exhibit features that cannot be implemented as plain ODE's. In other words, it becomes increasingly important to be able to model DDE's, hybrid systems, and to a lesser extent also DAE's.

Table 3.1: Comparison of Modelling and Virtual Experimentation Tools

Tool	High-level Textual Representation	Graphical Representation	Domain-specific Representation	Low-level Comprehensive Executable Models
ACSL	No	Yes	No	Yes
MATLAB/Simulink	No	Yes	No	No
Mathmatica	No	Yes	No	No
Mathcad	No	No	No	No
LabVIEW	No	Yes	No	No
Maple	No	No	No	No
Modelica	Yes	Yes	No	Yes
Aquasim	No	No	Yes	No
BioWin	No	Yes	Yes	No
GPS-X	No	Yes	Yes	Yes
Simba	No	Yes	Yes	No
WEST	Yes	Yes	Yes	Yes

It should be noted that the “domain-specific representation” label that is used in Table 3.1 is debatable. In computer science, a Domain-Specific Language (DSL)² is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain. Domain-specific languages are usually declarative. Consequently, they can be viewed as specification languages, as well as programming languages. Many DSL’s are supported by a DSL compiler that generates applications from DSL programs. In this case, the DSL compiler is referred to as application generator in the literature (Cleaveland, 1988), and the DSL as application-specific language.

In view of the above, UML profiles and also Modelica libraries might be regarded as domain-specific representations. Therefore, from a computer science point of view, Modelica should actually be labelled as supporting domain-specific representations in Table 3.1. However, most water quality researchers will be inclined to think that Modelica is too generic to be considered a domain-specific representation (even though it has libraries that are intended for modelling phenomena from different domains). With respect to Table 3.1, it was decided to let the point of view of water quality researchers prevail.

3.2 Complex Virtual Experimentation

When studying complex systems, the ability to perform dynamic simulations is of prime importance. Dynamic simulation allows for explaining past behavior and/or predicting future behavior of a system. However, a tool that merely allows for running dynamic simulations is no longer adequate in the present-day context. Other types of virtual experiments that for instance allow for parameter estimation, automated optimization, sensitivity analysis and uncertainty analysis have become equally important.

Most tools discussed in Chapter 2 have the ability to perform virtual experiments other than dynamic simulation. Generic tools typically allow for extensive flexibility in setting up virtual experiments through an appropriate combination of primitive statements in interpreted technical computing languages. Domain-specific tools usually only allow for a limited set of (often unstructured) virtual experiment types, and do not provide elaborate configuration options. In our view, both approaches are problematic. The approach taken by generic tools is not always desirable because it requires programming skills. Also, different users will often re-implement time and again commonly needed procedures.

²<http://homepages.cwi.nl/~arie/papers/dslbib>

Hence, this approach is not the best in terms of re-usability. The approach taken by domain-specific tools is equally problematic since the set of available virtual experiment types is often too limited and too rigid: new experiment types cannot be added easily (nor by users, nor by developers) and the number of configuration options is insufficient. From our perspective, there is a need for a tool that offers an extensible set of readily available and highly configurable virtual experiment types.

Virtual experiment types are usually very computationally intensive, especially those that are based on parameterized simulation runs. In order to reduce the computational bottleneck imposed by these types of procedures, distributed execution is often a natural solution. In distributed execution, an application is split into a number of components that are run on separate computational nodes and communicate through the network. Recently, several generic tools have seen the introduction of engines for distributed execution. However, as for the implementation of virtual experiments in generic tools, the use of these distributed execution capabilities requires some programming skills and can therefore not be considered fully transparent. For domain-specific tools the situation is even worse, since distributed execution features are nearly always unavailable.

3.3 Deployment and Integration Flexibility

During the past decades, tools for modelling and virtual experimentation were software environments that largely stood by themselves. Often they could only be used interactively, through a graphical user interface. Once a virtual experiment had been set up with one or the other tool, it was very difficult to deploy it in some other way, *e.g.*, in a command-line environment, web-based environment, or as a remote or embedded component.

Recent years have seen a growing need for integration of software systems. The boundaries between applications, programming languages and even operating systems have slowly dissolved. A logical consequence of this is that also modelling and virtual experimentation tools need to provide for sufficient openness. A virtual experiment that has been set up should be deployable in a variety of ways, and options for integration with other platforms should be available.

When evaluating the state-of-the-art tools discussed in Chapter 2, one must conclude that most generic tools have in recent years started to offer support for a wide variety of deployment and integration options. Domain-specific tools however are far behind in this respect.

3.4 Modern Architectural Standards

Notwithstanding the *plethora* of modern design and implementation options that are available nowadays, it must be noted that many scientific and technical computing solutions adhere to obsolete design principles and coding standards. Also, one will find that the bulk of scientific and technical computing algorithms that are available through libraries such as NETLIB³ and GAMS⁴ are still implemented in languages such as FORTRAN or C. Also frequently used books on numerical computing software such as Numerical Recipes (Press et al., 1992) present the software they discuss in these languages. Since the source code of most of the tools discussed in Chapter 2 is not publicly available, we can not comment on it specifically, however it may be assumed that a number of the tools discussed in Chapter 2 will most likely also suffer from a lack of adherence to modern architectural standards.

When discussing the topic of modern architectural standards in the scope of scientific and technical computing, the issue of performance is always raised. It is indeed a fact that in order to provide an acceptable level of performance, one sometimes may have to fall back to low-level approaches. However,

³<http://www.netlib.org>

⁴<http://gams.nist.gov>

in our view, it should always be investigated what these situations are in order to restrict the use of low-level approaches to only those situations that absolutely require them. Also, in case low-level approaches are adopted, these should be abstracted in such a way that the overall readability and maintainability of the software is not impeded.

3.5 Conclusion

As a conclusion, we can state that although a large number of generic and domain-specific tools are available that allow for modelling and virtual experimentation in the area of water quality management, none are fully able to aptly support the most recent research in this area in a timely and convenient fashion. Moreover, future maintainability and re-usability of existing software systems are impeded by a lack of adherence to modern architectures standards and insufficient options for deployment.

4

Requirements

4.1 Introduction

In Chapter 3 it has been pointed out that current state-of-the-art software frameworks are not fully up to the challenges that are brought by future environmental modelling and virtual experimentation problems, especially in water quality management. This chapter therefore gives an overview of the requirements that a framework that can overcome these future challenges is subjected to. These requirements have been used as a guideline during the design and development of the Tornado kernel.

4.2 Complex Modelling

When creating a model, *i.e.*, an abstract representation of a real-world system, it is of prime importance to be able to use a formalism that is appropriate with regard to the problem at hand. Appropriateness has many facets and is related to accuracy, convenience, readability, maintainability, *etc.* When investigating complex systems in general, it can be noted that many are heterogeneous, *i.e.*, they are comprised of sub-systems that pertain to different domains, use different timescales, have a different degree of continuity or discreteness, *etc.* As a result of this, creating a model of a complex system on the basis of a formalism that is only appropriate for one particular type of system, is usually not a very good option. In order to tackle this problem, several approaches have been used and/or suggested:

- **Component-based modelling:** In this approach sub-systems are modelled separately using the most appropriate formalism, and using a dedicated model executor for each individual sub-model. The combination of a sub-model and its executor forms a self-contained entity that is referred to as a component (also named model engine). The overall model is then comprised of a number of components that interact during model evaluation. Advantages of this approach are not only that for each sub-system the most appropriate formalism can be used, but also that legacy models can easily be re-used as a component to model a part of a larger system. Disadvantages are communication overhead and the fact that readability and maintainability of a component-based architecture are low, because of the large variety of tools and languages that can be used. Examples of component-based modelling frameworks are OpenMI, the Open Modelling Interface¹

¹<http://www.openmi.org>

(Gregersen et al., 2007), and SKF's TLM (Transmission Line Modelling) (Siemers et al., 2006).

- **Comprehensive modelling language:** A second approach is based on the use of comprehensive modelling languages that allow for modelling a complex system in all its aspects, using an extensive array of language constructs and only one executor. In other words, a comprehensive modelling language is a super language that attempts to make other languages and paradigms unnecessary. In principle, a true comprehensive language belongs to the realm of utopia, since we will never be able to devise one language (no matter how powerful and elaborate) that is capable of conveniently describing every possible aspect of every possible system. However, certain languages have managed to cover particular fields that well that they have rendered other languages unnecessary. A typical example is Modelica, which is increasingly imposing itself as a *de facto* standard for hybrid system modelling. Attempts have been made to extend the Modelica language in order to let it cover even more ground, but these have not always been equally successful. From our perspective, one should refrain from trying to create true super languages, for these attempts are doomed to fail as the resulting language will be too difficult to use and to maintain.
- **Multi-formalism modelling:** A third approach attempts to use a conversion mechanism to translate one formalism into another. In this way, sub-systems could be modelled in the most appropriate formalisms, as in the case of component-based modelling. However, instead of letting each sub-model be executed by its own executor, a conversion would occur that translates each sub-model into a representation in a common formalism. The set of sub-model representations in this common formalism would then be unified and executed by one single executor. To our knowledge, the multi-formalism approach has never been fully realized in practice, although (Vangheluwe, 2000) describes some initial work in this direction. It is often suggested that DEVS (Zeigler, 1976) could serve as a common formalism, to which other formalisms could be translated. DEVS is a very low-level formalism and could hence be regarded as the “assembly” of modelling formalisms.

Taking into account the current state of technology, we believe that for handling complex modelling in the field of environmental - *i.e.*, water quality - systems, an approach that uses a high-level language such as Modelica in combination with a component-based architecture is most favorable. Every aspect of a complex system that can be conveniently modelled in a high-level language, should be. Other aspects (or legacy models that one wishes to re-use) can then be additionally linked as separate components.

We believe that in order for high-level languages to be successful, they need to support the set of (partly overlapping) paradigms that is listed below:

- **Equation-based Modelling:** An equation-based model is a model that captures system characteristics by identifying system variables and describes the system with a set of equations relating to these variables. Equation-based modelling has a long history, and is widely used in many areas, including water quality management. The vast majority of models available through literature are based on mathematical equations. In water quality modelling, especially AE's, ODE's and DDE's are of importance.
- **Hierarchical Modelling:** In hierarchical modelling, a model is composed of a number of sub-models, in a recursive manner. The model building process can either be top-down or bottom-up. In water quality modelling, the integrated urban wastewater system composed of treatment plants, sewer network and river systems, is usually modelled by combining unit processes into larger wholes, typically through bottom-up modelling.
- **Object-oriented Modelling:** In this approach, the object-orientation paradigm that is known for programming languages such as C++ and Java is applied to modelling. This object-orientation

paradigm is based on principles such as instantiation, encapsulation, inheritance and polymorphism. It is well-suited for the description of large models and model libraries since it intrinsically allows for re-use and abstraction of information.

- **Declarative Modelling:** A declarative model is a model that describes objects and their relationships and states, without specifying how to evaluate those descriptions. A model in a general-purpose programming language is non-declarative, since next to information on objects and relationships (*e.g.*, mathematical equations), it will also contain information about how to evaluate the model (*e.g.*, which integration solver to use). A non-declarative model is unfavorable from the point of view of readability and maintainability since the actual model information is obfuscated by technical execution details.
- **Acausal Modelling:** In a causal model, the flow of data is determined upfront: inputs and outputs are clearly distinguished in the model representation. In an acausal model this is not the case, and it is left up to the model processing mechanism to automatically determine the causality. Acausal models are easy to build and modify, and as a result acausal modelling is a convenient way to express specifications. However, acausal models require elaborate tools to be handled efficiently. Causal models on the other hand are more difficult to build and modify, but generally require less elaborate tools. Water quality models are typically either described on the basis of masses or concentrations. Integration of models using different approaches is difficult since either all masses have to be recomputed on the basis of concentrations ($M = CV$), or *vice versa* ($C = M/V$). It would be much more convenient if it would be possible to simply express the general relationship between masses and concentrations ($0 = M - CV$) and leave everything else up to the model processing mechanism.
- **Multi-domain Modelling:** In multi-domain modelling, models are created that have aspects that pertain to different classical domains such as electronics, thermodynamics, fluid-mechanics, biology, *etc.* In water quality management, research was first focused on sub-domains such as treatment plants, sewers and rivers. Since these sub-domains are integrated more and more, it is to be expected that also integration with other domains will follow. The ability to perform multi-domain modelling is therefore important.

Next to the fact that powerful mechanisms are required for describing models in a convenient manner, it should also not be forgotten that timely execution of models is crucial. In principle, these requirements appear as a paradox, since optimal performance is normally obtained with low-level hand-crafted solutions, which are notoriously bad from the point of view of readability and maintainability. On the other hand, solutions that provide for convenience by adopting the paradigms discussed above will incur substantial overhead and will therefore not lead to timely execution. Luckily, a compromise exists that satisfies both goals: a convertor, more specifically a model compiler, can be used to process high-level, “convenient” model descriptions and generate low-level, “fast” executable models. We therefore believe that a model compiler based approach is required for solving the issue of complex modelling. Unfortunately, model compilers can also introduce their own set of issues. For instance, the compilation process may be slow, the generated code may be too large, or run-time error reporting may be difficult since the generated executable code has lost its relationship to the original high-level model representation. These are all potential problems that have to be taken into account when designing a model compiler based software system.

One last aspect that is of importance with respect to complex modelling, is the need for domain-specific representations (such as the Petersen matrix), next to the generic high-level model representations that were discussed hitherto. Experience has shown that without these representations, software tools are unlikely to become successful in the water quality domain.

4.3 Complex Virtual Experimentation

The use of models for solving problems in environmental sciences and other disciplines is a powerful methodology. However, not all questions can be solved by merely performing a dynamic simulation of a model and looking at the results. Often, more convoluted procedures are needed, *e.g.*, optimizations and parameter sweeps. We therefore believe that it is unfortunate that historically a clear separation has existed between simulation and what is often named as systems analysis. In our view, scientific software systems need to support a wide-range of techniques based on model evaluation, and more importantly, provisions for these different techniques should be made starting from the initial design and implementation of the software.

As has been pointed out in Chapter 3, domain-specific software tools usually do not provide an extensive set of model evaluation procedures, but those that are provided are readily available. Generic tools on the other hand have the capacity of implementing an unlimited number of procedures, but at the expense of having to develop them from scratch on the basis of a number of low-level primitives. We believe that a mixed approach is required, in which an extensible set of readily available procedures is provided. The design and implementation of these procedures should be based on common principles, so that they can easily be managed.

Below is a list of procedures based on model evaluation that we believe should be available in complex virtual experimentation software systems (in Chapter 8 most of these procedures are discussed in further detail):

- **Simulation:** Determines the dynamic behavior of a system over a specific time horizon.
- **Steady-state analysis:** Discovers the state of a system when transient effects have worn out, *i.e.*, when all derivatives have become zero.
- **Sensitivity analysis:** Analyzes how much the value of a computed variable changes when a small change is applied to parameters, input variables or initial conditions. The overall purpose is to determine the sensitivity of a system to small changes to its properties.
- **Optimization:** Discovers the values of parameters, input variables and initial conditions that minimize a certain objective function. This procedure can be used for parameter estimation, to optimize operational conditions, *etc.*
- **Uncertainty analysis:** Analyzes the effect of various possible uncertainties with respect to a model or its environment. Typically, uncertainties are related to parameters, input data, or the model structure itself.
- **Optimal experimental design:** Determines experimental conditions for real-world experiments that are optimal with respect to a certain objective function. Typically, optimal experiment design is used for more accurate parameter estimation or better model structure discrimination.

We believe that these types of procedures should be based on common principles and components, for instance for managing input and output, for the representation of data structures, for dealing with numerical algorithms, *etc.*

As mentioned before, the performance of a modelling and virtual experimentation system is of major importance. One must therefore take care not only to provide for efficient executable models, but also for a strict limitation of the overhead that is caused by model-based virtual experimentation procedures. Especially the efficiency of handling input and output is of major concern, more specifically when it occurs within the simulation's inner loop.

4.4 Deployment and Integration Flexibility

In the present-day context, a software system is doomed if it entirely stands by itself and cannot interact with other systems. An advanced modelling and virtual experimentation framework should therefore provide for deployment and integration flexibility, preferably on several platforms. We believe this can be realized by making a distinction within the framework between a kernel that is implemented in a platform-independent manner, and a set of (possibly platform-dependent) communication and programming interfaces that allow for communication with the outside world. In this way, it should be possible to use the kernel for the following types of applications:

- **Stand-alone Applications:** Consist of a user interface (graphical or command-line) that directly, *i.e.*, on the same workstation, interacts with the kernel, through a programming interface. This is the most classical type of application, and should evidently also be available for an advanced modelling and virtual experimentation framework.
- **Remote Applications:** Consist of a user interface (graphical or command-line) that interacts with a kernel that is located on a remote workstation, through a remote communication interface. For a modelling and virtual experimentation framework, this type of application can be useful in case client workstations are to be served by a kernel that runs on a powerful remote server workstation.
- **Web-based Applications:** Consist of a user interface (typically run from within a web browser) that interacts with a kernel that is located on a remote workstation, through a HTTP-based protocol. This type of application has become very popular in recent years.
- **Distributed Applications:** This type of application is comprised of a number of components that run on different workstations and interact through a communication protocol. The goal is usually the distribution of the computational load, which in a modelling and virtual experimentation framework is usually substantial.
- **Embedded Applications:** These are applications that reside within another application or system, and are not directly visible to the outside. Embedding of a modelling and virtual experimentation kernel inside a controller or SCADA system is often desired.

In short, the WORA² (Write Once, Run Anywhere) principle that lies at the basis of successful frameworks such as Java, should also be applicable to models and virtual experiments that are created in the scope of Tornado.

4.5 Modern Architectural Standards

It is often the case that modelling and virtual experimentation software is developed according to obsolete architectural standards. Reasons for this are typically either the fact that it is assumed that by using modern standards performance requirements cannot be met, or that the development is done by researchers who may be well-versed in environmental sciences, but only have a limited background in computer science and software engineering. We believe that for the development of an advanced modelling and virtual experimentation framework, modern architectural standards should be used to the largest possible extent, except for a limited number of situations where it can be clearly demonstrated that a low-level approach is required in order to comply with performance requirements. Also, the adoption of modern architectural standards should be such that the extension of the framework should be possible for researchers with a limited computer science background, by adhering to the rules and the basic architecture that were set up by the developers of the initial version of the framework.

²http://en.wikipedia.org/wiki/Write_once,_run_anywhere

Part II

Design and Implementation

5

Design Principles

5.1 Introduction

This chapter gives an overview of the principal architectural design decisions that were taken with respect to the Tornado modelling and virtual experimentation kernel. Some of these principles were already present in the first incarnation of the kernel, others however have only been introduced as of the second incarnation.

5.2 Overall Architecture

Figure 5.1 shows the top-level architecture of the Tornado kernel. As can be seen, a strict separation exists between the Modelling Environment and the Virtual Experimentation Environment. In the Modelling Environment, high-level model descriptions are constructed and subsequently translated to executable models by a model compiler. The Experimentation Environment uses these executable models during the course of virtual experiments in order to generate data, to be interpreted by the user of the system. The fact that a strict separation between both environments exists is a natural consequence of the use of binary executable models. However, it was also a conscious decision, since a strict separation allows for different implementations of Modelling or Experimentation Environments to be used together, as long as the same executable model interface is supported.

As Tornado is a kernel rather than an application, Figure 5.1 also shows that the main elements of the kernel, *i.e.*, model compiler and virtual experiment executor, are strictly separated from the application by a control logic layer.

The separation between Modelling and Experimentation Environment on the one hand, and between application and kernel on the other, has been the basis of the Tornado kernel since its inception, and has hitherto remained in effect.

5.3 Entities

In contrast to Figure 5.1, which gives a high-level overview of the data/control flow in the Tornado kernel, Figure 5.2 gives a high-level view of the main structural entities of the kernel. Entities are considered to

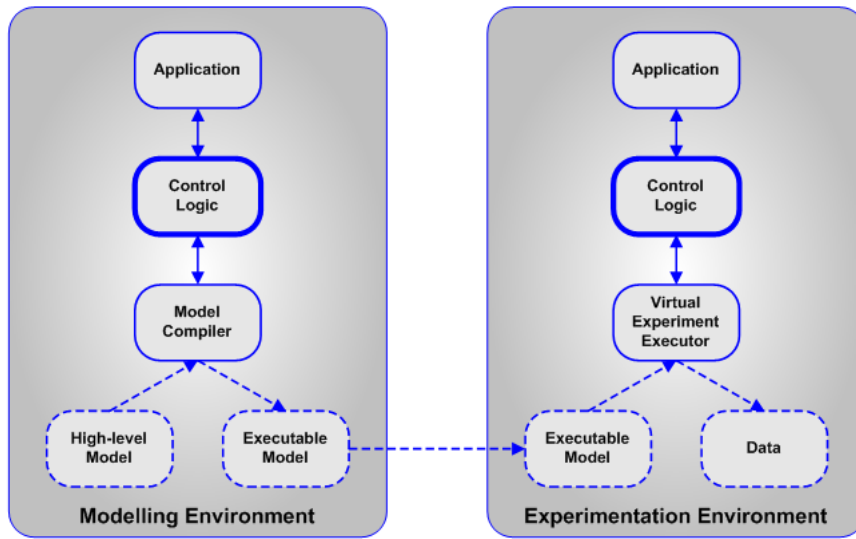


Figure 5.1: Overall Architecture of the Tornado Kernel

be objects that have a state and exhibit a certain behavior. Each entity has an internal representation and, if needed, also a persistent representation. The total number of entities in Tornado is currently 159. Only the most important entities, named **top-level entities**, are accessible from outside the kernel through interfaces (to be discussed later).

The Experimentation Environment has the following top-level entities:

- **Experiments** are instances of the various types of virtual experiments that have been implemented in Tornado. Experiments can be created, modified, run and saved.
- **Controls** contain specifications of graphical widgets that can be used to provide input to the model's input variables. Controls can be created, modified and saved. However, they cannot be rendered. Rendering controls is the responsibility of applications built on top of the Tornado kernel.
- **Plots** contain specifications of graphical widgets that can be used to accept output from output variables of the model. Plots can be created, modified and saved. However, as controls, they cannot be rendered since this is the responsibility of applications built on top of the kernel.
- **Solvers** implement numerical algorithms that are used during the execution of virtual experiments. Solvers can be created, modified, saved and run in the scope of the execution of a virtual experiment.

The top-level entities of the Modelling Environment are the following:

- **Layouts** are graphs in which nodes represent unit processes and edges represent connections between inputs and outputs of unit processes. Layouts can be created, modified and saved. However, rendering is left up to the application that sits on top of the kernel.
- **Model Libraries** are collections of high-level model descriptions. Model libraries can be created, modified and saved.

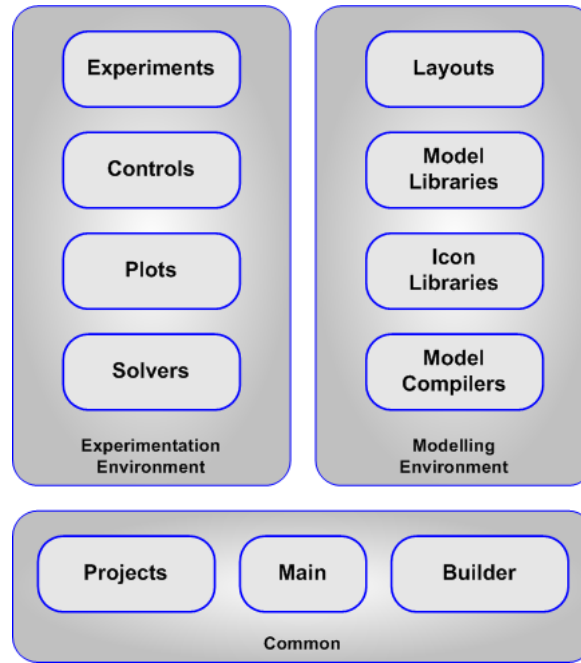


Figure 5.2: Entities of the Tornado Kernel

- **Icon Libraries** are collections of graphical representations of unit processes. Icon libraries can be created, modified and saved. Again, rendering is left up to the application.
- **Model Compilers** are translators that convert high-level model descriptions to low-level executable code.

Finally, three top-level entities are common to the Modelling and Experimentation Environments:

- The **Main** entity only has one instance. It is used to dynamically create other types of top-level entities and stores data items that have a kernel-wide scope.
- **Projects** are collections of layouts and related experiments. Projects can be created, modified and saved.
- The **Builder** entity is used to convert low-level executable model code to a binary representation that can be used by the virtual experiment executor.

Strictly speaking, the only entities that were supported by the initial version of the Tornado kernel were Experiments, Solvers, Model Libraries, Model Compilers and Builders. All other entities have been introduced at later stages.

5.4 Properties

When dealing with highly-configurable software systems, implementation is often hampered by the large number of accessor¹ and mutator² methods that need to be implemented in external interfaces (accessor

¹[http://en.wikipedia.org/wiki/Method_\(computer_science\)](http://en.wikipedia.org/wiki/Method_(computer_science))

²http://en.wikipedia.org/wiki/Mutator_method

methods - also known as *get* methods - allow for retrieving the state of an object, whereas mutator methods - also known as *set* methods - provide for a means to modify the state). In fact, also the first incarnation of Tornado was suffering from this problem. As a result, the kernel was difficult to maintain and external interfaces were easily broken due to constantly changing configuration options (which frequently required accessor and mutator methods to be added, removed and modified).

In order to circumvent the problems caused by large numbers of “volatile” accessor and mutator methods, a mechanism based on dynamically queryable properties has been foreseen from the onset of the design and development of Tornado-II. Each Tornado entity is able to report on the properties it supports, and for each property a description, data type, value and range of validity can be retrieved.

At the moment of writing, Tornado has 846 properties. In case the dynamic property mechanism would not be available, these would give rise to 1,692 methods to be implemented (846 accessors and 846 mutators). Each change to the signature of any of these methods, would break external interfaces.

5.5 Exceptions

By definition, a kernel is a software component that does not stand by itself and must be combined with other software components to create an application. Since a kernel cannot make any assumptions about the software components that it will be combined with, it needs to be as stable as possible. Hence, good error management is required. Traditionally, error handling in software is done on the basis of error codes. Object-oriented development however has introduced the concept of exceptions³. Exceptions are annotated instances of exception classes that are thrown as a result of error conditions, and can be caught at locations in the code where these error conditions can be remedied. Although exceptions are known to cause extra overhead and hence a degradation of performance, code based on exception-handling is usually far better structured and more stable than traditional code based on error codes.

The Tornado-I kernel was based on error codes, mainly for practical reasons inspired by the technology available at the time. Tornado-II however is entirely exception-based and has proven to be far more stable than its predecessor.

5.6 Events

As mentioned above, a kernel cannot make any assumptions about the environment (the application) in which it will eventually operate. This means that, in certain cases, the kernel should not make decisions about what is to be done when a specific condition arises, but should rather leave this decision up to the calling application. However, this implies that the kernel should be able to communicate the occurrence of these conditions to the caller. In traditional programming languages such a behavior can be implemented through a mechanism based on callbacks⁴, newer languages however provide for an even better mechanism through the use of events and event-handlers. In fact, event-handling has certain similarities with the exception-handling mechanism discussed above.

Tornado-I had a poorly designed callback mechanism. In Tornado-II this has been remedied, and has been accompanied by an event-based mechanism. It will be shown later how these two mechanisms relate to each other.

³http://en.wikipedia.org/wiki/Exception_handling

⁴[http://en.wikipedia.org/wiki/Callback_\(computer_science\)](http://en.wikipedia.org/wiki/Callback_(computer_science))

5.7 Design Patterns

In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code, but rather a description or template for how to solve a problem, which can be used in many different situations. In (Gamma et al., 2003) several design patterns are described, four of which are also used in the scope of the Tornado-II framework (the original Tornado-I framework was not based on any of the well-known design patterns).

5.7.1 Singleton

The singleton pattern is a creational design pattern that is used to restrict instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across a system. It must be admitted that the singleton pattern is also considered as an anti-pattern since it is often used as a euphemism for global variable.

In Tornado, the Main entity is a singleton since only one instance of this entity exists. Moreover, the Main entity has a lifetime which is equal to the lifetime of the Tornado kernel: it is created the moment the kernel becomes active, and is destroyed when the kernel is closed down.

5.7.2 Factory

The factory pattern is another creational design pattern that deals with the problem of creating objects (products) without specifying the exact class of the object that will be created. The factory pattern handles this problem by defining a separate method for creating the objects, which subclasses can then override to specify the derived type of product that will be created. More generally, the term factory method is often used to refer to any method whose main purpose is the creation of objects.

Next to being a singleton, the Main entity in Tornado is also a factory. It is used to generate instances of other entities, *i.e.*, Experiments, Controls, Plots, Layouts, Model Libraries, Icon Libraries, Model Compilers, Projects and Builders. In the case of Experiments, various experiment types can be generated, corresponding to the respective types that are implemented by the Tornado kernel.

5.7.3 Facade

The facade pattern is a structural design pattern that is used to provide a simplified interface to a larger body of code, such as a class library. A facade can make a software library easier to use and understand. It can also provide for more stability since it may restrict the number of possible interactions between library components to a limited subset that is considered safe.

In Tornado, Experiments adhere to the facade pattern. They are internally made up of several interacting objects, however to the outside only one single interface is provided. It must be admitted that providing a facade for Experiments also has a number of disadvantages such as the fact that extra code is to be written that could otherwise be avoided, and the fact that some of the object-oriented nature of the system is lost. It is however felt that in this case, these disadvantages do not outweigh the fact that a facade is safer and more straightforward in its use, especially when remote and distributed execution middleware technologies are involved.

5.7.4 Proxy

A proxy, in its most general form, is a class functioning as an interface to another item. This other item could be almost anything: a network connection, a large object in memory, a file, or some other resource

that is expensive or impossible to duplicate. Several types of proxy patterns exist, but the one that is most frequently used in the scope of the Tornado framework is the remote proxy pattern. It provides a reference to an object located in a different address space on the same or a different machine.

5.8 Thread-safety

By itself, Tornado-I only provided support for the execution of one virtual experiment at a time. As a result of the use of global variables, concurrent execution of multiple experiments was not possible. In Tornado-II, the use of global variables has been kept to an absolute minimum. In fact, only the Main entity acts as a global object. Consequently, it is not a problem to create multiple top-level entities (Experiments or other) and let them execute tasks concurrently in different threads⁵. A thread in computer science is short for a “thread of execution”. Threads are a way for a program to fork (or split) itself into two or more simultaneously (or pseudo-simultaneously) running tasks. Threads and processes differ from one operating system to another, but in general, the way that a thread is created and shares its resources is different from the way a process does. Mostly, a process will contain several threads.

5.9 Persistency

In Tornado, many different pieces of information need to be stored (*i.e.*, made persistent) to a storage medium in order to be retrieved later. To favor openness, it was decided to avoid the use of binary storage formats, unless issues such as performance would make this impossible. Also, it was decided to use XML⁶ to the largest possible extent for non-binary formats, unless - again - this would not be possible for reasons of performance or other. The Extensible Markup Language or XML is a general-purpose markup language that is classified as an extensible language because it allows for new tags to be defined. Its primary purpose is to facilitate the sharing of structured data across different information systems. XML is recommended by the World-Wide Web Consortium (W3C). It is a fee-free, open standard.

Grammars of XML specifications are usually expressed in DTD’s (Document Type Definitions)⁷ or XSD’s (XML Schema Definitions)⁸. XSD’s are considered more type-safe and more elegant, in the sense that XSD’s are a form of XML by themselves.

For most Tornado entities, data can be made persistent in XML format. An XSD has been defined for each of these formats. The entities that can be stored in XML are Experiments, Controls, Plots, Layouts, Model Libraries, Icon Libraries, Projects, and the Main singleton. Model Compilers and Builders do not have an associated XML format at this point, but may in the future. Data on Solver entities is made persistent as part of the persistent format for Experiments. XML grammars of Tornado entities very closely match the internal representation of these entities.

5.10 Naming

In Tornado, care has been taken to favor consistency and orthogonality with respect to naming. Table 5.1 gives an overview of names that are used in the scope of the respective Tornado entities. As can be seen from this table, the scheme is coherent and therefore favors readability and maintainability of the Tornado framework. The meaning of the columns in the table is the following:

⁵[http://en.wikipedia.org/wiki/Thread_\(computer_programming\)](http://en.wikipedia.org/wiki/Thread_(computer_programming))

⁶<http://www.w3.org/XML>

⁷http://en.wikipedia.org/wiki/Document_Type_Definition

⁸<http://www.w3.org/XML/Schema>

- **Name:** Name of entity type, as discussed above
- **C++ Interface:** Name of C++ abstract interface classes related to the internal representation of the entity type
- **C++ Impl.:** Name of C++ implementation classes related to the internal representation of the entity type
- **.NET:** Name of .NET classes related to the entity type's .NET API (to be discussed in Chapter 10)
- **CUI:** Name of the command-line tool that can be used for managing the entity type (to be discussed in Chapter 11)
- **XML:** Name of XML instance documents containing a persistent representation of the entity type
- **XSD:** Name of the XML Schema Definition that represents the grammar of persistent representation of the entity type

In the table, asterisks do not represent pointers, but wildcards. Names containing an asterisk are therefore to be interpreted as name patterns.

Table 5.1: Tornado's Coherent Naming Scheme

Name	C++ Interface	C++ Impl.	.NET	CUI	XML	XSD
<i>Common</i>						
Build	Tornado::IBuild	Tornado::CBuild	Tornado.NET::CBuild	tbuild	N/A	N/A
Project	Tornado::IProject*	Tornado::CProject*	Tornado.NET::CProject*	tproject	*.Project.xml	Tornado.Project.xsd
Tornado	Tornado::ITornado	Tornado::CTornado	Tornado.NET::CTornado	tmain	*.Main.xml	Tornado.Main.xsd
<i>Experimentation Environment</i>						
Control	Tornado::IControls*	Tornado::CControls*	Tornado.NET::CControls*	tcontrols	*.Controls.xml	Tornado.Controls.xsd
Exp	Tornado::IExp*	Tornado::CExp*	Tornado.NET::CExp*	texp	*.Exp.xml	Tornado.Exp.xsd
Plot	Tornado::IPlot*	Tornado::CPlot*	Tornado.NET::CPlot*	tplot	*.Plot.xml	Tornado.Plot.xsd
<i>Modelling Environment</i>						
IconLib	Tornado::IIconLib*	Tornado::CIconLib*	Tornado.NET::CIconLib*	ticonlib	*.IconLib.xml	Tornado.IconLib.xsd
Layout	Tornado::ILayout*	Tornado::CLayout*	Tornado.NET::CLayout*	tlayout	*.Layout.xml	Tornado.Layout.xsd
ModelLib	Tornado::IModelLib*	Tornado::CModelLib*	Tornado.NET::CModelLib*	tmodellib	*.ModelLib.xml	Tornado.ModelLib.xsd

6

Materials and Methods

6.1 Introduction

In this chapter, the technological choices are discussed that were made with respect to the implementation of the Tornado kernel. When evaluating various alternatives for the implementation of an advanced kernel for modelling and virtual experimentation, two criteria are of major importance: performance and quality of code base. The performance criterion refers to the need for speed and scalability in the generation of executable model code and execution of virtual experiments. The quality of code base criterion on the other hand refers to a need for readability, modularity, maintainability, extensibility, re-usability and stability:

- **Readability** points to the fact that the code should be easy to understand by a broad audience. It should not contain cryptic constructs, and should be organized in a consistent manner.
- **Modularity** relates to the composition of software by using different well-defined units or modules.
- **Maintainability** refers to the ability to maintain the code base over a longer period of time and the ease with which it can be tailored to follow technological evolutions.
- **Extensibility** is a major issue and refers to the ability to easily add new modules (*e.g.*, new experiment types), thereby extending the functionality of the kernel.
- **Re-usability** is strongly related to extensibility, but more specifically refers to the fact that already existing modules should be usable in new contexts, thereby avoiding duplication of code.
- **Stability** is an important factor for any type of software, but is especially needed in the case of kernels since in these cases one cannot make any assumptions about the applications that will be built on top of it. It is therefore required for the software to be self-contained with respect to error detection and management.

A wide variety of programming languages exists nowadays, many of which are being used for scientific and technical computing. Most popular languages in this respect are FORTRAN, C, C++, Java and C#. Languages such as FORTRAN, C and C++ have a long history and can present an extensive

track record. Java and C# are more recent and are built around modern programming principles and paradigms. Table 6.1 contains a superficial comparison of these 5 languages on a number of key points:

- **Object-oriented programming:** Does the language support concepts such as encapsulation, object instantiation, inheritance and polymorphism?
- **Generic programming:** Does the language support generic programming constructs such as template classes and parameterized data types?
- **Virtual machine:** Is the language compiled to an intermediate representation which is then executed by a virtual machine (*cf.* Java) or common language run-time (*cf.* C#)?
- **Dynamic memory allocation:** Does the language have constructs that allow for memory to be allocated dynamically?
- **Automatic garbage collection:** Does the language provide for automatic clean-up of dynamically allocated memory when it is no longer referenced?

Table 6.1: Comparison of Programming Languages

Language	Object-oriented Programming	Generic Programming	Virtual Machine	Dynamic Memory Allocation	Automatic Garbage Collection
FORTRAN 77	No	No	No	No	No
FORTRAN 90	To some extent	No	No	Yes	No
C	No	No	No	Yes	No
C++	Yes	Yes	No	Yes	No
Java	Yes	Yes	Yes	Yes	Yes
C#	Yes	Yes	Yes	Yes	Yes

Given the fact that FORTRAN and C do not support object-oriented programming, most will agree that using these languages it will be more difficult to come to a quality code base. A much more debatable issue however is performance. Intuitively it would appear that languages that are compiled to machine code, rather than to an intermediate representation, will be considerably faster. When Java was first released by SUN in 1995, the generated byte-code was indeed fairly slow. Java could therefore, at that time, not be considered as a viable candidate language for scientific and technical computing. Recent advances in compiler technology (*e.g.*, JIT¹ - Just In Time - and AOT - Ahead Of Time - compilation) have however drastically improved the speed of the code generated by compilers for languages such as Java or C#. Whether or not, given these recent compiler improvements, Java and C# can in practice be used for scientific and technical computing has therefore become a heavily debated issue.

A fairly elaborate performance comparison of C/C++, Java and .NET has been done by Thomas Bruckschlegel of ToMMTi Systems and was later reworked and published in Dr. Dobbs, a well-known journal for software development practitioners². Table 6.2 lists the overall benchmark results for algorithmic and trigonometric code (the benchmark for non-algorithmic code that is also published by ToMMTi is less relevant for Tornado and is therefore not discussed). These overall benchmark results were composed on the basis of a high number of individual results from so-called micro-benchmarks, *i.e.*, small test problems each exhibiting a particular computational complexity. The platforms that were tested are the following:

¹http://en.wikipedia.org/wiki/Just-in-time_compilation

²<http://www.ddj.com/showArticle.jhtml?documentID=cuj0507bruckschlegel>

- IBM Java compiler and virtual machine
- SUN Java compiler and virtual machine
- Excelsior JET Java compiler and virtual machine
- INTEL C/C++ compiler (ICC)
- GNU C/C++ compiler (GCC)
- Microsoft Visual C/C++ compiler (MSVC)
- Microsoft C# compiler and common language run-time for .NET
- Mono C# compiler and common language run-time for .NET

For ICC and GCC on Linux, a distinction is made between native implementations of the STL library, and the STLPORT³ open-source implementation. Also, for SUN Java a distinction is made between the client and server virtual machines. From the results it can be seen that overall C/C++ yields better performance than Java/C#, although recent versions of Java and C# slowly seem to be closing the gap. The hardware platforms that were used to run tests are not mentioned here since only the relative numbers are of importance.

Table 6.2: Timing Results of the ToMMTi Micro-benchmark for Algorithmic and Trigonometric Code

Platform	Time (ms)
(linux) IBM Java 1.4.2	810.25
(linux) SUN Java 5 Server	1312.75
(linux) SUN Java 5 Client	1551.00
(linux) SUN Java 1.4.2 Server	1061.50
(linux) SUN Java 1.4.2 Client	1460.00
(linux) MONO 1.0.6	1886.00
(linux) ICC 8.1 + STLPORT 5.0	815.00
(linux) GCC 3.4 + STLPORT 5.0	652.50
(linux) ICC 8.1	587.50
(linux) GCC 3.4	650.00
(win32) IBM Java 1.4.2	836.00
(win32) JET 3.6	1218.75
(win32) SUN Java 5 Server	1050.75
(win32) SUN Java 5 Client	1351.75
(win32) SUN Java 1.4.2 Server	863.25
(win32) SUN Java 1.4.2 Client	1386.50
(win32) MS .NET 2.0 Beta 2	890.25
(win32) MS .NET 1.1	1097.25
(win32) MS VC7.1	761.50

A benchmark such as the ToMMTi micro-benchmark provides some insight into performance ratios, but for specific applications these ratios can possibly differ substantially. Actually, for evaluating technologies in the scope of an application that is to be built, the only truly accurate benchmark would be the application itself. Since this evidently leads to a vicious circle, it may be advisable to establish a specific benchmark for an application at hand, including some constructs that are considered typical for that application. Since one of the most time-consuming parts of the Tornado kernel is the integration of

³<http://www.stlport.org>

systems of differential equations, it seemed most appropriate to devise a small program that integrates an ODE system, and implement it on different platforms. In fact, in order to favor simplicity, the program performs forward Euler integration (using the following scheme: $x_{i+1} = x_i + hf_i$) of a system consisting of two equations (*i.e.*, $dx_1/dt = -x_2$; $dx_2/dt = x_1$). Listing 6.1 shows the C# implementation of the program.

Listing 6.1: Forward Euler Integration of a Small ODE System in C#

```
using System;

namespace TestPerf
{
    class TestPerf
    {
        static double[] ComputeF(double t, double[] x)
        {
            int n = x.Length;
            double[] f = new double[n];

            f[0] = -x[1];
            f[1] = x[0];

            return f;
        }

        static void Euler(double h, double t0, double tf, double[] x0)
        {
            int i;
            int n = x0.Length;

            double t;
            double[] x = new double[n];
            double[] f = new double[n];

            t = t0;
            x = x0;

            while (t <= tf)
            {
                f = ComputeF(t, x);

                for (i = 0; i < n; i++)
                    x[i] = x[i] + h * f[i];

                t += h;
            }
        }

        static void Main(string[] args)
        {
            try
            {
                DateTime t1 = DateTime.Now;

                double[] x0 = new double[2];

                x0[0] = 1;
                x0[1] = 0;

                Euler(1e-5, 0, 2000 * 3.1415927, x0);

                DateTime t2 = DateTime.Now;

                TimeSpan Duration = t2 - t1;
                System.Console.WriteLine("Total processing time: " + Duration.Seconds);
            }
            catch (System.Exception Ex)
            {
                System.Console.WriteLine(Ex.ToString());
            }
        }
    }
}
```

```

    }
  }
}

```

The timing results of executing implementations of Listing 6.1 for various languages on a Windows machine are listed in Table 6.3. The following platforms were tested:

- Microsoft Visual C++ 8.0
- Microsoft Visual C 8.0
- Microsoft C# for .NET 2.0
- SUN Java 1.6.0 with client virtual machine
- SUN Java 1.6.0 with server virtual machine
- Active Tcl 8.4.14

Next to platform information and timing results, Table 6.3 also contains the following items:

- **Vector data type:** Specifies the data type that was used to represent real-valued vectors. For MSVC++, `std::vector<double>` refers to the template-based vector class that is part of the STL library; `Common::CVectorDouble` is a custom class that was implemented through the Common library that is part of Tornado (to be discussed later).
- **Indexing operator:** The operator that is used to retrieve a vector element.
- **Automatic garbage collection:** Indicates whether the instances of the vector data type are automatically cleaned up.
- **Index check:** Indicates whether a range check is being performed by the indexing operator on the index argument.
- **Resizable:** Indicates whether the vector can be resized after instantiation.
- **Dynamic insertion:** Indicates whether vector elements can be inserted and/or removed from the vector.

The results confirm that interpreted languages such as Tcl are evidently very unsuitable candidates for the type of code under consideration. Speed of execution is about 1,000 times slower than the speed of execution of the compiled languages that were studied. From these compiled languages, C is the fastest, followed very closely by an implementation in C++ where a combination of the `Common::CVectorDouble` and `double[]` data types were used. The exact nature of the `Common::CVectorDouble` data type will be discussed later. Another conclusion that can be drawn from the results is that for C++, the timing variability with respect to data types is substantial. Some data types lead to fast execution, others (especially those that are related to STL) are relatively slow.

In general, we can conclude from the results of Table 6.2 and Table 6.3 that C and C++ are good choices when performance is concerned, provided wise choices are made regarding the data type that is used to represent real-valued vectors and the policies that are used with respect to index range checking. Since from these two languages, C++ is certainly the better in view of the potential to establish a quality code base, this language was chosen as main implementation language for the Tornado kernel.

One might wonder whether a trivial program as in Listing 6.1 is representative for the type of processing that is done by a simulation kernel. Evidently, a larger program that uses a more complex integration

Table 6.3: Timing Results of a Simulation Benchmark Program

No.	Platform	Vector Data Type	Indexing Operator	Garbage Collection	Index Check	Resizable	Dynamic Insertion	Time (ms)
1	(win32) MS VC++ 8.0	std::vector<double>	.at()	Yes	Yes	Yes	Yes	58
2	(win32) MS VC++ 8.0	std::vector<double>	[]	Yes	No	Yes	Yes	48
3	(win32) MS VC++ 8.0	Common::CVectorDouble	.at()	Yes	Yes	Yes	No	53
4	(win32) MS VC++ 8.0	Common::CVectorDouble	[]	Yes	Yes	Yes	No	53
5	(win32) MS VC++ 8.0	Common::CVectorDouble	[]	Yes	No	Yes	No	29
6	(win32) MS VC++ 8.0	Common::CVectorDouble	..[]	Yes	No	Yes	No	20
7	(win32) MS VC++ 8.0	CVectorDouble + double[]	..[]	Yes	No	Yes	No	13
8	(win32) MS VC 8.0	double[]	[]	No	No	No	No	12
9	(win32) MS C# .NET 2.0	List<double>	[]	Yes	Yes	Yes	Yes	54
10	(win32) MS C# .NET 2.0	double[]	[]	Yes	Yes	No	No	16
11	(win32) SUN Java 1.6.0 Client	double[]	[]	Yes	Yes	No	No	24
12	(win32) SUN Java 1.6.0 Server	double[]	[]	Yes	Yes	No	No	20
13	(win32) Active Tcl 8.4.14	list	lindex	Yes	Yes	Yes	Yes	25,300

algorithm and a more elaborate model could be considered more interesting, but would also require more time to port to different platforms. One argument for the fact that the small program that was used actually *is* representative, is the fact that during the implementation of the Tornado simulation kernel in C++, many tests were performed with different model sizes and different real-valued vector implementations. All of the results that were obtained confirm the conclusions that can be drawn from Table 6.3. We can therefore consider the program to be representative.

6.2 Object-oriented Programming

In the prequel, the object-oriented programming paradigm has been referred to a number of times. Before continuing the rest of the discussion on technological choices, an overview will be given on some terminology and key concepts that are important with regard to object-orientation.

Object-oriented programming (OOP)⁴ (Meyer, 1997) is a programming paradigm that uses objects and their interactions to design applications and computer programs. It is based on several techniques, including inheritance, polymorphism and encapsulation, and was not commonly used in mainstream software application development until the early 1990's. At this moment however, many modern programming languages support OOP.

Object-orientated programming roots back to the 1960's, when the nascent field of software engineering had begun to discuss the idea of a software crisis. As hardware and software became increasingly complex, researchers studied how quality code could be maintained. Object-oriented programming was deployed to address this problem by strongly emphasizing modularity and re-usability. The Simula⁵ simulation language was the first to introduce the concepts underlying object-oriented programming as a superset of the ALGOL⁶ language. The first programming language that could truly be called object-oriented was Simula-67⁷.

Object-oriented programming may be seen as a collection of cooperating objects, as opposed to a traditional view in which a program can be regarded as merely a list of instructions to be executed. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects.

⁴http://en.wikipedia.org/wiki/Object-oriented_programming

⁵<http://en.wikipedia.org/wiki/Simula>

⁶<http://en.wikipedia.org/wiki/ALGOL>

⁷<http://en.wikipedia.org/wiki/Simula>

By virtue of its strong emphasis on modularity, object-oriented code is intended to be simpler to develop and easier to understand later on, lending itself to more direct analysis, coding and understanding of complex situations and procedures than less modular programming methods.

As mentioned before, the OOP paradigm is based on classes and objects:

- A **class** has **methods** and **data members**
- An **object** is an **instance** of a class
- An object's **state** is the set of **values** of its data members

Key concepts, found in the majority of object-oriented languages are:

- **Encapsulation** conceals the details of a class from the objects that interact with it. For each type of interacting object, the class can decide on the basis of interface descriptions which interactions are allowed and which are not.
- **Inheritance** is related to the concept of sub-classes, which are more specialized versions of a base class, inheriting attributes and behavior from the base class as well as introducing their own.
- **Abstraction** is simplifying complex reality by modelling classes appropriate to a problem, and working at the most appropriate level of inheritance for a given aspect of the problem.
- **Polymorphism** is the ability of objects belonging to different data types to respond differently to method calls with the same name, depending on the data type of the arguments.

6.3 Advanced C++

In 1980 Bjarne Stroustrup developed a new programming language that he named C++, because it extends C with object-oriented features (Stroustrup, 1997). Although C++ has been around for a number of years that can be considered an eternity in the scope of software engineering, it is surprising to see how much C++ code is still being developed where many of those features that make the language what it truly is (instead of a mocked up version C) are being neglected. Especially in the area of scientific and technical computing this is unfortunately quite apparent. In some cases, particular features of the C++ language are not used for good reasons (*e.g.*, performance), but most frequently the reason for not using C++ as it was intended is simply a lack of insight and/or insufficient background in software engineering.

Amongst those who have received a formal training in computer science, there is a growing resentment against C++, for it is considered old-fashioned compared to more recent languages such as Java and C#. Several features such as automatic garbage collection and the support for data types such as strings and vectors, which are commonplace in today's languages, are indeed lacking from the original specification of C++.

It is our belief that although it has been around for a very long time, C++ is still a very good option for many types of applications, especially those that require complex data types and high performance. Moreover, since its inception, a number of extensions have been proposed that, if well-adopted, allow for C++ to be used in a manner similar to Java and C#. These extensions have been proposed in the scope of Standard C++ and STL, the Standard Template Library. Below, a number of features are discussed that in practice prove to be very useful but, in our experience, appear not to be known by a surprisingly large number of developers. These features are either part of the original C++ specification, or the Standard C++ and STL specifications.

6.3.1 Coding Style

As the remainder of this section contains a number of code fragments, a number of coding style conventions that are used in this document (as well as throughout the entire Tornado code base) are given.

Table 6.4 lists naming conventions that are used for identifiers. Since the Tornado naming system is based on prefixes, it vaguely resembles the system that is known as Hungarian notation⁸. However, the Tornado system is much simpler and does not include data type information through prefixes.

Table 6.4: Naming Conventions for Identifiers in C++ Code

Item	Naming Example
Local variable	MyLocalVar
Global variable	g_MyGlobalVar
Member variable	m_MyMemberVar
Static member variable	s_MyStaticMemberVar
Pointer	pMyPointer
Interface class	IMyInterface
Implementation class	CMyClass
Simple type	TMyType

6.3.2 Standard Library

In C++, the Standard Library is a collection of classes and functions that are written in the core language. The C++ Standard Library also incorporates the ISO C90 Standard Library. The Standard Template Library (STL) is a subset of the C++ Standard Library that contains containers, algorithms, iterators, function objects, *etc.* Often the terms “STL” and “C++ Standard Library” are used interchangeably, although this is not fully correct.

Below, a selected number of classes from the C++ Standard Library are discussed. All of these are heavily used throughout the Tornado code base and - in our experience - are often not known to developers of modelling and simulation software.

Strings

Originally, working with strings in C++ was cumbersome, as only the original C-based mechanism was available. The C++ Standard Library however implements two classes (*std::string* for single-byte strings and *std::wstring* for double-byte strings) that allow for strings to be used as in many other languages that have a better reputation with respect to string use. Listing 6.2 presents an example.

Listing 6.2: Example of Strings in C++

```
using namespace std;

// ...

wstring s1 = L"This is ";
wstring s2 = L"a test ";

wcout << s1 << L"_" << s2 << endl;

wstring s = s1 + L"_" + s2;

wcout << s << endl;
```

⁸http://en.wikipedia.org/wiki/Hungarian_notation

Smart Pointers

Smart pointers (also known as automatic pointers) are classes that simulate pointers while providing additional features, such as automatic garbage collection. The `std::auto_ptr` class that is part of the C++ Standard Library is a smart pointer that takes ownership of the memory that is pointed to by a pointer. When an instance of the `std::auto_ptr` class goes out of scope, the instance's destructor will ensure that memory that the guarded pointer points to is released. Listing 6.3 shows two code fragments: one that does not use smart pointers, and one that does. In the first fragment memory must be deleted explicitly, the second fragment is more elegant and more safe in this respect.

Listing 6.3: Example of Smart Pointers in C++

```
// Not using smart pointers:

{
    CMyClass* pObject = new CMyClass;

    // ...

    throw CEx();

    // ...

    delete pObject; // No cleanup, memory leak!
}

// Using smart pointers:

using namespace std;

{
    auto_ptr<CMyClass> pObject(new CMyClass);

    // ...

    throw CEx();

    // ...

} // Automatic cleanup!
```

Vectors

The generic vector class that is part of STL allows for vectors to be created of any data type. These vectors can be dynamically resized and elements can be removed or inserted at will. STL vectors are very elegant, but unfortunately not very efficient, as can be seen in Table 6.3. Care must therefore be taken only to use STL vectors in code fragments that are not time-critical. Listing 6.4 shows a small example of the application of real-valued STL vectors.

Listing 6.4: Example of Vectors in C++

```
using namespace std;

// ...

vector<double> v;

v.push_back(10); v.push_back(20);
v.push_back(30); v.push_back(40);

for (int i = 0; i < v.size(); i++)
    cout << v[i] << endl; // Prints 10 20 30 40

v.pop_back();
```

```
for (int i = 0; i < v.size(); i++)  
    cout << v[i] << endl;    // Prints 10 20 30
```

Dictionaries

A dictionary (also known as associative array, map or lookup table) is an abstract data type that is composed of a collection of keys and a collection of values, where each key is associated with one value. The generic map class that is part of STL allows for dictionaries to be created using data types of choice for keys and values. Typically, strings are used as keys. STL maps are elegant and also fairly efficient, taking into account the potential inherent complexity of lookup operations. Listing 6.5 presents an example of an STL map using the *std::wstring* data type as key, and the *double* data type as value.

Listing 6.5: Example of Dictionaries in C++

```
using namespace std;  
  
// ...  
  
typedef map<wstring, double> TMap;  
TMap m;  
  
m[L"one"] = 1;  
m[L"two"] = 2;  
m[L"three"] = 3;  
  
for (TMap::iterator it = m.begin(); it != m.end(); it++)  
    wcout << it->first << L" = " << it->second << endl;    // Prints one = 1 etc.
```

6.3.3 Namespaces

In general, a namespace is an abstract container providing context for the items it holds and allowing disambiguation for items with the same name (residing in different namespaces). Listing 6.6 shows how a class with the same name can be declared twice through the use of two different namespaces.

Listing 6.6: Example of Namespaces in C++

```
namespace Foo  
{  
    class CMyClass {}  
}  
  
namespace Bar  
{  
    class CMyClass {}  
}  
  
// ...  
  
Foo::CMyClass MyClass1;  
Bar::CMyClass MyClass2;
```

6.3.4 Exceptions

In the discussion on design principles it was already pointed out that exceptions play an important role in ensuring the stability of software. Listing 6.7 gives an example of the use of exception classes in C++. Important to note is that exception classes can use data members to store additional information on the exception that has occurred. Also, catch clauses can be used to filter out those types of exceptions that can be handled, while ignoring the others.

Listing 6.7: Example of Exceptions in C++

```

using namespace std;

class CEx
{
public:
    CEx(const wstring& Message) :
        m_Message(Message) {}

public:
    wstring GetMessage() const { return m_Message; }

protected:
    wstring m_Message;
};

class CMyClass
{
public:
    void DoIt(double x)
    {
        // ...
        if (x < 10)
            throw CEx(L"x is smaller than 10");
        // ...
    }
};

int main()
{
    try
    {
        CMyClass MyClass;
        // ...
        MyClass.DoIt(5);
        // ...
    }
    catch (const CEx& Ex)
    {
        wcout << Ex.GetMessage() << endl;
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}

```

6.3.5 Streaming Operators

A streaming operator is an overloaded operator that writes out the internal state of an object in a human-readable fashion. Evidently, streaming can also be accomplished by adding a method that provides for this to the object's interface, but through the use of overloaded operators, the process becomes especially elegant. Listing 6.8 shows how the << operator can be overloaded to provide for streaming of a custom data type.

Listing 6.8: Example of Streaming Operators in C++

```

using namespace std;

class CMyClass
{
public:
    wstring& Str() { return m_Str; }
}

```

```
vector<double>& Vector() { return m_Vector; }

protected:
    wstring m_Str;
    vector<double> m_Vector;
};

wostream& operator<<(wostream& OS, CMyClass& MyClass)
{
    OS << MyClass.Str() << endl;

    for (int i = 0; i < MyClass.Vector().size(); i++)
        OS << MyClass.Vector()[i] << " ";

    return OS;
}

int main()
{
    CMyClass MyClass;

    MyClass.Str() = L"This is a test";
    MyClass.Vector().push_back(10);
    MyClass.Vector().push_back(20);
    MyClass.Vector().push_back(30);

    wcout << MyClass << endl;
}
```

6.3.6 Interfaces

Interfaces are descriptions of the communication boundaries between two entities. They refer to an abstraction that an entity provides of itself to the outside world. In contrast to languages such as Java and C#, C++ does not have a specific keyword that can be used in interface declarations. In C++, interfaces are to be represented as abstract base classes. Abstract base classes are classes that do not have data members and only have pure virtual methods, *i.e.*, methods without implementation that need to receive their implementation through derived classes. Listing 6.9 gives an example.

Listing 6.9: Example of Interfaces in C++

```
using namespace std;

class IShape
{
public:
    double GetSurface() = 0;
};

class CSquare : public virtual IShape
{
public:
    double GetSurface()
    {
        return m_Length * m_Length;
    }

    double m_Length;
};

class CCircle : public virtual IShape
{
public:
    double GetSurface()
```

```

{
    return m_Radius * m_Radius * 3.1415927;
}

double m_Radius;
};

int main()
{
    auto_ptr<IShape> pShape = new CSquare;

    // Remaining code is independent of shape implementations

    cout << pShape->GetSurface() << endl;
}

```

The definition of interfaces should be the first step in any design, and should in any case precede the definition of implementation classes. As far as possible, application code should use interfaces instead of implementation classes.

6.4 XML

As mentioned before, the Tornado design is centered around the use of XML for the persistent representation of entities. The XML grammars that are used are expressed as XML Schema Definitions and very closely mimic the internal representation of the respective entities. Also, XML grammars are independent of the properties (*i.e.*, dynamically queryable configuration options) that these entities support, as these properties are merely considered as name-value pairs. For properties, a grammar as in Listing 6.10 is used. A graphical representation is in Figure 6.1.

Listing 6.10: Props XML Schema Definition

```

<xs:element name="Props">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Prop" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:attribute name="Name" type="xs:string" use="required"/>
          <xs:attribute name="Value" type="xs:string" use="optional"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

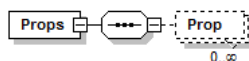


Figure 6.1: Props XML Schema Definition

Since XML is a text-based format, programmatically generating XML from the internal representation of an object is easy. All it requires is an appropriate streaming operator. However, creating an internal representation of an object from its XML-based persistent representation is less trivial. For this, a XML parser is needed. Traditionally, two types of parsing strategies for XML have been used: SAX and DOM.

SAX (Simple API for XML) is a lexical, event-driven interface in which a document is read serially and its contents are reported as callbacks to various methods on a custom handler object. SAX is fast and efficient to implement, but difficult to use for extracting information at random from the XML, since

it tends to burden the application author with keeping track of which part of the document is being processed. It is better suited to situations in which certain types of information are always handled the same way, no matter where they occur in the document.

DOM (Document Object Model) is an interface-oriented API that allows for navigation of the entire document as if it were a tree of node objects representing the document's contents. DOM implementations tend to be memory intensive, as they generally require the entire document to be loaded into memory and to be constructed as a tree of objects before access is allowed.

Given the ease of implementation of SAX parsers and the flexibility that they allow with respect to the implementation of permissive parsing schemes, all Tornado parsers were constructed on the basis of the SAX API.

6.5 Third-party Software Components

One of the requirements for the Tornado kernel was to restrict the use of third-party software components to a minimum. In the end, only two third-party components were used in the kernel code base itself (excluding the source code for the various API's): OpenSSL and OpenTop.

The **OpenSSL**⁹ project is a collaborative effort to develop a robust, commercial-grade, full-featured open source toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols as well as a full-strength general-purpose cryptography library. The project is managed by a world-wide community of volunteers that use the Internet to communicate, plan and develop the OpenSSL toolkit and its documentation. Tornado only uses OpenSSL in its Common Library. The Encryption module of this library relies on some of the cryptography features of OpenSSL to perform encryption of sensitive model libraries.

The **OpenTop** library is a commercial C++ library that is available from Elcel Technology¹⁰ and is aimed at the development of web services. The Tornado kernel uses OpenTop for the creation and synchronization of threads, as well as for XML parsing. Other features of the OpenTop library (that are not used by Tornado at this point) include resource management, platform abstraction, networking and support for Unicode, various types of I/O, and the SSL and HTTP protocols.

6.6 Development Environment

Although the source code of the Tornado kernel is platform-independent and build procedures for Windows and Linux are distributed with the source code (build procedures for other platforms could also be set up if desired), the development process itself solely took place on the Windows platform. More specifically, the Microsoft Visual Studio 2005¹¹ IDE (Integrated Development Environment) was adopted. This platform offers a high degree of productivity and includes source code editors with syntax highlighting¹² and code completion¹³, debuggers, project browsers, *etc.* Visual Studio 2005 is entirely .NET-capable and supports the notions of solutions and projects. A project is a set of related files that allow for building a software component using a particular programming language. A solution contains a set of projects. The projects used in a solution do not necessarily have to be related to the same programming language.

On the Windows platform, the Tornado source code is built on the basis of Microsoft Visual Studio solution files (*.sln) and project files (*.vcproj for C++ and *.csproj for C#). From these, makefiles could

⁹<http://www.openssl.org>

¹⁰<http://www.elcel.com>

¹¹<http://www.microsoft.com/vstudio>

¹²http://en.wikipedia.org/wiki/Syntax_highlighting

¹³http://en.wikipedia.org/wiki/Code_completion

potentially be generated, but this procedure has not been used in the scope of Tornado. On the Linux platform, the GNU C++ compiler (g++) in conjunction with plain, hand-written makefiles is adopted.

For all XML-related work, such as designing XML Schema Definitions and visualizing XML instance documents, the Altova XMLSpy¹⁴ IDE was used. XMLSpy is the industry-standard XML editor and development environment for modelling, editing, transforming and debugging XML-related technologies. It offers a powerful XML editor, a graphical schema designer, a code generator, file converters, debuggers, profilers, full database integration, support for XSLT, XPath, XQuery, WSDL, SOAP, and Office Open XML documents, plus Visual Studio .NET and Eclipse plug-ins.

¹⁴<http://www.altova.com>

7

Common Library

7.1 Introduction

The Common Library is a library that contains classes and functions for platform-abstraction and convenience. It was developed specifically for use by the Tornado kernel and related interfaces and applications. The Common Library is generic in the sense that none of the features it implements are related to modelling and virtual experimentation as such.

All processing that is platform-specific in the Tornado framework is implemented through the Common Library. In this way, the rest of the Tornado framework code base could be kept free from platform-specific statements, which evidently favors maintainability and readability. On the technical level, the Tornado framework code outside of the Common Library does not contain any platform-dependent conditional preprocessing statements, such as *ifdef*'s.

Large software projects are regularly built on top of libraries similar to the Common Library. For instance, the TAO¹ CORBA implementation was done on top of the ACE² library (actually, TAO stands for The ACE ORB). ACE is a very extensive library for adaptive communication environments that is also used in other software projects.

Below is an overview of the most relevant modules of the Tornado Common Library.

7.2 Encryption

The Encryption module contains a convenience class that is based on the OpenSSL library and allows for the encryption and decryption of text buffers. In Tornado this class is mainly used for the encryption and decryption of model libraries that contain sensitive information and therefore need to be protected.

Encryption is done using the Triple Data Encryption Algorithm (TDEA), commonly known as Triple DES³. Triple DES attempts to strengthen security through the threefold application of DES encryption to the same message, using different keys, *i.e.*, $DES(k_3, DES(k_2, DES(k_1, M)))$, where k_1 , k_2 and k_3

¹<http://www.cs.wustl.edu/~schmidt/TAO.html>

²<http://www.cs.wustl.edu/~schmidt/ACE.html>

³http://en.wikipedia.org/wiki/Triple_DES

are DES keys and M is the message to be encrypted. In general, Triple DES has a total key length of 168 bits, as each individual key has a length of 56 bits.

7.3 Dynamically-loaded Libraries

Both the Windows and the Linux platform support dynamic loading of binary code. On the Windows platform such dynamically-loadable modules are known as Dynamically-Loadable Libraries (DLL's), on Linux they are called Shared Objects (*so's*). Although the mechanisms that support both approaches are fairly different (and hence also the API's that are to be used), their practical use from an application developer's point of view is similar.

The Common Library contains a class that allows for loading dynamically-loadable modules and for retrieving symbols (functions, variables, *etc.*) from these modules. The class hides the platform-specific details of these operations. On Windows operations are implemented through the win32 API, while on Linux the *dlfcn* library is used.

In Tornado, dynamically-loadable modules are used for executable models and for numerical solver algorithms. Both types of items are therefore binary components that can be loaded at run-time through the platform-abstraction class provided by the Common Library.

7.4 Exceptions

It was mentioned in Chapter 5 that the use of exception classes is beneficial for the stability of software. In Tornado, error-handling is fully based on exceptions, although the approach that is used is fairly simple.

The Common Library implements an exception base class and a number of derived classes, as represented in Figure 7.1. Through a number of data members, the base class is capable of storing information on the source of the error condition (source file name and line number) as well as textual information. Two text messages are foreseen: one that is standard and is filled in through the constructor of derived classes (*e.g.*, the *CExAccess* class would store the string “*Access violation*” in this field), and one that can be used to store any additional information. The fact that an inheritance hierarchy is used for exception classes allows for filtering through selective catch-clauses.

7.5 Strings

The Strings module of the Common Library contains a number of convenience functions that further facilitate the use of Standard C++ single-byte and wide-character strings. The functionality that is offered through these functions is mainly related to the following:

- Conversion from wide-character strings to single-byte strings and *vice versa*
- Splitting and joining of strings on the basis of a configurable sub-string
- Conversion of strings to lower and upper case
- Trimming, *i.e.*, removal of leading and trailing white-space from strings
- Finding and replacing sub-strings

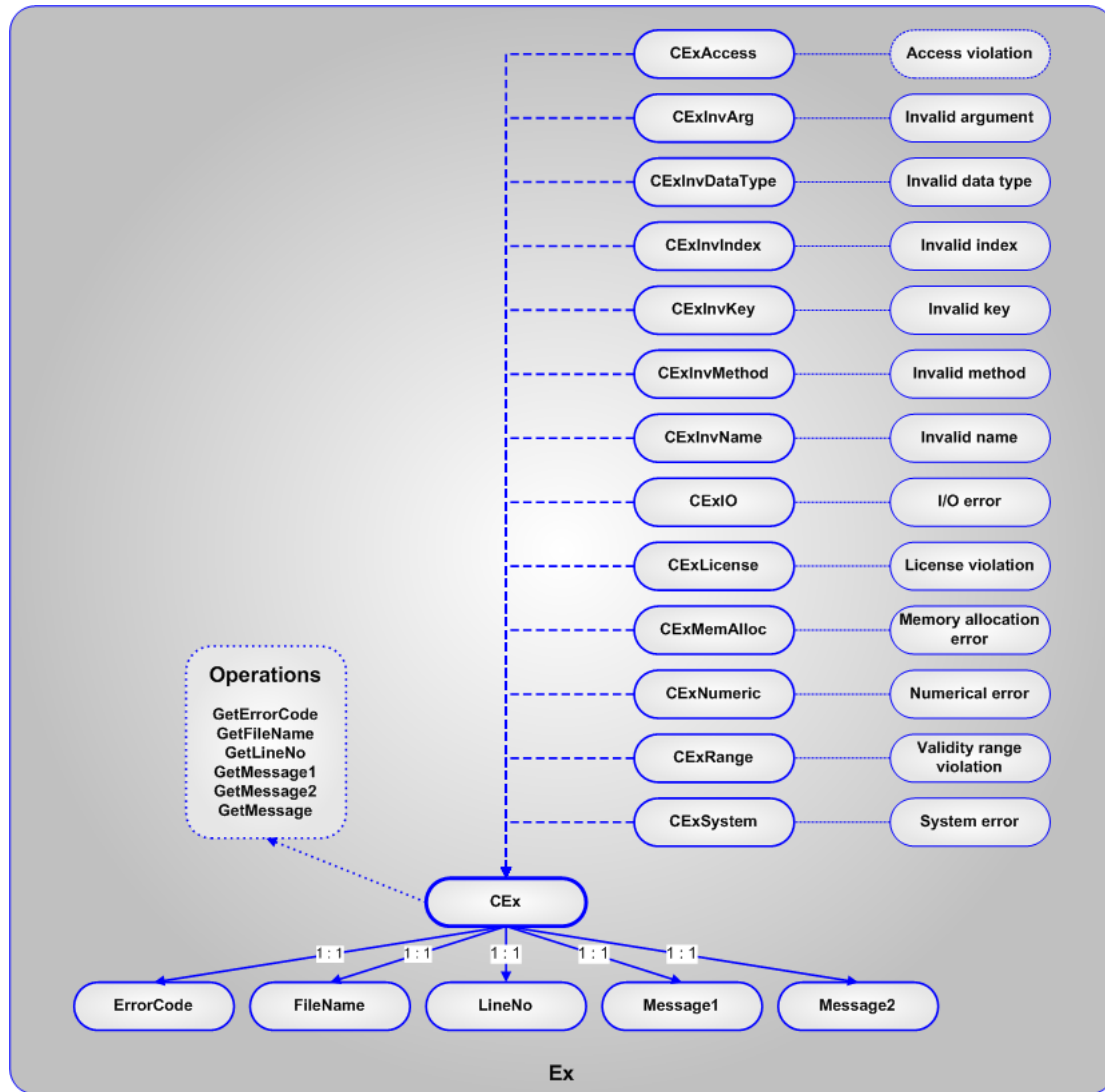


Figure 7.1: Common Library: Exception Classes

7.6 Properties

The Common Library implements a mechanism for representing dynamically-queryable properties. The information that is contained in each property is listed in Table 7.1. The *Common::TDataType* and *Common::TAccessType* data types are enumerations. The *CValue* data type is a class that mainly consists of a union in which containers for string, real, integer, boolean and binary data items are stored as overlays.

As can be seen from Figure 7.2, a properties structure (*Props*) is actually a pair that consists of a map of values (*ValueMap*) and a map of meta-information structures (*PropsInfo*). The value map is a collection of values that can be accessed through indices and names. The map of meta-information structures can also be accessed through indices and names and contains static meta-information for each corresponding item from the value map.

The properties mechanism that is implemented through the Common Library is a very valuable asset for the Tornado source code base since it alleviates the need for individual accessor and mutator methods for each entity configuration option. Entities that support the properties interface, normally establish

Table 7.1: Property Data Members

Name	Type	Description
Name	std::wstring	Property name
DataType	Common::TDataType	Boolean, Integer, Real, String or Binary
AccessType	Common::TAccessType	Read-only or Read/Write access
Desc	std::wstring	Textual description
Group	std::wstring	Name of group the property belongs to
Hidden	std::wstring	Indicates whether the property is for internal use only
Default	Common::CValue	Default value
Range	std::pair<CValue, CValue>	Lower bound and upper bound
Value	Common::CValue	Actual value

their meta-information structure once in their constructor. Afterwards this information can be queried and values can be retrieved or modified.

7.7 Manager

The Manager class is a convenience template class that implements a map with the data type for keys set to string, and the value data type set to a generic pointer. As string-based maps are frequently occurring throughout the Tornado code base, this class provides more conciseness and readability. Figure 7.3 shows that a Manager is actually a collection of *Key*, *ObjectPtr* pairs. The collection can be accessed through indices and keys.

7.8 Mathematics

The Mathematics module of the Common Library consists of a large number of mathematical functions that are typically needed in modelling and virtual experimentation software, but are not restricted to this type of software. Most important function categories are the following:

- Aggregation functions
- Sampling from statistical distributions
- Interpolation
- Evaluation of error criteria

7.8.1 Aggregation Functions

The functions in this category are mostly simple and generate aggregated values from one-dimensional and two-dimensional sets of data.

One-dimensional Data Sets

- **Min:** Finds the minimum of a set y of n values.

$$Min_y = \min_i y_i \quad (7.1)$$

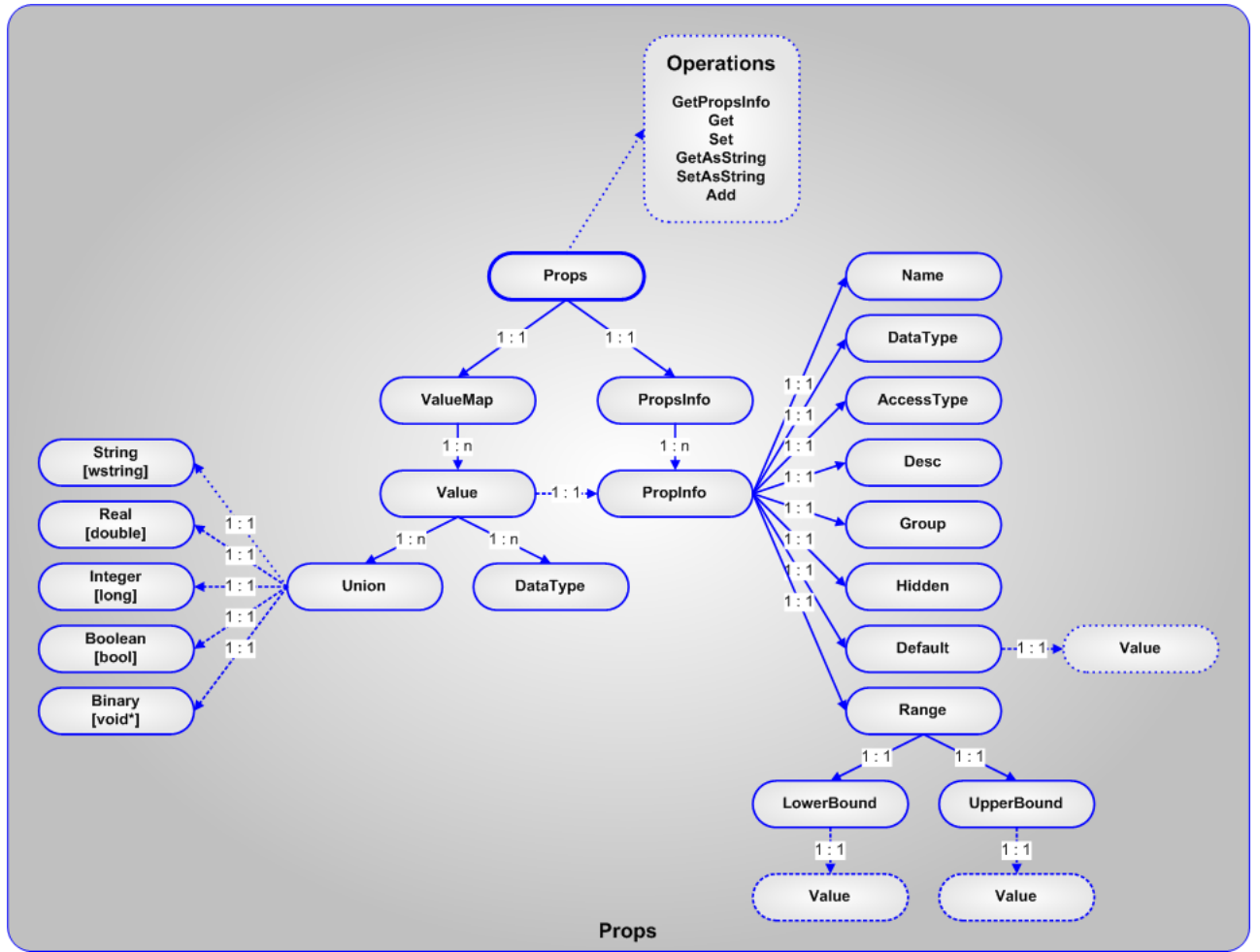


Figure 7.2: Common Library: Properties Class

- **Max:** Finds the maximum of a set y of n values.

$$Max_y = \max_i y_i \quad (7.2)$$

- **Mean:** Computes the mean of a set y of n values.

$$\mu_y = \frac{1}{n} \sum_{i=1}^n y_i \quad (7.3)$$

- **StdDev:** Computes the standard deviation⁴ of a set y of n values.

$$\sigma_y = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \mu_y)^2} \quad (7.4)$$

- **Median:** Finds the median⁵ of a set y of n values. The median of a finite set can be found by arranging all the observations from lowest to highest ($y_s \equiv \text{Sort}(y)$) and, if the number of values

⁴http://en.wikipedia.org/wiki/Standard_deviation

⁵<http://en.wikipedia.org/wiki/Median>

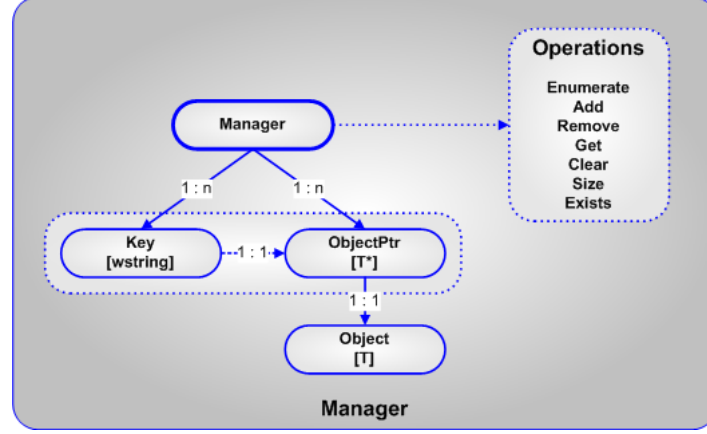


Figure 7.3: Common Library: Manager Class

is odd, picking the middle one. In case the number of values is even, the mean of the two middle values is taken.

$$Median_y = \begin{cases} y_{s,j} & \text{if } n \text{ is odd and } j = \frac{n+1}{2} \\ \frac{1}{2}(y_{s,j} + y_{s,j+1}) & \text{if } n \text{ is even and } j = \frac{n}{2} \end{cases} \quad (7.5)$$

- **Percentile:** Computes the p -th percentile⁶ of a set y of n values. The p -th percentile can be found by arranging all the observations from lowest to highest ($y_s \equiv \text{Sort}(y)$) and picking the value for which $p\%$ of the values are lower or equal. The median of a set of values is equal to the 50th percentile.

$$P_{y,p} = y_{s,j'} + (j - j')(y_{s,j'+1} - y_{s,j'}) \quad \text{where } j = n \times \frac{p}{100} \text{ and } j' = \text{floor}(j) \quad (7.6)$$

- **Skewness:** Computes the skewness⁷ of a set y of n values. Skewness is a measure of the asymmetry of the probability distribution of a real-valued random variable. Skewness is the third standardized moment and is defined as $\gamma_1 = \frac{\mu_3}{\sigma^3}$. For a sample of n values, the sample skewness is:

$$Skewness_y = \frac{\sqrt{n} \sum_{i=1}^n (y_i - \mu_y)^2}{(\sum_{i=1}^n (y_i - \mu_y)^2)^{\frac{3}{2}}} \quad (7.7)$$

- **Kurtosis:** Computes the kurtosis⁸ of a set y of n values. Kurtosis is a measure of the peakedness of the probability distribution of a real-valued random variable. Higher kurtosis means more of the variance is due to infrequent extreme deviations, as opposed to frequent modestly-sized deviations. Kurtosis is the fourth standardized moment and is defined as $\gamma_2 = \frac{\mu_4}{\sigma^4} - 3$. For a sample of n values, the sample kurtosis is:

$$Kurtosis_y = \frac{n \sum_{i=1}^n (y_i - \mu_y)^4}{(\sum_{i=1}^n (y_i - \mu_y)^2)^2} - 3 \quad (7.8)$$

⁶<http://en.wikipedia.org/wiki/Percentile>

⁷<http://en.wikipedia.org/wiki/Skewness>

⁸<http://en.wikipedia.org/wiki/Kurtosis>

- **MinPos:** Returns the position in the set y of n values that corresponds with the minimum value Min_y . In case several values are eligible, the first one is returned.

$$MinPos_y = j \quad \text{where } y_j = Min_y \quad (7.9)$$

- **MaxPos:** Returns the position in the set y of n values that corresponds with the maximum Max_y . In case several values are eligible, the first one is returned.

$$MaxPos_y = j \quad \text{where } y_j = Max_y \quad (7.10)$$

- **MeanPos:** Returns the position in the set y of n values that corresponds most closely to the mean μ_y . In case several values are eligible, the first one is returned.

$$MeanPos_y = j \quad \text{where } y_j = \min_i |y_i - \mu_y| \quad (7.11)$$

Two-dimensional Data Sets

- **Int:** Computes an approximation of the integral of y over x , where both are vectors of n elements. Two approximations are implemented, one that assumes a zero-hold behavior in between data points (Euler) and one that assumes a straight line (Trapezium rule).

$$Int_{x,y}^{Euler} = \sum_{i=1}^{n-1} y_i (x_{i+1} - x_i) \quad (7.12)$$

$$Int_{x,y}^{Trapezium} = \sum_{i=1}^{n-1} \frac{y_{i+1} + y_i}{2} (x_{i+1} - x_i) \quad (7.13)$$

- **Mean:** Computes an approximation of the mean of y over x , where both are vectors of n elements. The mean is computed by dividing the integral by the horizon of x .

$$Mean_y = \frac{Int_{x,y}}{x_n - x_1} \quad (7.14)$$

7.8.2 Sampling from Statistical Distributions

In Tornado, sampling from statistical distributions is mainly required in the scope of the Scenario Analysis and Monte Carlo Analysis experiment types, as well as for an input generator type that provides random values. All methods can operate in two modes: either they work on the basis of a uniformly distributed random number between 0 and 1 that is supplied by an external source and is passed as a parameter, or they internally generate such a value. Also, for some types of distributions truncation can be applied, *i.e.*, these distribution methods are forced to return a value between a user-specified minimum and maximum.

Uniform Distribution

Below are the Probability Density Function (PDF) and Cumulative Distribution Function (CDF) of the uniform distribution implemented by the Mathematics module. Parameters are a (minimum) and b (maximum).

$$f(x; a, b) = \begin{cases} \frac{1}{b-a} & a \leq x \leq b \\ 0 & x < a \text{ or } x > b \end{cases} \quad (7.15)$$

$$F(x; a, b) = \begin{cases} 0 & x < a \\ \frac{x-a}{b-a} & a \leq x \leq b \\ 1 & x > b \end{cases} \quad (7.16)$$

Normal Distribution

Below are the PDF and CDF of the normal distribution implemented by the Mathematics module. Parameters are μ (mean) and σ (standard deviation), erf is the error function.

$$f(x; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (7.17)$$

$$F(x; \mu, \sigma) = \frac{1}{2} + \frac{1}{2} \operatorname{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right) \quad (7.18)$$

Lognormal Distribution

Below are the PDF and CDF of the lognormal distribution implemented by the Mathematics module. Parameters are μ (mean) and σ (standard deviation), erf is the error function.

$$f(x; \mu, \sigma) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(\ln(x)-\mu)^2}{2\sigma^2}} \quad (7.19)$$

$$F(x; \mu, \sigma) = \frac{1}{2} + \frac{1}{2} \operatorname{erf}\left(\frac{\ln(x)-\mu}{\sigma\sqrt{2}}\right) \quad (7.20)$$

Exponential Distribution

Below are the PDF and CDF of the exponential distribution implemented by the Mathematics module. The parameter is λ (rate or inverse scale).

$$f(x; \lambda) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases} \quad (7.21)$$

$$F(x; \lambda) = \begin{cases} 1 - e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases} \quad (7.22)$$

Weibull Distribution

Below are the PDF and CDF of the Weibull distribution implemented by the Mathematics module. Parameters are λ (scale) and k (shape).

$$f(x; \lambda, k) = (k/\lambda) (x/\lambda)^{k-1} e^{-(x/\lambda)^k} \quad (7.23)$$

$$F(x; \lambda, k) = 1 - e^{-(x/\lambda)^k} \quad (7.24)$$

Triangular Distribution

Below are the PDF and CDF of the triangular distribution implemented by the Mathematics module. Parameters are a (minimum), b (maximum) and c (mode).

$$f(x; a, b, c) = \begin{cases} \frac{2(x-a)}{(b-a)(c-a)} & a \leq x \leq c \\ \frac{2(b-x)}{(b-a)(b-c)} & c \leq x \leq b \end{cases} \quad (7.25)$$

$$F(x; a, b, c) = \begin{cases} \frac{(x-a)^2}{(b-a)(c-a)} & a \leq x \leq c \\ 1 - \frac{(b-x)^2}{(b-a)(b-c)} & c \leq x \leq b \end{cases} \quad (7.26)$$

Multi-modal Distribution

Finally, the Mathematics module also contains a multi-modal distribution implementation. Parameters in this case are the number of modes (n_m), a list of probabilities for each mode, and a list of specifications of the distribution corresponding with each mode. Algorithm 1 shows how the multi-modal distribution is implemented.

7.8.3 Interpolation

Interpolation has many applications in Tornado and can either be applied to simulated trajectories or to time series supplied by the user. Given $(x_1, y_1 = f(x_1)), (x_2, y_2 = f(x_2)), \dots, (x_n, y_n = f(x_n))$ with $k = 1, \dots, n$, below are descriptions of the interpolation schemes that are provided by the Mathematics module of the Common Library.

Zero-hold

$$f(x) = \begin{cases} y_1 & x < x_1 \\ y_k & x_k \geq x < x_{k+1}, \forall k \\ y_n & x \geq x_n \end{cases} \quad (7.27)$$

Linear

$$f(x) = \begin{cases} y_1 & x < x_1 \\ y_k + (x - x_k) \frac{(y_{k+1} - y_k)}{(x_{k+1} - x_k)} & x_k \geq x < x_{k+1}, \forall k \\ y_n & x \geq x_n \end{cases} \quad (7.28)$$

Algorithm 1 Generation of a Random Value from a Multi-modal Distribution

```
v1 = RandUniform()
SumProbabilities := 0
for i := 1 to nm do
    SumProbabilities := SumProbabilities + Probability[i]
    if v1 < SumProbabilities then
        j := i
    end if
end for
if Typej = Uniform then
    v2 = RandUniform(Minj, Maxj)
else
    if Typej = Normal then
        v2 = RandNormal( $\mu_j$ ,  $\sigma_j$ )
    else
        if Typej = LogNormal then
            v2 = RandLogNormal( $\mu_j$ ,  $\sigma_j$ )
        else
            if Typej = Exp then
                v2 = RandExp( $\lambda_j$ )
            else
                if Typej = Weibull then
                    v2 = RandWeibull( $\lambda_j$ )
                else
                    if Typej = Triangular then
                        v2 = RandTriangular(Medianj, Minj, Maxj)
                    end if
                end if
            end if
        end if
    end if
end if
end if
```

Lagrange

In this case, a Lagrange polynomial is fitted through the m points of the time series, denoted as $(x'_1, y'_1), \dots, (x'_m, y'_m)$, which are closest to x :

$$P(x) = \sum_{j=1}^m P_j(x) \quad (7.29)$$

$$P_j(x) = y'_j \prod_{k=1, k \neq j}^m \frac{x - x'_k}{x'_j - x'_k} \quad (7.30)$$

The interpolated value is then taken from this polynomial:

$$f(x) = \begin{cases} y_1 & x < x_1 \\ P(x) & x_1 \leq x < x_n \\ y_n & x \geq x_n \end{cases} \quad (7.31)$$

Cubic Spline

If the value of the argument is smaller than the argument of the first list element, the value of the first element is returned. If the value of the argument is larger than the value of the argument of the last list element, the value of the last list element is returned. If the value of the argument is within the range of the arguments of the list, the interpolated value is given by cubic polynomials between neighboring data points, which are determined by the conditions of continuous first and second derivatives at inner data points and by zero first derivatives at the end points.

Algorithm 2 and Algorithm 3 (Cheney and Kincaid, 2003) respectively show how cubic spline coefficients can be computed and used later on during the interpolation process. In these algorithms, i is an index, n is the number of known data points, x and y are the abscissa and ordinates of the known data points, z are the cubic spline coefficients, \hat{x} is the argument for the interpolation and \hat{y} is the result of the interpolation. The other variables (h, b, u, v and tmp) that are used in the algorithms are temporary worker variables.

Table 7.2: Cubic Spline Symbol Descriptions

Symbol	Description
i	Index
n	Number of points
x	Abscissa
y	Ordinate
h, b, u, v, tmp	Temporary variables
\hat{x}	Argument of interpolation
\hat{y}	Result of interpolation

7.9 Platform

The Platform module contains a number of functions that are abstractions of platform-specific routines, such as:

Algorithm 2 Computation of Cubic Spline Coefficients

```

for  $i := 0$  to  $n - 1$  do
     $h_i := x_{i+1} - x_i$ 
     $b_i := (y_{i+1} - y_i)/h_i$ 
end for
 $u_1 := 2.0 \times (h_0 + h_1)$ 
 $v_1 := 6.0 \times (b_1 - b_0)$ 
for  $i := 2$  to  $n - 1$  do
     $u_i := 2.0 \times (h_i + h_{i+1}) - (h_{i-1} \times h_{i-1})/u_{i-1}$ 
     $v_i := 6.0 \times (b_i - b_{i-1}) - (h_{i-1} \times v_{i-1})/u_{i-1}$ 
end for
 $z_n := 0$ 
for  $i := n - 1$  to  $1$  do
     $z_i := (v_i - h_i \times z_{i+1})/u_i$ 
end for
 $z_0 := 0$ 

```

Algorithm 3 Interpolation using Cubic Spline Coefficients

```

for  $j := n - 2$  to  $1$  do
    if  $\hat{x} - x_j \geq 0$  then
        break
    end if
end for
 $h := x_{j+1} - x_j$ 
 $tmp = (z_j \times 0.5) + (\hat{x} - x_j) \times (z_{j+1} - z_j)/(6.0 \times h)$ 
 $tmp = -(h/6.0) \times (z_{j+1} + 2.0 \times z_j) + (y_{j+1} - y_j)/h + (\hat{x} - x_j) \times tmp$ 
 $\hat{y} = y_j + (\hat{x} - x_j) \times tmp$ 

```

- Retrieving and modifying operating system environment variables
- Retrieving the current user name
- Retrieving the host name
- Executing an operating system command
- Halting execution of the current process for a specified number of milliseconds

In Tornado, functions from the Platform module are mainly used in file-related operations.

7.10 Vectors and Matrices

During the implementation of the first incarnation of the Tornado kernel, the use of STL for the representation of vectors and matrices was not an option, since at that time, STL was not yet available for the majority of C++ compilers (and even if it was, it was not yet sufficiently stable). During the design process of the second incarnation of the kernel, STL had however become mainstream. It has therefore been investigated whether this library, and more specifically the `std::vector` data type, would be a good option for the implementation of vectors and matrices.

As has been shown earlier in Table 6.3, STL can still be considered slow for heavy-duty numerical computations, and does not come near to the speed of plain C-style arrays. This of course causes a dilemma: on the one hand STL allows for improving code base quality, but on the other it does not provide for the speed that is needed for high-end computational applications. In order to solve this

dilemma, it was decided to implement a number of data types in the Common Library that offer some of the advantages of STL vectors, but do not suffer from the same performance issues. The *CVectorChar*, *CVectorInt*, *CVectorLong* and *CVectorDouble* data types act as smart pointers (they perform clean-up of the memory pointed to by the pointer when instances of the class go out of scope) and provide an interface that is a subset of the *std::vector* interface. More specifically, the *CVector** data types have implementations for the *empty()*, *size()*, *resize()*, *clear()* and *at()* methods and also implement the *[]*, *=*, *==* and *!=* operators. Other methods such as *push_back* and *pop_back* are not available.

The *Common::CVector** classes all have a similar implementation but use a different data type. In order to avoid duplication of code, and to avoid the use of templates (which is one of the reasons why the *std::vector* data type is slow), an old-fashioned, but efficient macro mechanism was used.

As can be seen from Table 6.3, *CVectorDouble* is considerably faster than *std::vector<double>* (e.g., comparing Items 1 and 5 in the table), but still not as fast as *double[]* (Item 8 in the table). One of the reasons is the use of *[]* as an overloaded operator. In order to circumvent this, direct access to the pointer that is guarded by the *CVectorDouble* class is provided through the *_* (underscore) member function. Using this direct access method, performance can further be improved (Item 6 in the table). Finally, one can question whether smart pointer classes can be used everywhere in the code. In the case of Tornado, the answer is no since executable models are represented in C (see Chapter 8). Executable models will therefore always use *double[]* as a data type. The timing results of a test that conforms to this situation (Item 7 in the table) show that performance is almost equal to implementations that only use *double[]* (Item 8 in the table), when *double[]* is used for the model and *CVectorDouble* plus the *_* member is used for the encapsulating code.

The advantage of using the *CVector** data types, even if direct access to the guarded pointer is allowed and even if *double[]* is used for executable models, remains the fact that automatic memory clean-up is provided. Also, another advantage is the possibility to alter the implementation of the *[]* operator. One can for instance let it perform index checking only in case the code is compiled in debug mode, which will lead to slow but safe algorithmic code. In general, the situation would then be as in Table 7.3. Running the code in debug mode will allow for problems to be detected. Once the code is considered error-free, it can be recompiled to a much faster but less safe release version.

Table 7.3: Index Checking Strategies for the *Common::Vector** Data Type

Indexing Operator	Perform Index Check
<i>._[]</i>	Never
<i>.at()</i>	Always
<i>.[]</i>	Only in debug mode

Next to the *CVector** data types, the Common Library also implements *CMatrix** data types. These are used to represent two-dimensional matrices and are entirely analogous to the vector data types.

8

Experimentation Environment

8.1 Introduction

Although during normal operation, models evidently first need to be built before they can be used in a virtual experiment, the sequence will be reversed while discussing the implementation of the Tornado kernel. For, experience has shown that starting by explaining the details of the Experimentation Environment rather than by first discussing the Modelling Environment facilitates understanding. Also, during the development process itself, the Experimentation Environment was tackled first. This applies to Tornado-I as well as to Tornado-II.

The Experimentation Environment consists of the following elements, which are discussed in the remainder of this chapter:

- An extensible framework for hierarchical structuring of highly-configurable virtual experiments
- A generalized framework for abstraction and dynamic loading of numerical solvers
- A framework for managing graphical input provider and output acceptor representations
- Provisions for internationalization and localization

Finally, also the improvement of simulation speed through *a priori* exploration of the solver setting space is discussed, as a special feature of the Experimentation Environment.

Parts of this chapter have previously been published in (Claeys et al., 2006b), (Claeys et al., 2006e) and (Claeys et al., 2006f).

8.2 Hierarchical Virtual Experimentation

8.2.1 Basic Concepts

Tornado currently implements 15 different types of virtual experiments. Each of these experiment types are referred to by a full name and/or a shorthand, as listed below:

- **ExpSimul**: Dynamic Simulation Experiment

- **ExpSSRoot**: Steady-state Analysis Experiment with Root Finder
- **ExpSSOptim**: Steady-state Analysis Experiment with Optimizer
- **ExpObjEval**: Objective Evaluation Experiment
- **ExpScen**: Scenario Analysis Experiment
- **ExpMC**: Monte Carlo Analysis Experiment
- **ExpOptim**: Optimization Experiment
- **ExpCI**: Confidence Information Analysis Experiment
- **ExpSens**: Sensitivity Analysis Experiment
- **ExpStats**: Statistical Analysis Experiment
- **ExpEnsemble**: Ensemble Simulation Experiment
- **ExpSeq**: Sequential Execution Experiment
- **ExpMCOptim**: Optimization Monte Carlo Analysis Experiment
- **ExpScenOptim**: Optimization Scenario Analysis Experiment
- **ExpScenSSRoot**: Steady-state Analysis with Root Finder Scenario Analysis Experiment

Experiments are entities that support properties. All experiment types are implemented as classes that are derived from a common base class, which is named *Exp*. Figure 8.1 depicts the inheritance hierarchy of experiment types.

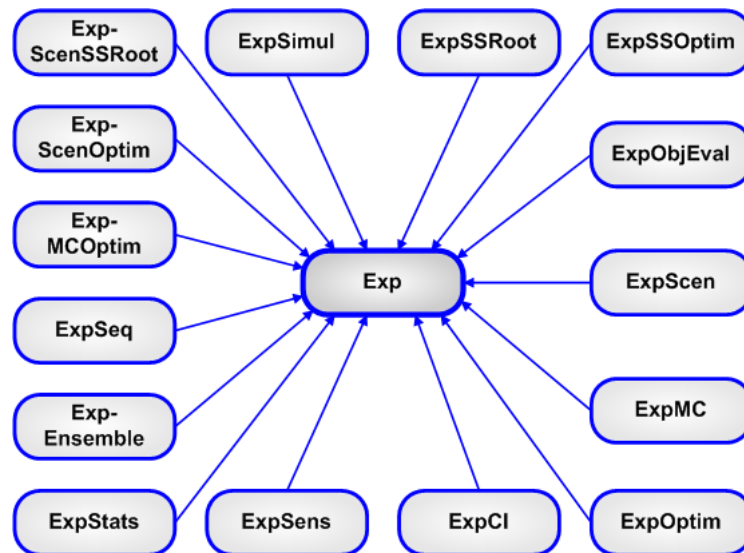


Figure 8.1: Experiment Inheritance Hierarchy

Next to an inheritance relationship, there is also a composition relationship between experiment types. Some experiments are **atomic** and can therefore not be further decomposed. Others however are **compound**, in the sense that they are internally based on one or more other experiment types. Figure 8.2 depicts the composition hierarchy of experiment types. In this figure, full lines point to the fact that

one experiment contains another either through **embedding** or through **referral**. Dashed lines point to a relationship that is based on referral only. An experiment is embedded in another if its description is fully part of the encapsulating experiment. On the other hand, an experiment is referred to by another if it has a separate description that is merely pointed to by the other experiment, instead of being fully contained within. One may notice that the ExpSeq experiment type is not mentioned on the figure. Reason for this is that ExpSeq can contain one or more experiments of any type. Expressing this in the figure would be very unfavorable for readability.

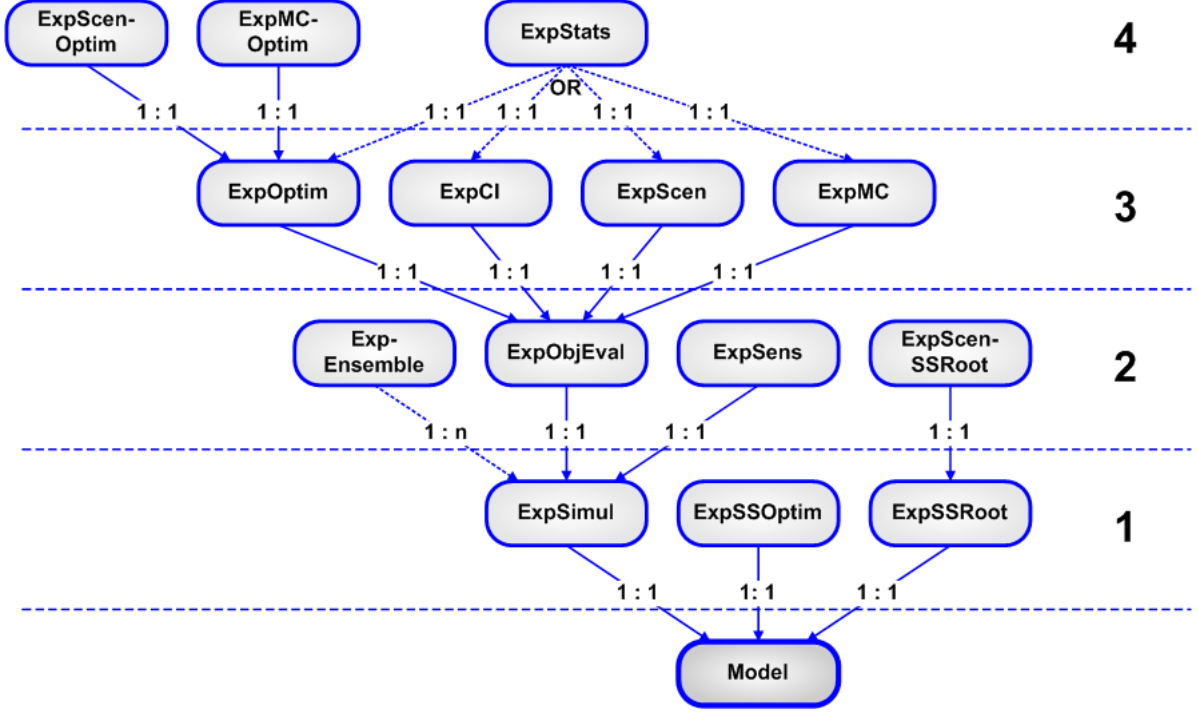


Figure 8.2: Experiment Composition Hierarchy

An alternative manner of presenting composition relationships that has been adopted in the scope of Tornado is $Exp_1[Exp_2[\dots Exp_n[Model]\dots]]$, i.e., Exp_1 contains Exp_2 which on turns contains one or more other experiments. Using this representation system, the composition relationships of Figure 8.2 can be represented as follows:

- 1-level
 - $ExpSimul[Model]$
 - $ExpSSRoot[Model]$
 - $ExpSSOptim[Model]$
- 2-level
 - $ExpObjEval[ExpSimul] \rightarrow ExpObjEval[ExpSimul[Model]]$
 - $ExpSens[ExpSimul] \rightarrow ExpSens[ExpSimul[Model]]$
 - $ExpEnsemble[ExpSimul+] \rightarrow ExpEnsemble[ExpSimul[Model]+]$
 - $ExpScenSSRoot[ExpSSRoot] \rightarrow ExpScenSSRoot[ExpSSRoot[Model]]$
- 3-level
 - $ExpScen[ExpObjEval] \rightarrow ExpScen[ExpObjEval[ExpSimul[Model]]]$

- $ExpCI[ExpObjEval] \rightarrow ExpCI[ExpObjEval[ExpSimul[Model]]]$
- $ExpOptim[ExpObjEval] \rightarrow ExpOptim[ExpObjEval[ExpSimul[Model]]]$
- $ExpMC[ExpObjEval] \rightarrow ExpMC[ExpObjEval[ExpSimul[Model]]]$
- 4-level
 - $ExpScenOptim[ExpOptim] \rightarrow ExpScenOptim[ExpOptim[ExpObjEval[ExpSimul[Model]]]]$
 - $ExpMCOptim[ExpOptim] \rightarrow ExpMCOptim[ExpOptim[ExpObjEval[ExpSimul[Model]]]]$
 - $ExpStats[ExpScen|ExpCI|ExpOptim|ExpMC]$
- n-level
 - $ExpSeq[Exp+]$

In the above, $[]$ can be regarded as a containment operator. The $+$ operator refers to one or more occurrences of the item to which it is applied, while $|$ refers to items that are alternatives. Finally, \rightarrow can be regarded as an expansion operator.

Figure 8.2 depicts how experiment descriptions are composed from sub-experiments, however it does not give any information on the number of evaluations of these sub-experiments. Figure 8.3 contains this information. A value of 1 means that a sub-experiment is executed exactly once, a value of n means it is executed a number of times for different initial values. The meaning of “initial values” is different for each type of sub-experiment.

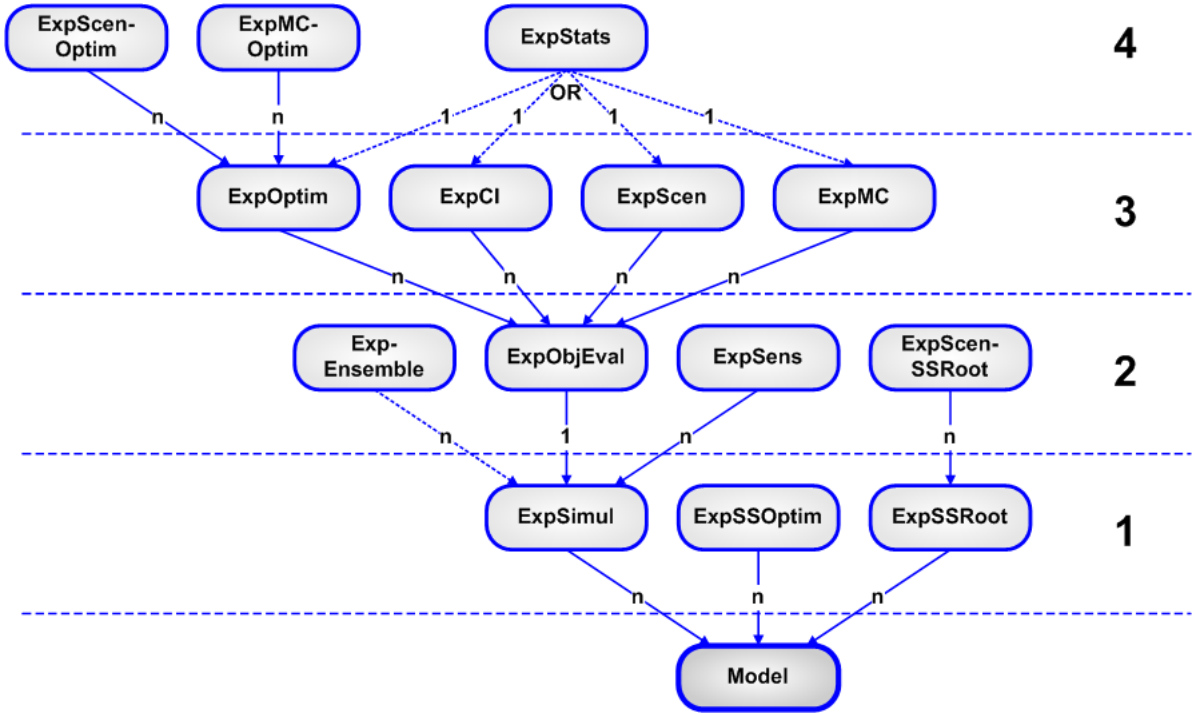


Figure 8.3: Experiment Composition Hierarchy and Number of Runs

Although all experiment types have a different nature, they all use a subset from the same collection of concepts. These concepts are described below:

- **Callbacks** are references to abstract interfaces implemented by the application encapsulating the Tornado kernel. Methods from these interfaces are called by the Tornado kernel with the appropriate arguments when certain conditions arise. All experiment types use one or more of the following callbacks:

- **CallbackMessage:** Requests the encapsulating application to display a text message. Messages can be informational, warnings or errors.
- **CallbackTime:** Requests the encapsulating application to display the current simulation time.
- **CallbackStop:** Requests a boolean flag from the encapsulating application indicating whether processing is to be stopped.
- **CallbackRunNo:** Requests the encapsulating application to display the current sub-experiment run number, in case of iterative execution of the same sub-experiment.
- **CallbackControls:** Requests the encapsulating application to capture data using a particular graphical input provider.
- **CallbackPlot:** Requests the encapsulating application to render data using a particular graphical output acceptor.

The way the callbacks are implemented is entirely to be decided by the application, *e.g.*, a CUI application may present error messages in the console window, while a GUI application may present them using a dialog box.

- **Solvers** are numerical algorithms implemented as dynamically-loadable modules. Solvers are used in those cases where various alternative numerical algorithms exist to solve a particular problem. Alternative solver implementations usually have a different set of properties. The interpretation of the term “solver” is very broad in the scope of Tornado and is by no means limited to integration or optimization solvers. In fact, every experiment type is free to introduce its own experiment-specific interpretation of the solver concept.
- **Inputs** are conglomerates of input providers that supply input data to experiments.
- **Outputs** are conglomerates of output acceptors that process output data from experiments.
- **Logs** are used for unformatted textual information generated during the execution of an experiment.
- **Managers** are collections of other types of items.
- **Vars** or Variables are entities of a compound experiment that can be set to a different value in between sub-experiment executions. Typically, experiment variables are linked to parameters, input variables or initial conditions of the model. For the remainder of this document, it is very important to clearly distinguish experiment variables from model variables. Model variables can be set to different values at every time point; experiment variables can only be set to different values in between sub-experiment runs.
- **Quantities** are items that are used for the computation of various objectives during the course of an experiment. In the scope of Tornado, the word *quantity* is adopted as a collective term that is used to denote parameters, independent variables, input variables, output variables, state (derived) variables and worker (algebraic) variables. So, when P, T, U, Y, X and W respectively represent the sets of parameters, independent variables, input variables, output variables, state variables and worker variables of a model, then $Q = P \cup T \cup U \cup Y \cup X \cup W$, where Q is the set of quantities. Clearly, the interpretation of the term *quantity* in the scope of Tornado is somewhat different from its classical interpretation where quantity for instance refers to mass, volume, temperature, time, force, *etc.*
- **Times** are specifications of time bases. They include at least information on start and stop times, but may also include information on how to proceed from start to stop, *e.g.*, through linearly or logarithmically spaced intervals.

- **Objs** or Objectives are procedures that compute one or more objective values. Objectives are normally the heart of a virtual experiment and work in unison with solvers and time bases. In some cases however, experiments do not have a separate objective entity as most of the processing is done by the solvers.
- **ObjValues** or Objective Values are data items that are computed by objectives.
- **Models** are executable models that have been generated through the Tornado Modelling Environment

8.2.2 Input Providers

Several experiment types rely on input data during their operation. Various types of input providers have been implemented, although not every experiment type that requires input data supports all of these. The input providers that exist within Tornado are the following:

- **InputFiles** are data files containing one or more time series. Input files are self-descriptive, *i.e.*, they provide information on the names and units of the time series they contain. Information can be retrieved through the interpolation schemes that are implemented by the Common Library.
- **InputBuffers** are internal data buffers containing one or more time series. Input buffers are self-descriptive, *i.e.*, they provide information on the names and units of the time series they contain. Also, information can be retrieved through the interpolation schemes that are implemented by the Common Library.
- **InputCalcVars** generate data values on the basis of a custom function and the current simulation time (*CalcVar* stands for calculator variable). The custom function that the user provides is compiled to a binary plug-in, which is loaded by the Tornado kernel at run-time.
- **InputGenerators** generate data values on the basis of one of several hard-coded functions and the current simulation time. Unlike input calculator variables, input generators are contained within the Tornado kernel itself and hence incur less overhead.
- **InputControls** are channels that can be used to retrieve data from the graphical input providers that are described through the Controls module of the Experimentation Environment.

Below follows an overview of the input generators that have been implemented in Tornado. A description of the symbols that are used can be found in Table 8.1.

Clock

$$y = \begin{cases} y_0 & t < t_0 \\ y_0 + (t - t_0) & t \geq t_0 \end{cases} \quad (8.1)$$

Constant

$$y = y_0 \quad (8.2)$$

Sine

$$y = \begin{cases} y_0 & t < t_0 \\ y_0 + A \times \sin(2\pi f(t - t_0) + \phi) & t \geq t_0 \end{cases} \quad (8.3)$$

Table 8.1: Input Generator Symbol Descriptions

Symbol	Description
A	Amplitude
$Rand$	Random number
T	Period or sampling interval
μ	Mean
ϕ	Phase
σ	Standard deviation
c	Damping coefficient
f	Frequency
h	Height
t	Current time
t_0	Start time
t_d	Duration
w	Width
y	Output of generator
y_0	Offset for output
y_{prev}	Previously generated output
y_{max}	Maximum output
y_{min}	Minimum output

DoubleSine

$$y = \begin{cases} y_0 & t < t_0 \\ y_0 + A_1 \times \sin(2\pi f_1(t - t_0) + \phi_1) + A_2 \times \sin(2\pi f_2(t - t_0) + \phi_2) & t \geq t_0 \end{cases} \quad (8.4)$$

ExpSine

$$y = \begin{cases} y_0 & t < t_0 \\ y_0 + A \times e^{-c(t-t_0)} \times \sin(2\pi f(t - t_0) + \phi) & t \geq t_0 \end{cases} \quad (8.5)$$

Step

$$y = \begin{cases} y_0 & t < t_0 \\ y_0 + h & t \geq t_0 \end{cases} \quad (8.6)$$

Pulse

$$y = \begin{cases} y_0 & t < t_0 \\ y_0 + A & t \geq t_0, \text{ mod}(t - t_0, T) < (w * T/100) \\ y_0 & t \geq t_0, \text{ mod}(t - t_0, T) \geq (w * T/100) \end{cases} \quad (8.7)$$

Ramp

$$y = \begin{cases} y_0 & t < t_0 \\ y_0 + (t - t_0) \times h/t_d & t < t_d \\ y_0 + h & t \geq t_d \end{cases} \quad (8.8)$$

SawTooth

$$y = \begin{cases} y_0 & t < t_0 \\ y_0 + A \times \text{mod}(t - t_0, T)/T & t \geq t_0 \end{cases} \quad (8.9)$$

Rand

$$y = \begin{cases} y_0 & t < t_0 \\ y_0 + \text{Rand} & t \geq t_0, t > t_{prev} + T \\ y_{prev} & t \geq t_0, t \leq t_{prev} + T \end{cases} \quad (8.10)$$

$$\text{Rand} = \begin{cases} y_0 + \text{RandUniform}(y_{min}, y_{max}) \\ y_0 + \text{RandTruncNormal}(\mu, \sigma, y_{min}, y_{max}) \\ y_0 + \text{RandTruncLogNormal}(\mu, \sigma, y_{min}, y_{max}) \\ y_0 + \text{RandTruncExp}(\lambda, y_{min}, y_{max}) \\ y_0 + \text{RandTruncWeibull}(\lambda, k, y_{min}, y_{max}) \\ y_0 + \text{RandTriangular}(y_{median}, y_{min}, y_{max}) \\ y_0 + \text{RandMultiModal}(\text{Specs}, P) \end{cases} \quad (8.11)$$

As was mentioned above, an interpolation method can be defined for input files and input buffers. Possible options are zero-hold, linear, Lagrange and cubic spline interpolation, all of which are implemented through the Common Library. For input files and input buffers, it is also possible to enable repetition, which allows for indefinite repetition of a specified fragment (window) of the input file or buffer data.

8.2.3 Output Acceptors

Most experiment types send a selection of the data that is generated during their execution to output acceptors. Several types of output acceptors have been implemented, although not every experiment type supports all of these. It is possible to send all data that is generated to an output acceptor, or to use a specific communication interval. Communication points can be linearly or logarithmically spaced. The output acceptors that exist within Tornado are the following:

- **OutputFiles** are data files in which one or more time series can be stored. In addition to the time series data, also name and unit information is stored in files, in order to make them self-descriptive.
- **OutputBuffers** are internal data buffers in which one or more time series can be stored. In addition to the time series data, also name and unit information is stored in buffers, in order to make them self-descriptive.
- **OutputCalcVars** compute values from simulated data according to a custom function (*CalcVar* stands for calculator variable). The custom function that the user provides is compiled to a binary plug-in, which is loaded by the Tornado kernel at run-time.
- **OutputPlots** are channels that can be used to send data to the graphical output acceptors that are described through the Plot module of the Experimentation Environment.

8.2.4 General Form of a Virtual Experiment

Bringing together the virtual experiment concepts that were discussed so far, Figure 8.4 depicts the general form of a virtual experiment. A core virtual experiment may consist of an objective (*Obj*) that interacts with one or more solvers (*Solvers*) and time bases (*Times*). The objective generates new variable

values (*Vars*) to be applied to one or more sub-experiments (*Exps*). After or during execution, quantity values (*Quantities*) are retrieved from these sub-experiments and objective values (*ObjectiveValues*) are computed. During the execution of the virtual experiment, progress and status information is communicated to the encapsulating application by activating callbacks for messages (*CallbackMessage*), current time points (*CallbackTime*) and current run numbers (*CallbackRunNo*). Also, during the execution of the virtual experiment, input data is being retrieved from input providers, and output data is being sent to output acceptors. Some input providers operate entirely within the Tornado kernel (*InputBuffers*, *Input-Generators*), while others are partly situated at the application level (*InputControls*) or operating system level (*InputFiles*). For output acceptors, the situation is analogous: some output acceptors operate entirely within the Tornado kernel (*OutputBuffers*), while others are partly situated at the application level (*OutputPlots*) or operating system level (*OutputFiles*). Finally, logging information (*Logs*) may be generated during the course of an experiment and user interrupts (*CallbackStop*) may be provided by the encapsulating application.

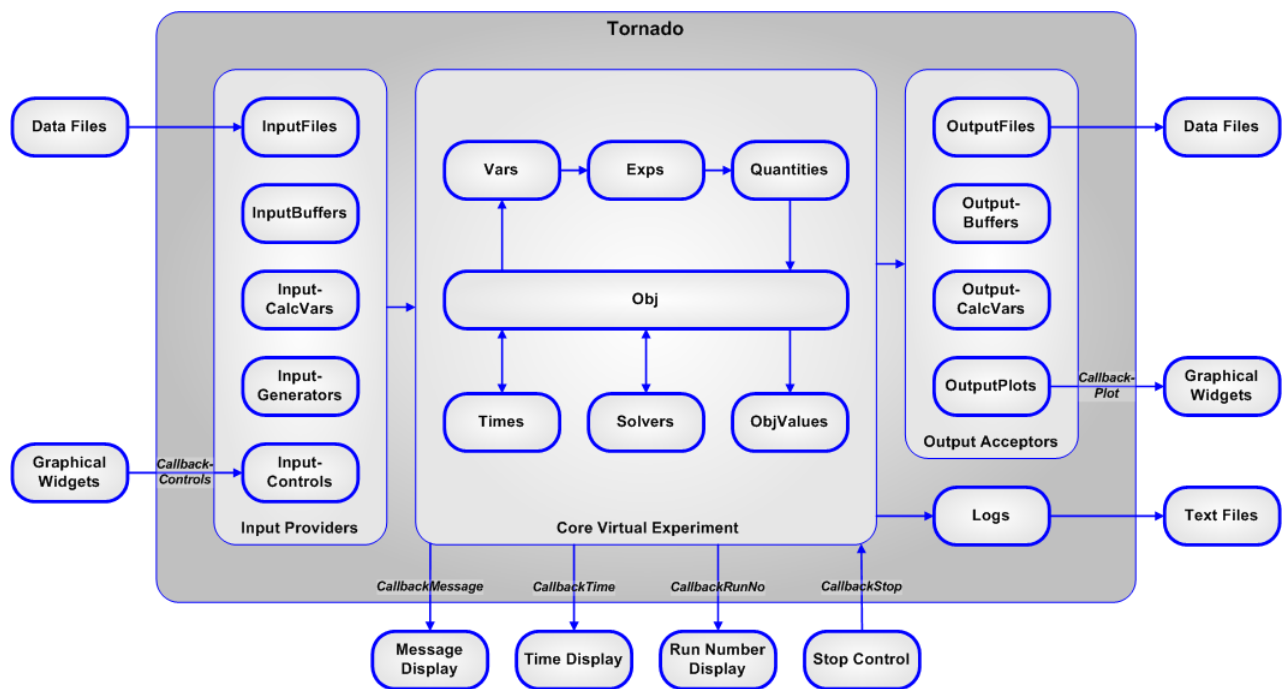


Figure 8.4: General Form of a Virtual Experiment

8.3 Virtual Experiment Types

As mentioned before, the Tornado kernel currently implements 15 experiment types. Below follows a description of each of these. For each experiment, the description is completed by an informal entity-relationship (ER) diagram that shows the internal representation of the experiment, by depicting the associations between the experiment's entities and other items. The following can be used as a legend when interpreting these ER diagrams:

- Boxes with solid lines represent classes
- Boxes with dashed lines represent callbacks
- Items that are underlined support the properties mechanism (and are therefore entities). Items that are not underlined do not support the properties mechanism.

- For each association, the cardinality is given (1 : 1 refers to a one-to-one relationship, 1 : n to a one-to-many relationship).

Next to ER diagrams, also graphical renderings of the XML Schema Definition of each experiment's persistent representation are given. These persistent representations are embedded in the generic experiment representation, as is illustrated by Figure 8.5. The following can be used as a legend when interpreting graphical representations of XML Schema Definitions:

- Square boxes with solid lines represent mandatory tags
- Square boxes with dashed lines represent optional tags. The number of occurrences is mentioned below the box (usually this is $0 \dots \infty$).
- Round boxes with three horizontal dots represent a sequence
- Round boxes with a switch and three vertical dots represent a set of alternatives

So, in Figure 8.5 the Exp tag (representing an experiment) contains a sequence consisting of the Props tag and one of several alternatives (to be discussed later). The Props tag contains a sequence of zero or more occurrences of the Prop tag.

8.3.1 ExpSimul: Dynamic Simulation

The ExpSimul experiment type implements a dynamic simulation. It either computes the time-dependent state trajectories of an ODE / AE model, or of a model containing only algebraic equations. This type of experiment is atomic, it works directly on an executable model (*Model*) and does not contain any sub-experiments. A time base (*SimulTime*) provides information on start and stop time, while simulation progress and status information are communicated to the encapsulating application through callbacks (*CallbackTime* and *CallbackMessage*). Also, a simulation user stop can be incurred through a callback (*CallbackStop*). This experiment type does not contain an objective entity, and is entirely governed by an integration solver (*SolveInteg*). For solving implicit loops, a root-finding solver (*SolveRoot*) is also available. Integration and root-finding solvers are loaded at run-time from a collection of binary solver plug-ins that can be provided to Tornado. An overview of the integration and root-finding solvers that have been implemented as a solver plug-in can be found in Appendix A and Appendix B, respectively.

Of all experiment types, ExpSimul experiments allow for the widest variety of input providers and output acceptors to be used. For input one can use files, buffers, calculator variables and generators (in a later stage, controls will also be added to this list). For output one can use files, buffers, calculator variables and plots.

An ER diagram of ExpSimul can be found in Figure 8.6. In close correspondence with the ER diagram is the XML Schema Definition that is presented in Figure 8.7.

8.3.2 ExpSSRoot and ExpSSOptim: Steady-state Analysis

The ExpSSRoot and ExpSSOptim experiment types both allow for computing the steady-state of a system. A system is in steady-state when transient effects have worn out, *i.e.*, when the derivatives of all state variables are zero. Although both experiment types have the same goal, they operate in a somewhat different manner: ExpSSRoot relies on a non-linear solver (root solver), whereas ExpSSOptim uses a regular optimization algorithm.

As discussed in Chapter 2, (8.12) represents the state equations of a system. As in steady-state, the derivatives of state variables must be zero, this vector equation reduces to (8.13), which is no longer a

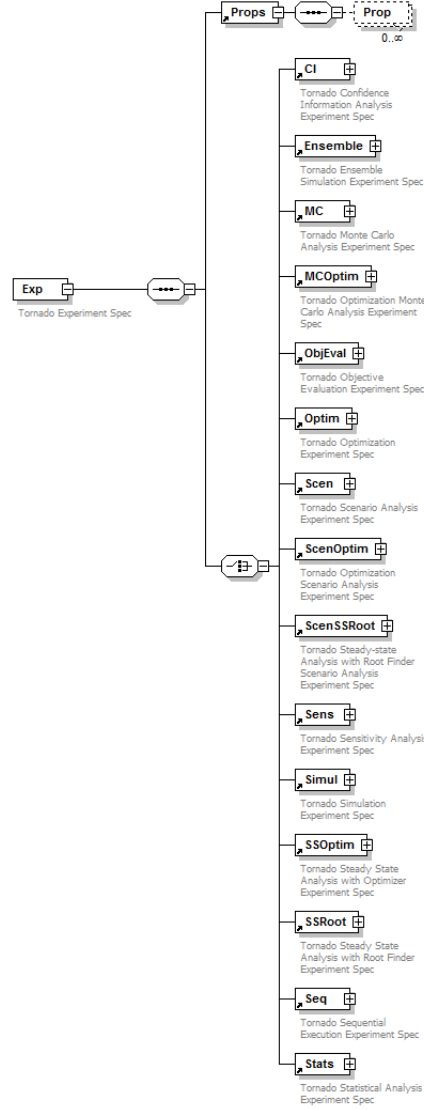


Figure 8.5: Exp XML Schema Definition

differential equation, but a regular non-linear algebraic equation. In order to find the steady-state of X , (8.13) needs to be solved for X , keeping U constant.

$$\frac{dX}{dt} = f(X(t), \Theta, U(t), W(t), t) \quad (8.12)$$

$$0 = f(X, \Theta, U, W) \quad (8.13)$$

Since at the executable level, equations need to be transformed to assignments, (8.13) needs to be presented as (8.14) to the root finding algorithm. Here, R represents a vector of residues that need to be forced to 0 by the root solver.

$$R = f(X, \Theta, U, W) \quad (8.14)$$

An algebraic system such as (8.13) can potentially have multiple roots, a unique solution to the steady-state problem is therefore not guaranteed. Actually, the initial guess that is required by the root-

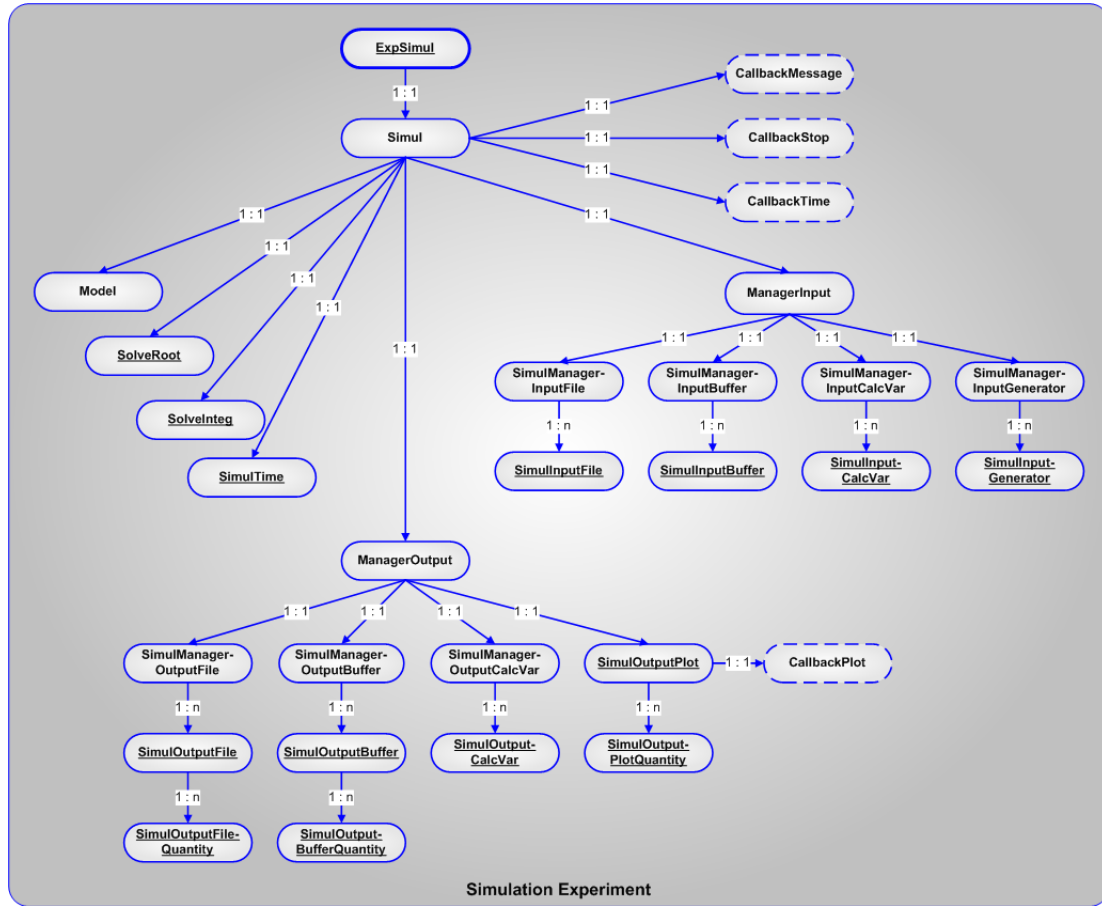


Figure 8.6: ExpSimul ER Diagram

finding algorithm will determine the root that is returned as a solution. A user may therefore have to try out different initial guesses in order to find a solution that is sound from a biological point of view.

The ExpSSRoot experiment type is an atomic experiment type that works directly on an executable model (*Model*). It does not contain any sub-experiments and does not generate time-based trajectories. It only uses a callback for messages (*CallbackMessage*) and a callback to poll for user interrupts (*CallbackStop*). It does not contain a separate objective entity since the root solver (*SolveRoot*) governs the entire process.

An ER diagram of the internal representation of the ExpSSRoot experiment type is in Figure 8.8. An XML Schema Definition of the persistent representation of this experiment type can be found in Figure 8.9.

As mentioned before, the goal of the ExpSSOptim experiment type is also to find the steady-state of a system, however in contrast to ExpSSRoot it will use a regular optimization solver for this. In order to make this possible, the model needs to be evaluated for different state variable values, and one single objective value needs to be associated with each model evaluation. Since the goal is to bring the derivatives of all state variables to zero, either (8.15) or (8.16) can be used for this. In these equations, J is the objective value that is to be minimized, n is the number of state variables (and hence the number of derivatives) and R_i are the elements of the vector of residues.

$$J = \sum_{i=0}^n R_i^2 \quad (8.15)$$

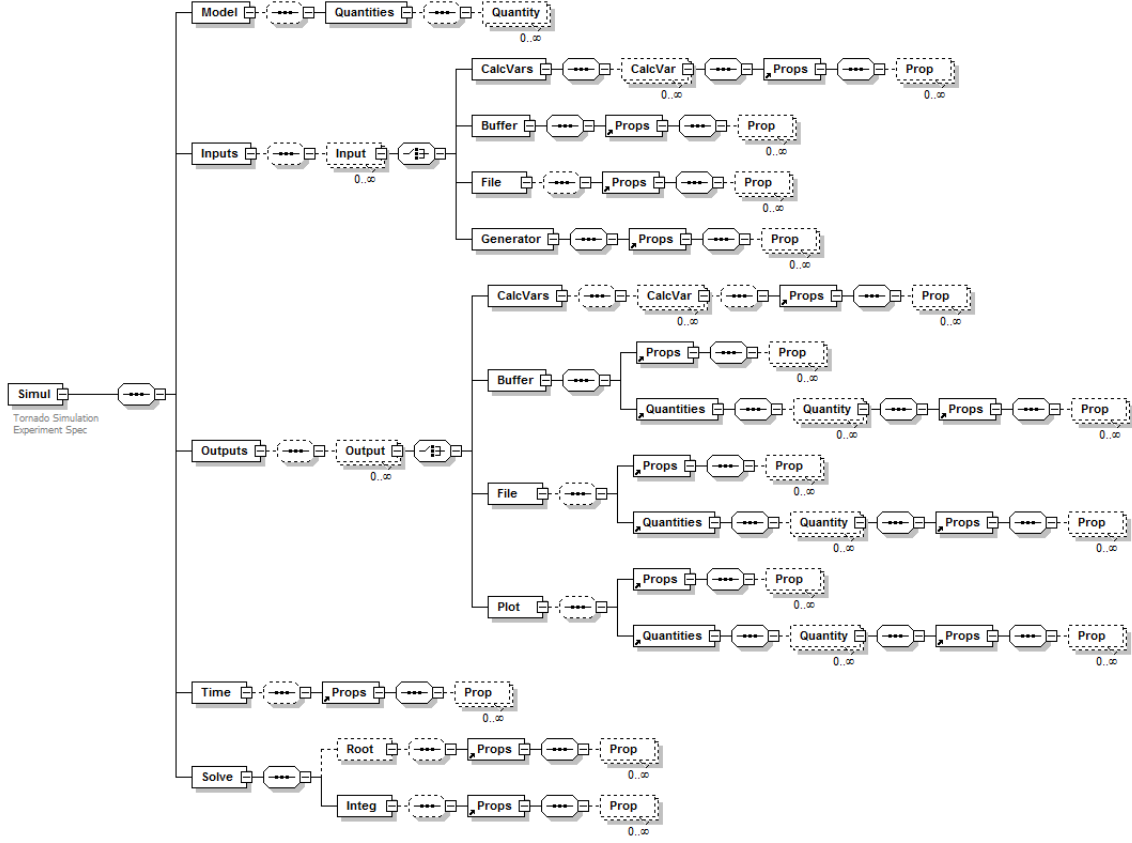


Figure 8.7: ExpSimul XML Schema Definition

$$J = \sum_{i=0}^n |R_i| \quad (8.16)$$

The ExpSSOptim experiment type is an atomic experiment that works directly on a model (*Model*) and does not contain any sub-experiments. It uses a callback for messages (*CallbackMessage*) and a callback to poll for user interrupts (*CallbackStop*). It does not contain a separate objective entity since the optimization solver (*SolveOptim*) governs the entire process. ExpSSOptim has a collection of variables (*OptimVars*) that mimic the set of state variables and is manipulated by the optimization solver. Finally, the experiment type also has a log (*OptimLogFile*).

An ER diagram of the internal representation of the ExpSSOptim experiment type is in Figure 8.10. An XML Schema Definition of the persistent representation of this experiment type can be found in Figure 8.11.

One may wonder why two different strategies are available for locating the steady-state of a system. For many problems, ExpSSRoot will work best since the solver that is used was specifically designed for solving sets of non-linear equations. However, in some cases ExpSSOptim may also prove to be interesting since it allows for a wide variety of optimization solvers to be used: gradient-based algorithms, genetic algorithms and algorithms based on simulated annealing. Also, some optimization solvers allow for constrained optimization. Having the ability to restrict the search space may be beneficial to locate roots that may otherwise be difficult to find.

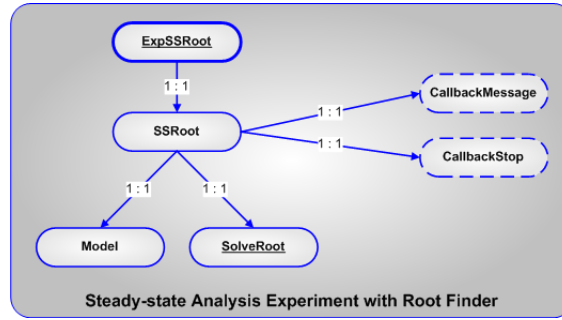


Figure 8.8: ExpSSRoot ER Diagram

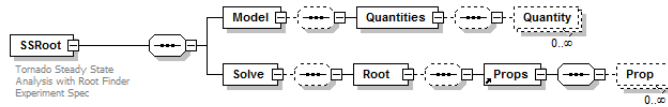


Figure 8.9: ExpSSRoot XML Schema Definition

8.3.3 ExpObjEval: Objective Evaluation

The ExpObjEval experiment type is capable of computing a large variety of objective values on the basis of the results of a dynamic simulation experiment. To do so, it first runs the simulation experiment and stores the trajectories of a number of selected quantities into an internal buffer. Subsequently, the buffer is used to compute the required objective values. Finally, all individual objective values are weighted and summed in order to come to one overall objective value (weighted multi-criteria objective). The objective types that can be used include simple aggregation functions (such as minimum, maximum and mean), as well as more complicated comparisons of simulated trajectories against reference time series.

The ExpObjEval experiment is a 2-level compound experiment since it contains an ExpSimul experiment (*ExpSimul*), either through embedding or through referral. It uses a callback for messages (*CallbackMessage*) and a callback to poll for user interrupts (*CallbackStop*). It has a collection of input files (*ObjEvalInputFile*) that can be used to describe the reference time series that are to be used in comparisons with simulated trajectories (other types of input providers, such as buffers, are not available at the moment but can easily be added at a later stage if desired). Also, it has an objective entity that

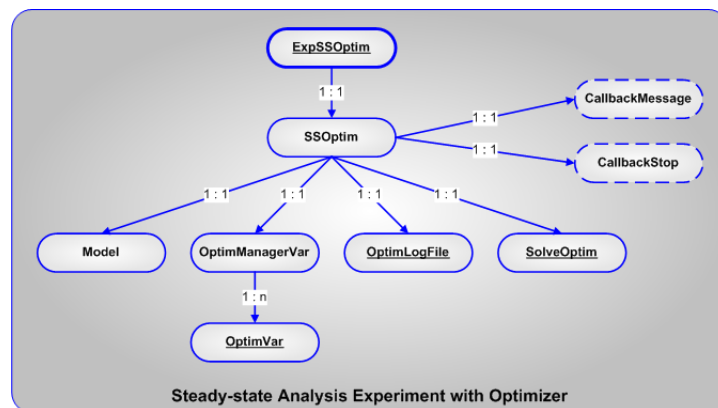


Figure 8.10: ExpSSOptim ER Diagram

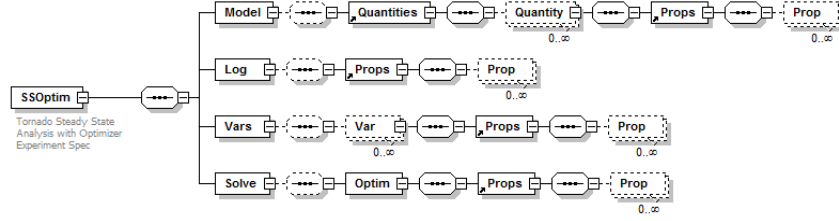


Figure 8.11: ExpSSOptim XML Schema Definition

executes the actual objective value computations (*ObjEvalObj*). Some objective values are related to a specific quantity (*ObjEvalObjQuantity*), while others apply to the overall objective.

An ER diagram of the internal representation of the ExpObjEval experiment type is in Figure 8.12. An XML Schema Definition of the persistent representation of this experiment type can be found in Figure 8.13.

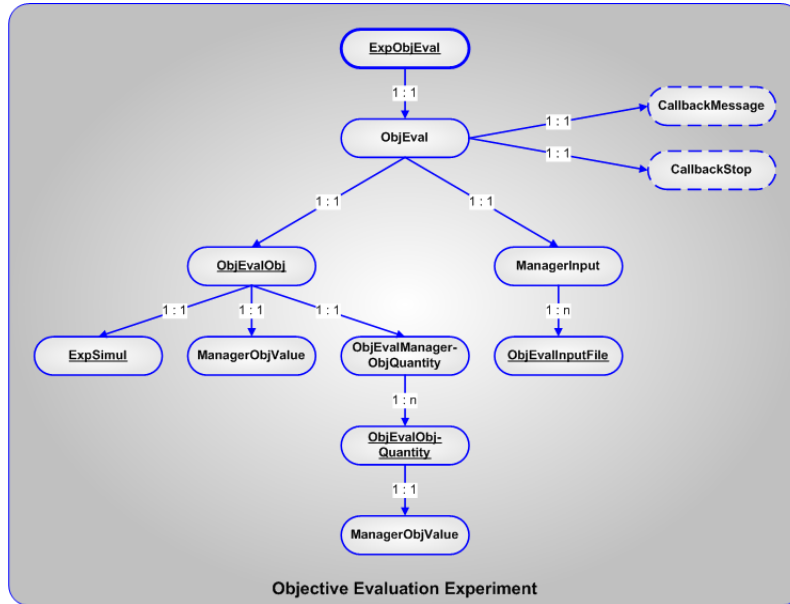


Figure 8.12: ExpObjEval ER Diagram

In the sequel, a full description is given of the objective computation process, which is in fact a 4-level procedure. Table 8.2 gives an overview of the symbols that are used. The description follows a top-down path, because it is easier to explain than the actual computation, which works bottom-up.

Overall Objective Value

The overall objective value that is computed by the ExpObjEval experiment type is a weighted mean of quantity objective values, as is shown in (8.17). The quantities that are to be used in the objective evaluation process, and the weights that are to be used in the computation of the overall objective value can be specified by the user.

$$J = \frac{\sum_{i=1}^{n_q} w_{q_i} \times J_{q_i}}{\sum_{i=1}^{n_q} w_{q_i}} \quad (8.17)$$

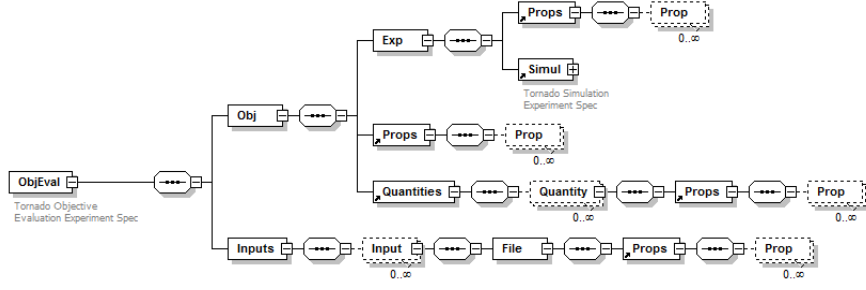


Figure 8.13: ExpObjEval XML Schema Definition

Quantity Objective Value

Each quantity objective value is a weighted mean of sub-objective differences, as is shown in (8.18). Sub-objective differences are obtained by computing sub-objective values (*e.g.*, minimum, maximum, mean, *etc*) and comparing them with a desired value. For the comparison, four different error criteria can be used: absolute error, relative error, squared error and squared relative error. The weights, sub-objective desired values and error criteria are all user-configurable.

$$J_{q_i} = \begin{cases} \frac{\sum_{j=1}^{n_o} w_{o_{i,j}} \times |J_{o_{i,j}} - \hat{J}_{o_{i,j}}|}{\sum_{j=1}^{n_o} w_{o_{i,j}}} \\ \frac{\sum_{j=1}^{n_o} w_{o_{i,j}} \times \left| \frac{J_{o_{i,j}} - \hat{J}_{o_{i,j}}}{J_{o_{i,j}}} \right|}{\sum_{j=1}^{n_o} w_{o_{i,j}}} \\ \sqrt{\frac{\sum_{j=1}^{n_o} w_{o_{i,j}} \times (J_{o_{i,j}} - \hat{J}_{o_{i,j}})^2}{\sum_{j=1}^{n_o} w_{o_{i,j}}}} \\ \sqrt{\frac{\sum_{j=1}^{n_o} w_{o_{i,j}} \times \left(\frac{J_{o_{i,j}} - \hat{J}_{o_{i,j}}}{J_{o_{i,j}}} \right)^2}{\sum_{j=1}^{n_o} w_{o_{i,j}}}} \end{cases} \quad (8.18)$$

Sub-objective Value of Quantity

As was shown in (8.18), multiple sub-objective values can be used for each quantity. An overview of the different types of sub-objectives is given in (8.19). Note that a weight set to 0 excludes a sub-objective from being used. $J_{o_i}^{Min}$ through $J_{o_i}^{Int}$ are implemented through the Common Library. $J_{o_i}^{End}$ and $J_{o_i}^{Time(\hat{t})}$ are trivial. $J_{o_i}^{MeanDiff}$ through $J_{o_i}^{PercUBV(u)}$ are explained further.

Table 8.2: ExpObjEval Symbol Descriptions

Symbol	Description
J	Overall objective value
J_{min}	Minimum objective value
J_{max}	Maximum objective value
J_{q_i}	Objective value of quantity i
$J_{o_i,j}$	Value of sub-objective j of quantity i
$\hat{J}_{o_i,j}$	Desired value of sub-objective j of quantity i
w_{q_i}	Weight of quantity i
$w_{o_i,j}$	Weight of sub-objective j of quantity i
$w_{s_i,l}$	Weight of reference time series l of quantity i
$w_{t_l,m}$	Weight of time point m of reference time series l
n_s	Number of reference time series
n_o	Number of quantity sub-objectives
n_t	Number of time points
n_r	Number of runs since last reset
t	Time
\hat{y}	Reference value
y	Simulated value
$J_{o_i}^{Min}$	Minimum value of quantity i
$J_{o_i}^{Max}$	Maximum value of quantity i
$J_{o_i}^\mu$	Mean of quantity i
$J_{o_i}^\sigma$	Standard deviation of quantity i
$J_{o_i}^{Median}$	Median of quantity i
$J_{o_i}^{Percentile(p)}$	p -th percentile of quantity i
$J_{o_i}^{Int}$	Integral of quantity i
$J_{o_i}^{End}$	End-value of quantity i
$J_{o_i}^{Time(\hat{t})}$	Value at \hat{t} of quantity i
$J_{o_i}^{MeanDiff}$	Mean difference with respect to reference time series of quantity i
$J_{o_i}^{MaxDiff}$	Maximum difference with respect to reference time series of quantity i
$J_{o_i}^{TIC}$	Theil's Inequality Coefficient with respect to reference time series of quantity i
$J_{o_i}^{NumLBV(l)}$	Number of violations of lower bound l for quantity i
$J_{o_i}^{NumUBV(u)}$	Number of violations of upper bound u violations for quantity i
$J_{o_i}^{PercLBV(l)}$	Percentage of time in violation of lower bound l for quantity i
$J_{o_i}^{PercUBV(u)}$	Percentage of time in violation of upper bound u for quantity i

$$\begin{aligned}
J_{o_i,1} &= J_{o_i}^{Min} = Min_y \\
J_{o_i,2} &= J_{o_i}^{Max} = Max_y \\
J_{o_i,3} &= J_{o_i}^\mu = \mu_y \\
J_{o_i,4} &= J_{o_i}^\sigma = \sigma_y \\
J_{o_i,5} &= J_{o_i}^{Median} = Median_y \\
J_{o_i,6} &= J_{o_i}^{Percentile(p)} = P_{y,p} \\
J_{o_i,7} &= J_{o_i}^{Int} = Int_{t,y} \\
J_{o_i,8} &= J_{o_i}^{End} = y_i(t_{n_t}) \\
J_{o_i,9} &= J_{o_i}^{Time(\hat{t})} = y_i(\hat{t}) \\
J_{o_i,10} &= J_{o_i}^{MeanDiff} \\
J_{o_i,11} &= J_{o_i}^{MaxDiff} \\
J_{o_i,12} &= J_{o_i}^{TIC} \\
J_{o_i,13} &= J_{o_i}^{NumLBV(l)} \\
J_{o_i,14} &= J_{o_i}^{NumUBV(u)} \\
J_{o_i,15} &= J_{o_i}^{PercLBV(l)} \\
J_{o_i,16} &= J_{o_i}^{PercUBV(u)}
\end{aligned} \tag{8.19}$$

Mean Difference The $J_{o_i}^{MeanDiff}$ sub-objective allows for simulated trajectories to be compared with reference time series. For the comparison of a particular simulated trajectory, more than one reference series can be used. The value of the $J_{o_i}^{MeanDiff}$ sub-objective is therefore defined as a weighted mean of the mean differences between the simulated trajectory and each of the reference series, as is expressed by (8.20). Again, the user has full control over the weights that are used.

$$J_{o_i}^{MeanDiff} = \frac{\sum_{l=1}^{n_s} w_{s_i,l} \times J_{s_i,l}^{MeanDiff}}{\sum_{l=1}^{n_s} w_{s_i,l}} \quad (8.20)$$

The mean difference between a simulated trajectory and a reference series is computed according to (8.21). Differences are computed at each time point and then aggregated. For the difference computation at individual time points, four alternative error criteria can be used: absolute error, relative error, squared error and squared relative error. The weights that are used here are time-dependent and can also be provided by the user.

$$J_{s_i,l}^{MeanDiff} = \begin{cases} \frac{\sum_{m=1}^{n_{t_l}} w_{t_l,m} \times |y_i(t_{l,m}) - \hat{y}_i(t_{l,m})|}{\sum_{m=1}^{n_{t_l}} w_{t_l,m}} \\ \frac{\sum_{m=1}^{n_{t_l}} w_{t_l,m} \times \left| \frac{y_i(t_{l,m}) - \hat{y}_i(t_{l,m})}{y_i(t_{l,m})} \right|}{\sum_{m=1}^{n_{t_l}} w_{t_l,m}} \\ \sqrt{\frac{\sum_{m=1}^{n_{t_l}} w_{t_l,m} \times (y_i(t_{l,m}) - \hat{y}_i(t_{l,m}))^2}{\sum_{m=1}^{n_{t_l}} w_{t_l,m}}} \\ \sqrt{\frac{\sum_{m=1}^{n_{t_l}} w_{t_l,m} \times \left(\frac{y_i(t_{l,m}) - \hat{y}_i(t_{l,m})}{y_i(t_{l,m})} \right)^2}{\sum_{m=1}^{n_{t_l}} w_{t_l,m}}} \end{cases} \quad (8.21)$$

In the current implementation, the time points at which differences between a reference series and a simulated trajectory are computed, are determined by the reference series. As a result, one of the interpolation methods implemented by the Common Library is to be applied to the simulated trajectory. However, if needed, this behavior could easily be reversed in order to compute differences at the time points provided by the simulated trajectory. Another option would be to establish a collection of time points that is the union of the time points provided by the reference and the simulated trajectories and to perform interpolation in both trajectories for each of these points.

Maximum Difference The $J_{o_i}^{MaxDiff}$ sub-objective is another way for comparing simulated trajectories with reference time series. For the comparison of a particular simulated trajectory, more than one reference series can be used. The value of the $J_{o_i}^{MaxDiff}$ sub-objective is therefore defined as the maximum of the maximum differences between the simulated trajectory and each of the reference series, as is expressed by (8.22).

$$J_{o_i}^{MaxDiff} = \max_{l=1}^{n_s} J_{s_i,l}^{MaxDiff} \quad (8.22)$$

The maximum difference between a simulated trajectory and a reference series is computed according to (8.23). Differences are computed at each time point and then aggregated. For the difference computation at individual time points, four alternative error criteria can again be used: absolute error, relative error, squared error and squared relative error.

$$J_{s_i,l}^{MaxDiff} = \begin{cases} \max_{m=1}^{n_{t_l}} |y_i(t_{l,m}) - \hat{y}_i(t_{l,m})| \\ \max_{m=1}^{n_{t_l}} \left| \frac{y_i(t_{l,m}) - \hat{y}_i(t_{l,m})}{y_i(t_{l,m})} \right| \\ \max_{m=1}^{n_{t_l}} (y_i(t_{l,m}) - \hat{y}_i(t_{l,m}))^2 \\ \max_{m=1}^{n_{t_l}} \left(\frac{y_i(t_{l,m}) - \hat{y}_i(t_{l,m})}{y_i(t_{l,m})} \right)^2 \end{cases} \quad (8.23)$$

Theil's Inequality Coefficient Theil's Inequality Coefficient (TIC) (Leuthold, 1975) is a third alternative for comparing simulated trajectories with reference time series. The interesting aspect of this measure is that it always provides a value between 0 and 1, where a value close to 0 refers to good correspondence, and a value close to 1 points to total disagreement. Again, for the comparison of a particular simulated trajectory, more than one reference series can be used. The value of the $J_{o_i}^{TIC}$ sub-objective is therefore defined as a weighted mean of the TIC values representing the differences between the simulated trajectory and each of the reference series, as is expressed by (8.25). Once more, the user has full control over the weights that are used.

$$J_{o_i}^{TIC} = \frac{\sum_{l=1}^{n_s} w_{s_i,l} \times J_{s_i,l}^{TIC}}{\sum_{l=1}^{n_s} w_{s_i,l}} \quad (8.24)$$

The TIC value representing the difference between a simulated trajectory and a reference series is computed according to (8.25).

$$J_{s_i,l}^{TIC} = \frac{\sqrt{\frac{\sum_{m=1}^{n_{t_l}} (y_i(t_{l,m}) - \hat{y}_i(t_{l,m}))^2}{n_{t_l}}}}{\sqrt{\frac{\sum_{m=1}^{n_{t_l}} (y_i(t_{l,m}))^2}{n_{t_l}} + \frac{\sum_{m=1}^{n_{t_l}} (\hat{y}_i(t_{l,m}))^2}{n_{t_l}}}}} \quad (8.25)$$

Number of Lower Bound and Upper Bound Violations The $J_{o_i}^{NumLBV(l)}$ sub-objective allows for computing the number of exceedances of a configurable lower bound for a simulated trajectory. Algorithm 4 shows how this is accomplished. The $J_{o_i}^{NumUBV(u)}$ sub-objective allows for computing the number of exceedances of a configurable upper bound for a simulated trajectory. The algorithm for this is analogous to Algorithm 4 and is therefore not given.

Algorithm 4 Computation of the Number of Lower Bound Violations

```

NumViolations := 0
InViolation := false
for  $i := 1$  to  $n_t$  do
  if  $y_i(t_i) < l$  and  $\neg InViolation$  then
    NumViolations := NumViolations + 1
    InViolation := true
  else
    if  $y_i(t_i) \geq l$  then
      InViolation := false
    end if
  end if
end for

 $J_{o_i}^{NumLBV(l)} := NumViolations$ 

```

Percentage of Time in Violation of Lower Bound and Upper Bound The $J_{oi}^{PercLBV(l)}$ sub-objective allows for computing the percentage of time a system is in violation of a configurable lower bound. Algorithm 5 shows how this is accomplished. The $J_{oi}^{PercUBV(u)}$ sub-objective allows for computing the percentage of time a system is in violation of a configurable upper bound. The algorithm for this is analogous to Algorithm 5 and is therefore not given.

Algorithm 5 Computation of the Percentage of Time in Violation of Lower Bound

```

TotalTimeInViolation := 0
StartTimeViolation := 0
InViolation := false
for i := 1 to nt do
  if yi(ti) < l and !InViolation then
    StartTimeViolation := ti
    InViolation := true
  else
    if yi(ti) ≥ l and InViolation then
      StopTimeViolation := ti
      InViolation := false
      TotalTimeInViolation :=
        TotalTimeInViolation + StopTimeViolation - StartTimeViolation
    end if
  end if
end for

JoiPercLBV(l) := TotalTimeInViolation / (tnt - t1) × 100

```

Overall Minimum and Maximum Objective Value Next to aggregating all quantity objectives into one overall objective value, the ExpObjEval experiment also maintains some information on the evolution of the overall objective over previous ExpObjEval runs: the minimum objective value reached so far is given by (8.26), the maximum objective value is in (8.27).

$$J_{min} = \min_{r=1}^{n_r} J(r) \quad (8.26)$$

$$J_{max} = \max_{r=1}^{n_r} J(r) \quad (8.27)$$

8.3.4 ExpScen: Scenario Analysis

The ExpScen experiment type allows for performing a scenario analysis, *i.e.*, the execution of a dynamic simulation experiment a number of times using different parameter values, input variable values and/or initial conditions. During each run, the simulated trajectories of a number of selected quantities are stored in an internal buffer. Subsequently, the buffer can be used to compute a variety of objective values. The values of parameters, input variables and/or initial conditions that are to be used for each simulation run can be determined in three different ways:

- **Spacing:** Generates values through linear or logarithmic equidistant placement between a lower bound and upper bound
- **Sampling:** Generates values through sampling from a number of standard statistical distributions.
- **Manual Placement:** Allows for the user to specify values explicitly

The ExpScen experiment is a 3-level compound experiment, for it contains an ExpObjEval experiment (*ExpObjEval*), either through embedding or through referral. This ExpObjEval experiment in turn contains an ExpSimul experiment. The ExpScen experiment uses a callback for messages (*CallbackMessage*), a callback to report on new execution runs (*CallbackRunNo*) and a callback to poll for user interrupts (*CallbackStop*). The ExpScen process is steered by an objective (*ScenObj*) that loops through all ExpScen variable (*ScenVar*) combinations and runs an ExpObjEval experiment for each of these. The order in which variable value combinations are used, and the total number of variable value combinations are determined by a pluggable algorithm (*SolveScen*). After the computation of objective values for each ExpObjEval run, results can be stored in output files (*ScenOutputFile*) or sent to a graphical output channel (*ScenOutputPlot*). If desired, buffers could be added at a later stage. Finally, there is also a log file (*ScenLogFile*) that is used for unformatted textual logging information on the progress of the scenario analysis.

An ER diagram of the internal representation of the ExpScen experiment type is in Figure 8.14. An XML Schema Definition of the persistent representation of this experiment type can be found in Figure 8.15.

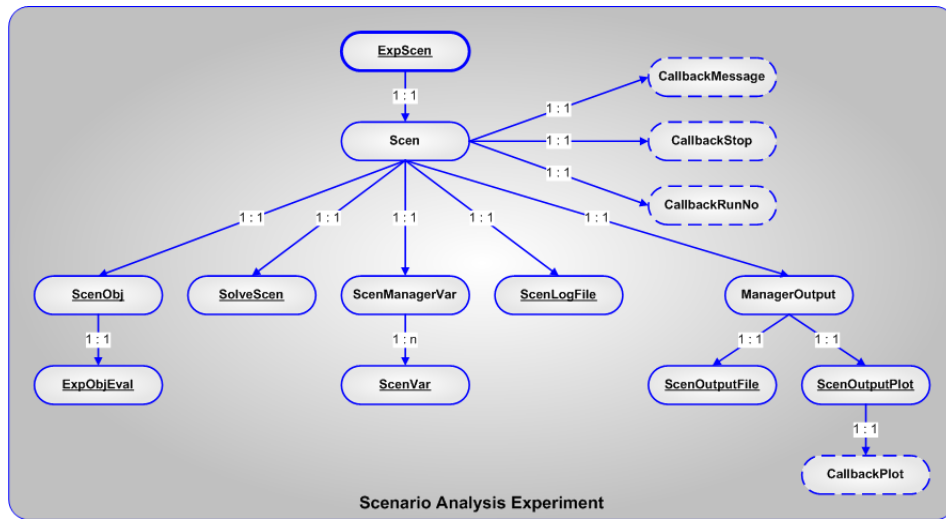


Figure 8.14: ExpScen ER Diagram

In the sequel, a full description is given of the scenario analysis process. Table 8.3 presents an overview of the symbols that are used. The description follows a top-down path, starting with the generation of the ExpScen variable values.

Distribution Methods

Below the various ways of determining values for parameters, input variables and/or initial conditions for the simulation contained in the ExpScen experiment are given. As mentioned before, these so-called distribution methods can be based on spacing, sampling or manual placement.

Linear Spacing Linear spacing provides equidistant linear placement of values between a lower bound and upper bound. Linear spacing either works on the basis of a user-supplied number of values, or a user-supplied interval.

In case the desired number of values n_{s_i} for variable i is given, the interval between these values is determined by (8.28).

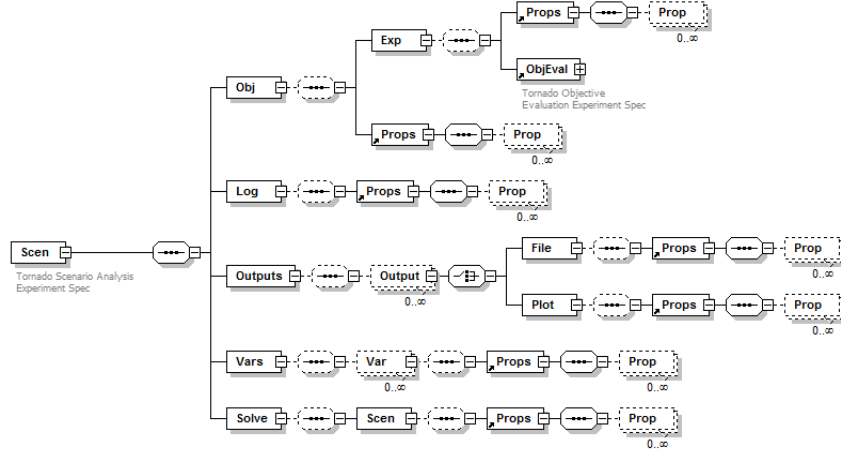


Figure 8.15: ExpScen XML Schema Definition

Table 8.3: ExpScen Symbol Descriptions

Symbol	Description
$y_{i,j}$	Computed value of shot j of variable i
$y'_{i,j}$	User-supplied value of shot j of variable i
n_v	Number of variables
n_{s_i}	Number of shots for variable i
y_i^{\min}	Minimum of variable i
y_i^{\max}	Maximum of variable i
μ_i	Mean of variable i
σ_i	Standard deviation of variable i
λ_i	Rate of variable i
k_i	Shape of variable i
Δ_i	Spacing for variable i

$$\Delta_i = \frac{y_i^{\max} - y_i^{\min}}{n_{s_i} - 1} \quad (8.28)$$

In case the desired interval Δ_i for variable i is given, the number of values is determined by (8.29).

$$n_{s_i} = \text{int} \left(\frac{y_i^{\max} - y_i^{\min}}{\Delta_i} + 1 \right) \quad (8.29)$$

Once n_{s_i} and Δ_i are determined, the individual values are given by the recursive scheme shown in (8.30).

$$y_{i,j} = \begin{cases} y_i^{\min} & j = 1 \\ y_{i,j-1} + \Delta_i & j = 2, \dots, n_{s_i} \end{cases} \quad (8.30)$$

In case $\text{mod}(y_i^{\max} - y_i^{\min}, \Delta_i) \neq 0$, a user-supplied flag indicates the strategy to follow:

- **Include Upper Bound:** Include the upper bound as last value, as in (8.31)

$$y_{i,n_{s_i}+1} = y_i^{max} \quad (8.31)$$

- **Include Computed:** Include as last value a value that is computed according to the recursive scheme, as in (8.32)

$$y_{i,n_{s_i}+1} = y_{i,n} + \Delta_i \quad (8.32)$$

- **Exclude:** Do not include an additional value, hence no $y_{i,n_{s_i}+1}$ is defined

Logarithmic Spacing Logarithmic spacing provides equidistant logarithmic placement of values between a lower bound and upper bound. Logarithmic spacing either works on the basis of a user-supplied number of values, or a user-supplied interval (actually, it is a factor or multiplier).

In case the desired number of values n_{s_i} for variable i is given, the interval between these values is determined by (8.33).

$$\Delta = (y_i^{max} / y_i^{min})^{\frac{1}{n_{s_i}-1}} \quad (8.33)$$

In case the desired interval Δ_i for variable i is given, the number of values is determined by (8.34).

$$n_{s_i} = \text{int} \left(\frac{\log(y_i^{max} / y_i^{min})}{\log(\Delta_i)} + 1 \right) \quad (8.34)$$

Once n_{s_i} and Δ_i are determined, the individual values are given by the recursive scheme shown in (8.35).

$$y_{i,j} = \begin{cases} y_i^{min} & j = 1 \\ y_{i,j-1} \times \Delta_i & j = 2, \dots, n_{s_i} \end{cases} \quad (8.35)$$

In case $\text{mod}(\log(y_i^{max} / y_i^{min}), \log(\Delta_i)) \neq 0$, a user-supplied flag indicates the strategy to follow:

- **Include Upper Bound:** Include the upper bound as last value, as in (8.36)

$$y_{i,n_{s_i}+1} = y_i^{max} \quad (8.36)$$

- **Include Computed:** Include as last value a value that is computed according to the recursive scheme, as in (8.37)

$$y_{i,n_{s_i}+1} = y_{i,n_{s_i}} \times \Delta_i \quad (8.37)$$

- **Exclude:** Do not include an additional value, hence no $y_{i,n_{s_i}+1}$ is defined

Uniform Distribution A uniform distribution can be used to generate a user-supplied number of values n_{s_i} for variable i , as in (8.38). Sampling from uniform distributions is implemented through the Common Library.

$$y_{i,j} = \text{RandUniform}(y_i^{min}, y_i^{max}) \quad j = 1, \dots, n_{s_i} \quad (8.38)$$

Truncated Normal Distribution A truncated Normal distribution can be used to generate a user-supplied number of values n_{s_i} for variable i , as in (8.39). Sampling from truncated Normal distributions is implemented through the Common Library.

$$y_{i,j} = RandTruncNormal(\mu_i, \sigma_i, y_i^{min}, y_i^{max}) \quad j = 1, \dots, n_{s_i} \quad (8.39)$$

Truncated Lognormal Distribution A truncated Lognormal distribution can be used to generate a user-supplied number of values n_{s_i} for variable i , as in (8.40). Sampling from truncated Lognormal distributions is implemented through the Common Library.

$$y_{i,j} = RandTruncLogNormal(\mu_i, \sigma_i, y_i^{min}, y_i^{max}) \quad j = 1, \dots, n_{s_i} \quad (8.40)$$

Truncated Exponential Distribution A truncated exponential distribution can be used to generate a user-supplied number of values n_{s_i} for variable i , as in (8.41). Sampling from truncated exponential distributions is implemented through the Common Library.

$$y_{i,j} = RandTruncExp(\lambda_i, y_i^{min}, y_i^{max}) \quad j = 1, \dots, n_{s_i} \quad (8.41)$$

Truncated Weibull Distribution A truncated Weibull distribution can be used to generate a user-supplied number of values n_{s_i} for variable i , as in (8.42). Sampling from truncated Weibull distributions is implemented through the Common Library.

$$y_{i,j} = RandTruncWeibull(\lambda_i, k_i, y_i^{min}, y_i^{max}) \quad j = 1, \dots, n_{s_i} \quad (8.42)$$

Triangular Distribution A truncated triangular distribution can be used to generate a user-supplied number of values n_{s_i} for variable i , as in (8.43). Sampling from truncated triangular distributions is implemented through the Common Library.

$$y_{i,j} = RandTriangular(\mu_i, y_i^{min}, y_i^{max}) \quad j = 1, \dots, n_{s_i} \quad (8.43)$$

Manual Placement

$$y_{i,j} = y'_{i,j} \quad j = 1, \dots, n_{s_i} \quad (8.44)$$

Generation of an Exhaustive List of Variable Value Vector Index Combinations

Once values for individual variables have been determined through spacing, sampling or manual placement, an exhaustive list of value combinations is created. More specifically, it is an exhaustive list of variable value vector index combinations. Algorithm 6 depicts the process that is followed to establish this list.

Determining Filtering and Execution Order

The ExpScen experiment allows for dynamically-loadable algorithm plug-ins to be used to order, and possibly also filter, the list of variable value vector index combinations. In this case, it can be determined for which ExpScen variable value combinations simulations will be executed, and also what the order of execution will be. At the moment, five different algorithms (referred to as scenario analysis solvers) are available, which have been named Sequential, Random, Fixed, Grid and Cross.

Algorithm 6 Generation of an Exhaustive List of Variable Value Vector Index Combinations

```

Indices := {}
for i := 1 to nv do
  if i = 1 then
    for j := 1 to nsi do
      Indices.append(j)
    end for
  else
    NewIndices := {}
    for j := 1 to nsi do
      for k := 1 to Indices.size() do
        Temp := Indices[k]
        Temp.append(j)
        NewIndices.append(Temp)
      end for
    end for
    Indices := NewIndices
  end if
end for

```

Sequential The Sequential solver simply steps through the list of variable value vector index combinations as it has been generated. Consequently, it will run a simulation for each tuple and maintains the ordering as it was generated through the application of Algorithm 6. The total number of simulations that is run is given by (8.45). The value vectors that are ultimately applied to the simulation experiment (represented in (8.46)) are generated by the Sequential solver as in Algorithm 7.

$$n_s = \prod_{i=1}^{n_v} n_{s_i} \quad (8.45)$$

$$\underline{v}_k = (v_{k,1}, \dots, v_{k,n_v}) \quad k = 1, \dots, n_s \quad (8.46)$$

Algorithm 7 Generation of Sequentially-ordered Value Vectors

```

v.resize(ns, nv)
for k := 1 to ns do
  for i := 1 to nv do
    v[k, i] := y[i, Indices[k, i]]
  end for
end for

```

Random The Random solver is similar to the Sequential solver in the sense that it will eventually handle all variable value vector index combinations. However, the order in which this is done is determined by a random number generator with a uniform distribution. The total number of simulations that is run is given by (8.45). The value vectors that are ultimately applied to the simulation experiment (represented in (8.46)) are generated by the Random solver as in Algorithm 8.

Fixed The Fixed solver does not run simulations for every variable value vector index combination. In fact, it does not even use the exhaustive list of variable value vector index combinations. Instead it

Algorithm 8 Generation of Randomly-ordered Value Vectors

```

v.resize( $n_s, n_v$ )
 $k := 0$ 
while Indices  $\neq \{\}$  do
     $k := k + 1$ 
    Pos := RandUniform(1, Indices.size())
    for  $i := 1$  to  $n_v$  do
         $v[k, i] := y[i, \textit{Indices}[\textit{Pos}, i]]$ 
    end for
    Indices.erase(Pos)
end while

```

requires all variable value vectors to be of equal size and simply generates value vectors for each shot by taking elements from variable value vectors at the same position. In (8.47) it is shown that the overall number of shots is the same as the number of shots for each variable. Algorithm 9 shows how the value vectors that are ultimately applied to the simulation experiment are generated.

$$\forall i = 1, \dots, n_v : n_{s_i} = n_s \quad (8.47)$$

Algorithm 9 Generation of Fixed Value Vectors

```

v.resize( $n_s, n_v$ )
for  $k := 1$  to  $n_s$  do
    for  $i := 1$  to  $n_v$  do
         $v[k, i] := y[i, k]$ 
    end for
end for

```

Grid The Grid solver generates value vectors by stepping through the list of variable value vector index combinations. Subsequently, the list of value vectors is sorted according to their distance to a reference location in the value vector space, represented by (8.48).

$$\underline{r} = (r_1, \dots, r_{n_v}) \quad (8.48)$$

A user-supplied flag indicates whether the reference vector itself is also to be added to the value vector list. In case it is not, the number of value vectors, and hence the total number of simulation to be run, is again as in (8.45), otherwise the total number of shots is given by (8.49).

$$n_s = 1 + \prod_{i=1}^{n_v} n_{s_i} \quad (8.49)$$

As a metric for the distance between a value vector and the reference vector, the Euclidian distance is used, as represented by (8.50).

$$\delta_k = \sqrt{\sum_{i=1}^{n_v} (v_{k,i} - r_i)^2} \quad (8.50)$$

The generation of the value vector list proceeds as in Algorithm 10, *i.e.*, first an unordered list is generated from the list of variable value vector index combinations, subsequently the list is sorted according

to the δ criterion presented above, and finally a user-supplied flag is consulted to determine whether the reference vector is to be prepended to the list.

Algorithm 10 Generation of Grid-ordered Value Vectors

```

v.resize( $n_s, n_v$ )
for  $k := 1$  to  $n_s$  do
  for  $i := 1$  to  $n_v$  do
     $v[k, i] := y[i, \text{Indices}[k, i]]$ 
  end for
end for
v.sort( $\delta$ )
if IncludeRef then
  v.prepend( $r$ )
end if

```

Cross The Cross solver is similar to the Grid solver in the sense that it also uses a δ criterion based on the distance to a reference location to sort the list of value vectors. However, unlike the Grid solver it does not handle all possible combinations of variable values. In fact, as the Fixed solver, it does not use the list of variable value vector index combinations at all. Instead it generates value vectors by starting from the reference vector and letting all elements of the reference vector, one after the other, vary using the values of the corresponding variable value vectors. The generated value vectors are sorted according to their distance to the reference vector. A user-supplied flag indicates whether the reference vector itself is to be added to the list. In case it is not, the total number of simulations to be run is given by (8.51).

$$n_s = \sum_{i=1}^{n_v} n_{s_i} \quad (8.51)$$

In case the reference vector is prepended to the list, the total number of simulations to be run is given by (8.52).

$$n_s = 1 + \sum_{i=1}^{n_v} n_{s_i} \quad (8.52)$$

Using the same representation for reference vector as in (8.47) and the same definition for δ as in (8.50), Algorithm 11 depicts the procedure that is followed to generate the value vectors in case of the Cross solver.

In Figure 8.16 is an example of the application of the Sequential, Random, Grid, Fixed and Cross solvers to a scenario analysis with 2 variables for which 4 values have been generated through linear spacing. The numbers in the boxes indicate the order in which the points from the variable space will be used.

8.3.5 ExpMC: Monte Carlo Analysis

The ExpMC experiment type has a somewhat misleading name since it is not the only experiment type that performs Monte Carlo simulation, *i.e.*, iterative execution of a dynamic simulation using randomly-generated parameter values, input values and/or initial conditions. Also the ExpScen experiment type allows for this using appropriate distribution method settings. The main difference between ExpMC and ExpScen however is that in the case of ExpMC the total number of shots is determined before the sampling occurs, and that sampling is done directly on the multi-dimensional ExpMC variable vector

Algorithm 11 Generation of Cross-ordered Value Vectors

```
v.resize( $n_s, n_v$ )
 $k := 1$ 
for  $i := 1$  to  $n_v$  do
  for  $j := 1$  to  $n_{s_i}$  do
     $l := 1$ 
    while  $l < i$  do
       $k := k + 1$ 
       $v[k, l] := r[l]$ 
    end while
     $k := k + 1$ 
     $v[k, i] := y[i, j]$ 
     $l := i + 1$ 
    while  $l \leq n_v$  do
       $k := k + 1$ 
       $v[k, l] := r[l]$ 
    end while
  end for
end for
v.sort( $\delta$ )
if IncludeRef then
  v.prepend( $r$ )
end if
```

space instead of on individual ExpMC variables. The goal is to obtain a set of shots that is evenly distributed over the entire variable vector space, instead of clustered in one particular area (which could be - intentionally or non-intentionally - the result of the type of sampling that is done in ExpScen). Also, in the case of ExpMC no sorting or filtering can be applied to the variable value vectors. The solvers that are available for ExpMC experiments therefore do not implement sorting and/or filtering strategies, but implement various algorithms for selecting a specified number of evenly distributed points from a multi-dimensional space. An overview of the Monte Carlo solvers that have been implemented as a solver plug-in can be found in Appendix D.

From a structural point of view, ExpMC is very similar to ExpScen. The ExpMC experiment is a 3-level compound experiment, for it contains an ExpObjEval experiment (*ExpObjEval*), either through embedding or through referral. This ExpObjEval experiment in turn contains an ExpSimul experiment. The ExpMC experiment uses a callback for messages (*CallbackMessage*), a callback to report on new execution runs (*CallbackRunNo*) and a callback to poll for user interrupts (*CallbackStop*). The ExpMC process is steered by an objective (*MCObj*) that loops through all ExpMC variable (*MCVar*) vectors and runs an ExpObjEval experiment for each of these. After the computation of objective values for each ExpObjEval run, results can be stored in output files (*MCOutputFile*) or sent to a graphical output channel (*MCOutputPlot*). If desired, buffers could be added at a later stage. Finally, there is also a log file (*MCLogFile*) that is used for unformatted textual logging information on the progress of the Monte Carlo analysis.

An ER diagram of the internal representation of the ExpMC experiment type is in Figure 8.17. An XML Schema Definition of the persistent representation of this experiment type can be found in Figure 8.18.

In the sequel, a full description is given of the Monte Carlo analysis process. Table 8.4 presents an overview of the symbols that are used.

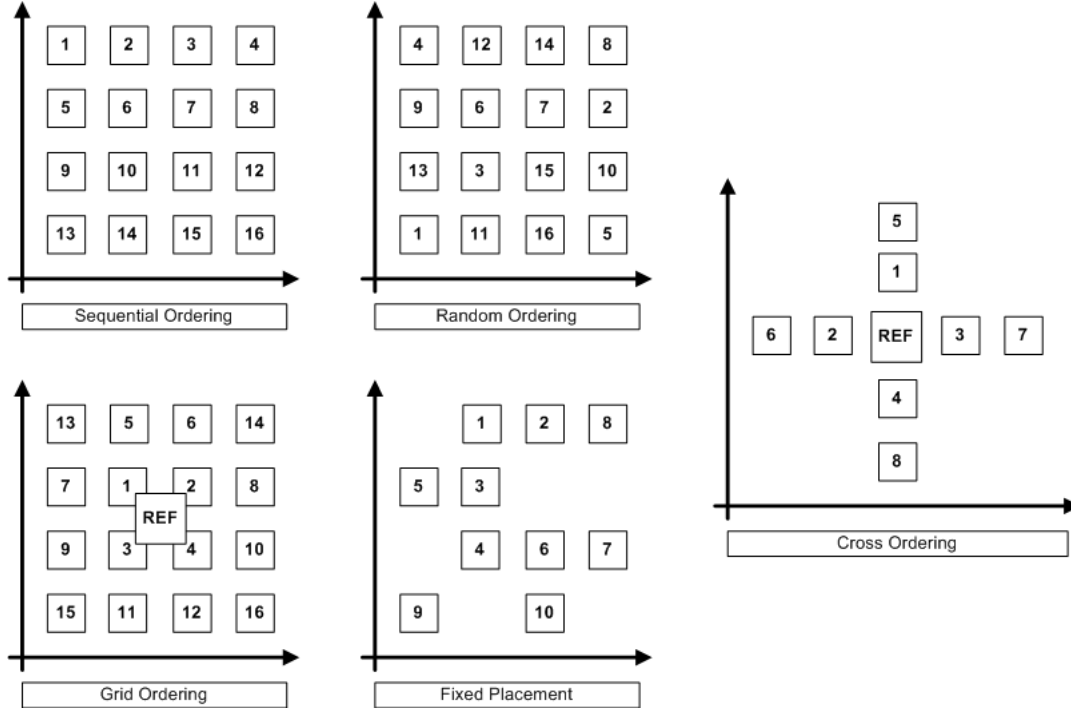


Figure 8.16: ExpScen Solver Procedures

Selection of Points from the Multi-Dimensional Variable Space

Using the user-supplied number of variables n_v and the user-supplied number of shots n_s , one of several so-called Monte Carlo analysis solvers are activated (CVT, IHS, LHS or PR). These methods are implemented as dynamically-loadable plug-ins and return a matrix \underline{r} , as in (8.53) and (8.54). Each value vector that is contained in \underline{r} represents a point from the variable space that was chosen at random by the Monte Carlo analysis solver. The latter is assumed to ensure that the points are chosen in such a way that an even spread is obtained, which implies that in fact Latin Hypercube Sampling (LHS) is performed.

$$\underline{r} = \begin{cases} CVT(n_v, n_s; P_{CVT}) \\ IHS(n_v, n_s; P_{IHS}) \\ LHS(n_v, n_s; P_{LHS}) \\ PR(n_v, n_s; P_{PR}) \end{cases} \quad (8.53)$$

$$0 \leq r_{i,j} \leq 1 \quad i = 1, \dots, n_v; j = 1, \dots, n_s \quad (8.54)$$

Application of Distributions

The points obtained during the first step of the procedure are used to retrieve values from a number of statistical distributions. In fact, for each ExpMC variable, a specific distribution can be configured. The final variable value vectors \underline{v}_j that are applied to the simulation experiments that are run from the ExpMC experiment, are then given by (8.55) and (8.56).

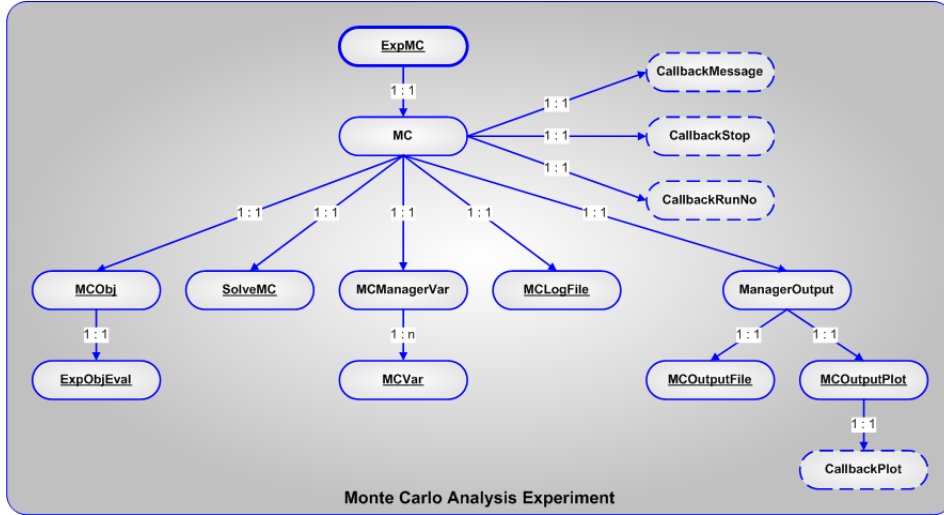


Figure 8.17: ExpMC ER Diagram

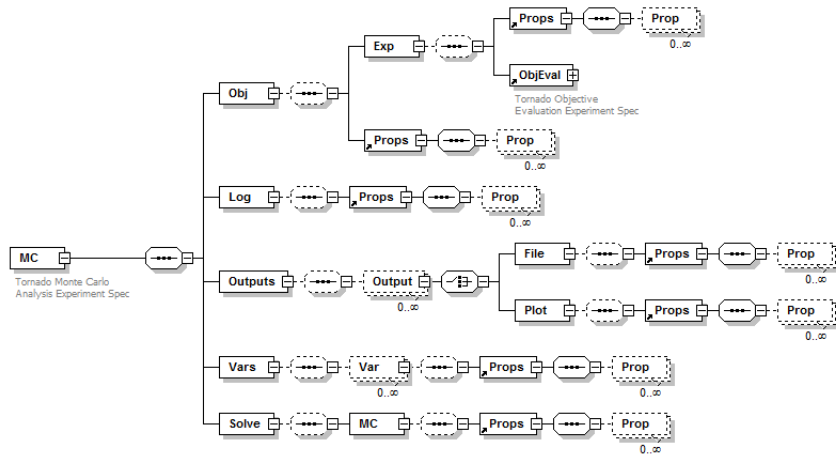


Figure 8.18: ExpMC XML Schema Definition

$$\forall j = 1, \dots, n_s : y_{i,j} = \begin{cases} \text{RandUniform}(r_{i,j}, y_i^{\min}, y_i^{\max}) \\ \text{RandNormal}(r_{i,j}, \mu_i, \sigma_i) \\ \text{RandLogNormal}(r_{i,j}, \mu_i, \sigma_i) \\ \text{RandExp}(r_{i,j}, \lambda_i) \\ \text{RandWeibull}(r_{i,j}, \lambda_i, k_i) \\ \text{RandTriangular}(r_{i,j}, \text{Mode}_i, y_i^{\min}, y_i^{\max}) \end{cases} \quad (8.55)$$

$$\underline{y}_j = (y_{1,j}, \dots, y_{n_s,j}) \quad (8.56)$$

8.3.6 ExpSens: Sensitivity Analysis

The ExpSens experiment type allows for computing a numerical approximation of the local sensitivity of state variables, worker variables or output variables with respect to parameters, input variables and/or

Table 8.4: ExpMC Symbol Descriptions

Symbol	Description
$r_{i,j}$	Random value between 0 and 1 to be used for shot j of variable i
$y_{i,j}$	Actual value of shot j of variable i
n_v	Number of ExpMC variables
n_s	Number of shots, <i>i.e.</i> , the number of values per variable
P_{CVT}	Additional parameters for the CVT solver
P_{IHS}	Additional parameters for the IHS solver
P_{LHS}	Additional parameters for the LHS solver
P_{PR}	Additional parameters for the PR solver
y_i^{min}	Minimum value
y_i^{max}	Maximum value
μ_i	Mean
σ_i	Standard deviation
λ_i	Rate
k_i	Shape

initial conditions. The sensitivity function $S(t)$ of $Y(t)$ versus θ is defined as the partial derivative of $Y(t)$ with respect to Θ , as shown in (8.57).

$$S(t) = \frac{\partial Y(t)}{\partial \Theta} \quad (8.57)$$

The simplest way of calculating local sensitivities is to use the **finite difference approach**. This technique is also called the brute force method or indirect method (see also (De Pauw, 2005) and (Vandenbergh, 2008) for an overview). The partial derivative presented in (8.57) can mathematically also be formulated as (8.58). This equation shows that the application of the finite difference method requires the solution of the model using the nominal values of Θ , and p solutions using perturbed values $\theta_j + \Delta\theta_j$, where p is the number of parameters, input variables and/or initial conditions involved in the sensitivity analysis. It should be noted that only one value is perturbed at a time, while all others are kept to their nominal setting.

$$\frac{\partial y_i}{\partial \theta_j} = \lim_{\Delta\theta_j \rightarrow 0} \frac{y_i(t, \theta_j + \Delta\theta_j) - y_i(t, \theta_j)}{\Delta\theta_j} \quad (8.58)$$

In software applications, $\Delta\theta_j \rightarrow 0$ is not a tractable approach. One therefore has to aim for an approximate numerical solution by choosing a sufficiently small value for $\Delta\theta_j$, as in (8.59). In practice, $\Delta\theta_j$ is implemented as the nominal value of θ_j multiplied by a user-supplied perturbation factor ξ . The choice of the perturbation factor largely determines the quality of the sensitivity function. A factor that is too large will not yield accurate results, while a factor that is too small may cause numerical problems (see (De Pauw and Vanrolleghem, 2007) for a detailed discussion on the importance of choosing an appropriate perturbation factor).

$$\frac{\partial y_i}{\partial \theta_j} \approx \frac{y_i(t, \theta_j + \Delta\theta_j) - y_i(t, \theta_j)}{\Delta\theta_j} \quad (8.59)$$

The finite difference that is presented in (8.59) is a **forward difference**. However a **backward difference** can also be considered, as is shown in (8.60) and (8.61). Actually, it can be shown that the average of the forward and backward difference, which is referred to as **central difference**, often yields a better accuracy than only using forward or backward difference.

$$\frac{\partial y_i}{\partial \theta_{j+}} = \frac{y_i(t, \theta_j + \xi \theta_j) - y_i(t, \theta_j)}{\xi \theta_j} \quad (8.60)$$

$$\frac{\partial y_i}{\partial \theta_{j-}} = \frac{y_i(t, \theta_j) - y_i(t, \theta_j - \xi \theta_j)}{\xi \theta_j} \quad (8.61)$$

The ExpSens experiment allows for computing various types of sensitivity functions based on forward, backward and central differences (see below). It also generates aggregated sensitivity quality information and performs a check on the reliability of sensitivities.

The ExpSens experiment is a 2-level compound experiment since it contains an ExpSimul experiment (*ExpSimul*), either through embedding or through referral. It uses a callback for messages (*CallbackMessage*), a callback to report on new simulation runs (*CallbackRunNo*) and a callback to poll for user interrupts (*CallbackStop*). The ExpSens process is steered by an objective (*SensObj*) that loops through all sensitivity functions that are to be computed. The items that are to be perturbed during simulation runs are defined as ExpSens variables (*SensVar*), and sensitivity function definitions (*SensFunc*) determine the quantities (*SensObjQuantity*) for which the effect of small changes of the ExpSens variables need to be studied. Computed sensitivity function trajectories can be sent to files (*SensFuncOutputFile*), buffers (*SensFuncOutputBuffer*) and graphical plot channels (*SensFuncOutputPlot*). Also, the aggregated sensitivity quality information can be sent to a file (*SensOutputFile*) and to a graphical output channel (*SensOutputPlot*). Finally, unformatted textual logging information is sent to a log (*SensLogFile*).

An ER diagram of the internal representation of the ExpSens experiment type is in Figure 8.19. An XML Schema Definition of the persistent representation of this experiment type can be found in Figure 8.20.

In the sequel, a full description is given of the sensitivity analysis process. Table 8.5 presents an overview of the symbols that are used. The description starts with the computation of sensitivity functions. Subsequently follow the computation of sensitivity quality information and the sensitivity reliability check.

Backward Sensitivity

Four backward sensitivity functions are defined: Backward Absolute Sensitivity (BAS), Backward Relative Sensitivity (BRS), Backward Partial Relative Sensitivity with respect to Variable (BPRSV) and Backward Partial Relative Sensitivity with respect to Quantity (BPRSQ). These are respectively determined by (8.62), (8.63), (8.64) and (8.65)

$$BAS_{i,j}(t) = \frac{\partial y_i}{\partial \theta_{j-}} \quad (8.62)$$

$$BRS_{i,j}(t) = \frac{BAS_{i,j}(t) \times \theta_j}{y_i(t, \theta_j)} \quad (8.63)$$

$$BPRSV_{i,j}(t) = BAS_{i,j}(t) \times \theta_j \quad (8.64)$$

$$BPRSQ_{i,j}(t) = \frac{BAS_{i,j}(t)}{y_i(t, \theta_j)} \quad (8.65)$$

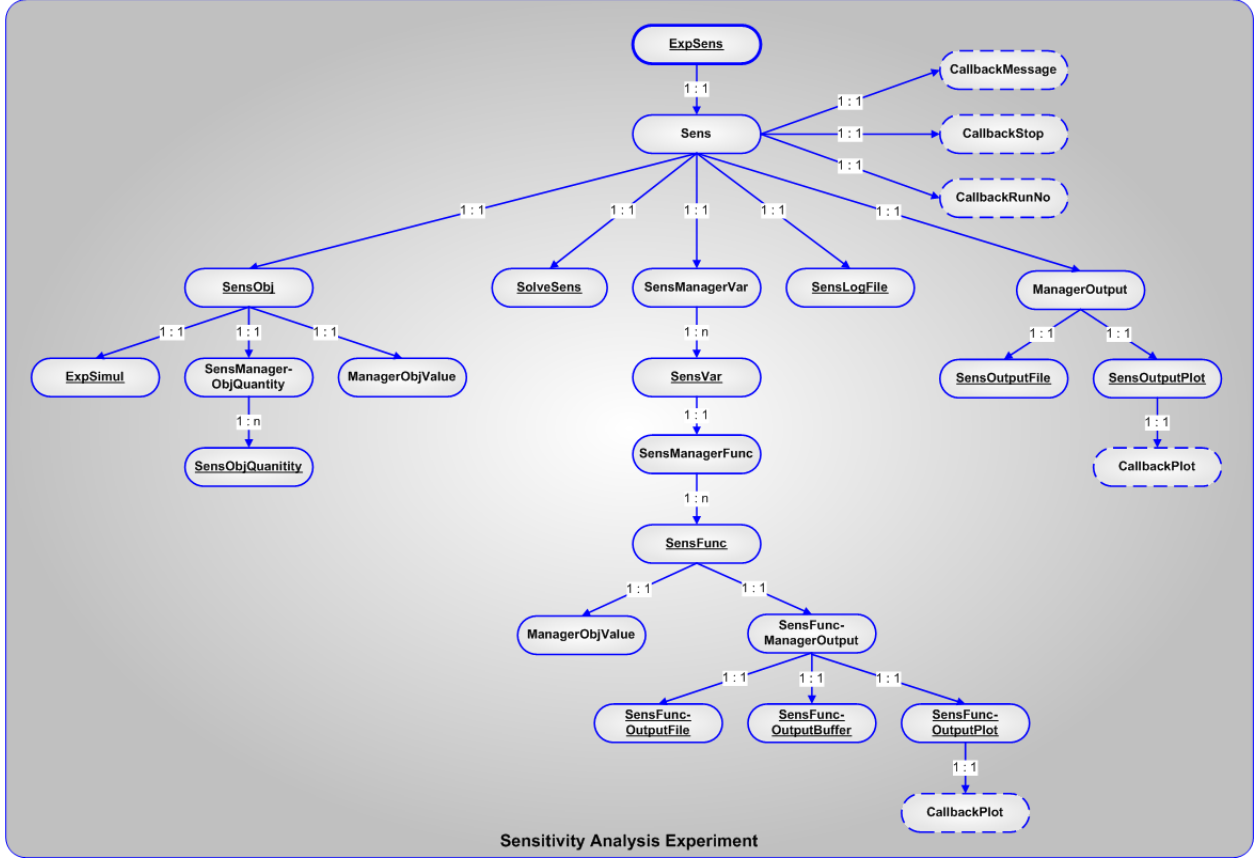


Figure 8.19: ExpSens ER Diagram

Forward Sensitivity

Four forward sensitivity functions are defined: Forward Absolute Sensitivity (FAS), Forward Relative Sensitivity (FRS), Forward Partial Relative Sensitivity with respect to Variable (FPRSV) and Forward Partial Relative Sensitivity with respect to Quantity (FPRSQ). These are respectively determined by (8.66), (8.67), (8.68) and (8.69)

$$FAS_{i,j}(t) = \frac{\partial y_i}{\partial \theta_{j+}} \quad (8.66)$$

$$FRS_{i,j}(t) = \frac{FAS_{i,j}(t) \times \theta_j}{y_i(t, \theta_j)} \quad (8.67)$$

$$FPRSV_{i,j}(t) = FAS_{i,j}(t) \times \theta_j \quad (8.68)$$

$$FPRSQ_{i,j}(t) = \frac{FAS_{i,j}(t)}{y_i(t, \theta_j)} \quad (8.69)$$

Central Sensitivity

Four central sensitivity functions are defined: Central Absolute Sensitivity (CAS), Central Relative Sensitivity (CRS), Central Partial Relative Sensitivity with respect to Variable (CPRSV) and Central Partial

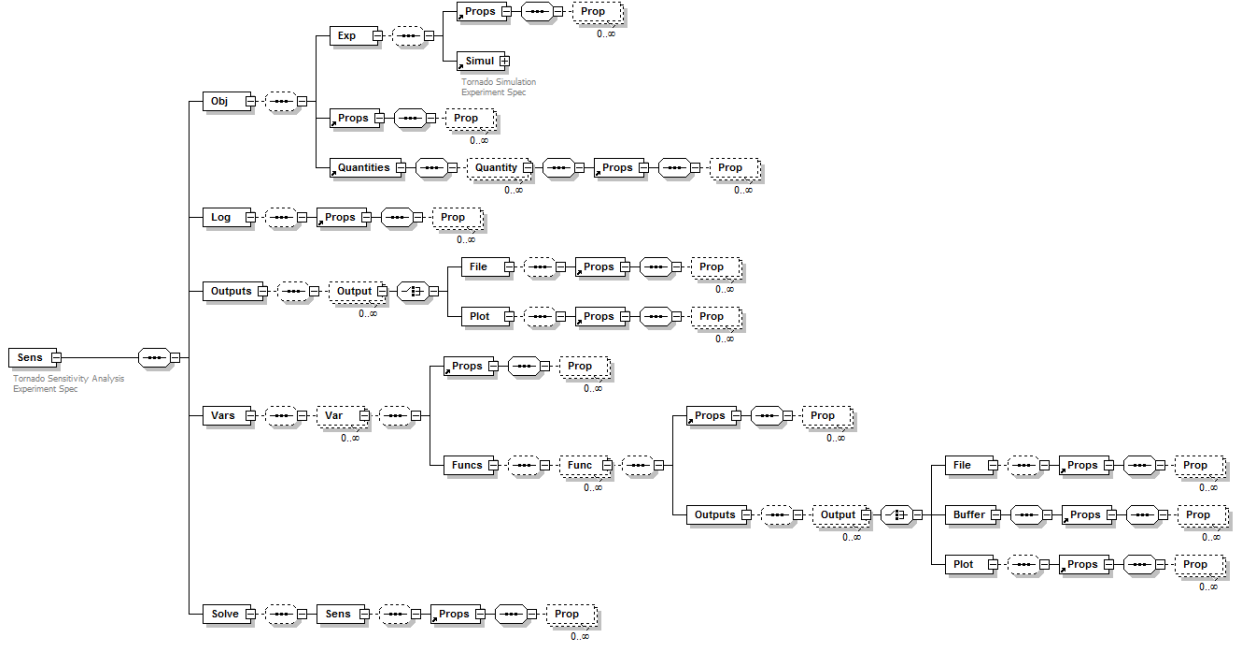


Figure 8.20: ExpSens XML Schema Definition

Relative Sensitivity with respect to Quantity (CPRSQ). These are respectively determined by (8.70)-(8.72), (8.73), (8.74) and (8.75)

$$CAS_{i,j}(t) = \frac{\frac{\partial y_i}{\partial \theta_{j+}} + \frac{\partial y_i}{\partial \theta_{j-}}}{2} \quad (8.70)$$

$$CAS_{i,j}(t) = \frac{FAS_{i,j}(t) + BAS_{i,j}(t)}{2} \quad (8.71)$$

$$CAS_{i,j}(t) = \frac{y_i(t, \theta_j + \xi \theta_j) - y_i(t, \theta_j - \xi \theta_j)}{2 \times \xi \theta_j} \quad (8.72)$$

$$CRS_{i,j}(t) = \frac{CAS_{i,j}(t) \times \theta_j}{y_i(t, \theta_j)} \quad (8.73)$$

$$CPRS_{i,j}(t) = CAS_{i,j}(t) \times \theta_j \quad (8.74)$$

$$CPRS_{i,j}(t) = \frac{CAS_{i,j}(t)}{y_i(t, \theta_j)} \quad (8.75)$$

Sensitivity Quality Information

Sensitivity quality information tries to provide measures for the difference between backwardly and forwardly perturbed trajectories (De Pauw, 2005). If the difference is too large, the sensitivity may not be sufficiently accurate. Three error criteria are defined, which are evaluated at every time point: absolute error (AE), relative error (RE) and squared error (SE). These are respectively defined by (8.76), (8.77) and (8.78).

Table 8.5: ExpSens Symbol Descriptions

Symbol	Description
FAS	Forward Absolute Sensitivity
FRS	Forward Relative Sensitivity
FPRSV	Forward Partial Relative Sensitivity with respect to Variable
FPRSQ	Forward Partial Relative Sensitivity with respect to Quantity
BAS	Backward Absolute Sensitivity
BRS	Backward Relative Sensitivity
BPRSV	Backward Partial Relative Sensitivity with respect to Variable
BPRSQ	Backward Partial Relative Sensitivity with respect to Quantity
CAS	Central Absolute Sensitivity
CRS	Central Relative Sensitivity
CPRSV	Central Partial Relative Sensitivity with respect to Variable
CPRSQ	Central Partial Relative Sensitivity with respect to Quantity
AE	Absolute Error
RE	Relative Error
SE	Squared Error
MAE	Mean Absolute Error
MRE	Mean Relative Error
RMSE	Root Mean Squared Error
MaxAE	Maximum Absolute Error
MaxRE	Maximum Relative Error
MaxSE	Maximum Squared Error
LimitMAE	Upper limit for Mean Absolute Error
LimitMRE	Upper limit for Mean Relative Error
LimitRMSE	Upper limit for Root Mean Squared Error
LimitMaxAE	Upper limit for Maximum Absolute Error
LimitMaxRE	Upper limit for Maximum Relative Error
LimitMaxSE	Upper limit for Maximum Squared Error
θ	Variables
y	Quantities
n	Number of equidistant time points at which sensitivity is evaluated
ξ	Perturbation factor
i	Index of quantity
j	Index of variable
k	Index of time point

$$AE_{i,j,k} = |FAS_{i,j}(t_k) - BAS_{i,j}(t_k)| \quad (8.76)$$

$$RE_{i,j,k} = \left| \frac{FAS_{i,j}(t_k) - BAS_{i,j}(t_k)}{FAS_{i,j}(t_k)} \right| \quad (8.77)$$

$$SE_{i,j,k} = (FAS_{i,j}(t_k) - BAS_{i,j}(t_k))^2 \quad (8.78)$$

The error values are aggregated over all time points in order to come to the sensitivity quality measures. The first type of aggregation is based on the computation of mean error values, which leads to a Mean Absolute Error (MAE), Mean Relative Error (MRE) and Root Mean Squared Error (RMSE). These are respectively defined by (8.79), (8.80) and (8.81).

$$MAE_{i,j} = \frac{1}{n} \sum_{k=0}^n AE_{i,j,k} \quad (8.79)$$

$$MRE_{i,j} = \frac{1}{n} \sum_{k=0}^n RE_{i,j,k} \quad (8.80)$$

$$RMSE_{i,j} = \sqrt{\frac{1}{n} \sum_{k=0}^n SE_{i,j,k}} \quad (8.81)$$

The second type of aggregation is based on the computation of maximum error values, which leads to a Maximum Absolute Error (MaxAE), Maximum Relative Error (MaxRE) and Maximum Squared Error (MaxSE). These are respectively defined by (8.82), (8.83) and (8.84).

$$MaxAE_{i,j} = \max_k AE_{i,j,k} \quad (8.82)$$

$$MaxRE_{i,j} = \max_k RE_{i,j,k} \quad (8.83)$$

$$MaxSE_{i,j} = \max_k SE_{i,j,k} \quad (8.84)$$

Sensitivity Reliability Check

The sensitivity reliability check that is performed by the ExpSens experiment simply verifies whether all sensitivity quality measures are below certain user-supplied threshold values, *cf.* (8.85). These threshold values are set as properties of the Tornado Main entity, and therefore apply to all ExpSens experiment instances.

$$\begin{aligned} & \text{assert}(MAE_{i,j} \leq LimitMAE) \\ & \text{assert}(MRE_{i,j} \leq LimitMRE) \\ & \text{assert}(RMSE_{i,j} \leq LimitRMSE) \\ & \text{assert}(MaxAE_{i,j} \leq LimitMaxAE) \\ & \text{assert}(MaxRE_{i,j} \leq LimitMaxRE) \\ & \text{assert}(MaxSE_{i,j} \leq LimitMaxSE) \end{aligned} \quad (8.85)$$

8.3.7 ExpOptim and ExpCI: Optimization and Confidence Information

The ExpOptim experiment type tries to find values for a selected set of model parameters, input variables and initial conditions that minimizes an objective function, implemented through an ExpObjEval experiment. The optimization process is governed by one of several optimization solver plug-ins that can be dynamically-loaded at run-time. An overview of the optimization solvers that have been implemented as solver plug-ins can be found in Appendix C.

The ExpOptim experiment is a 3-level compound experiment, for it contains an ExpObjEval experiment (*ExpObjEval*), either through embedding or through referral. This ExpObjEval experiment in turn contains an ExpSimul experiment. The ExpOptim experiment uses a callback for messages (*CallbackMessage*), a callback to report on new execution runs (*CallbackRunNo*) and a callback to poll for user interrupts (*CallbackStop*). The ExpOptim process is steered by an objective (*OptimObj*) that invokes an ExpObjEval experiment. Values for a selected set of model parameters, input variables and initial conditions (*OptimVar*) for each ExpObjEval run are provided by an optimization solver (*SolveOptim*).

Information on the evolution of the objective value can be stored in output files (*OptimOutputFile*) or sent to a graphical output channel (*OptimOutputPlot*). If desired, buffers could be added at a later stage. Finally, there is also a log file (*OptimLogFile*) that is used for unformatted textual logging information on the progress of the optimization.

An ER diagram of the internal representation of the ExpOptim experiment type is in Figure 8.21. An XML Schema Definition of the persistent representation of this experiment type can be found in Figure 8.22.

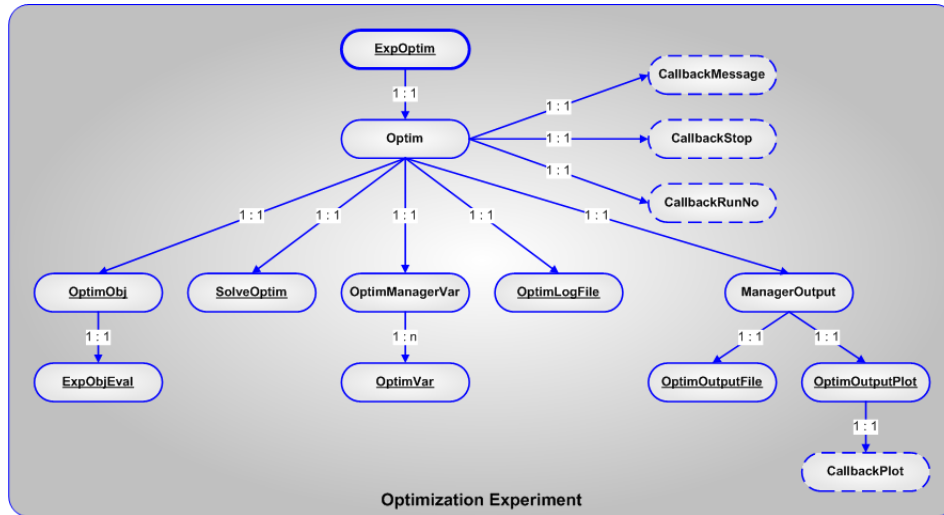


Figure 8.21: ExpOptim ER Diagram

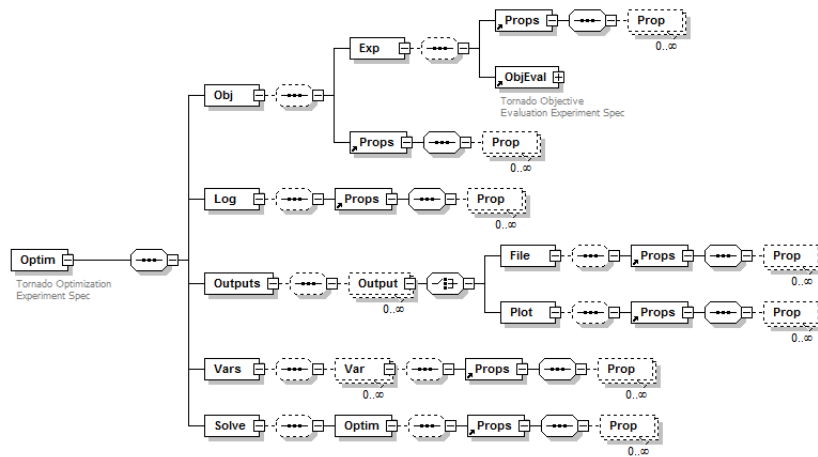


Figure 8.22: ExpOptim XML Schema Definition

Related to the ExpOptim experiment is the Confidence Information (ExpCI) experiment. It can be used to gain knowledge about the accuracy with which parameters (and/or input variables and initial conditions) have been estimated using an ExpOptim experiment. Confidence information is provided through the parameter estimation covariance matrix. This matrix is derived from the inverse of the Hessian matrix, which is calculated by finite differences (implemented through the Richardson solver plug-in) or through a quadratic approximation (implemented by the Nelder & Mead solver plug-in) of the objective surface. The ExpCI experiment type that is implemented in Tornado is due to Dirk De

Pauw and is extensively described in (De Pauw, 2005). The mathematical background will therefore not be given in this dissertation.

The structure of the ExpCI experiment is entirely analogous to the ExpOptim experiment. Actually, in the Tornado-I kernel the optimization and confidence information experiment types were integrated into one experiment. However, given the complexity of the confidence information procedure and the fact that potentially multiple alternative methods can be foreseen to compute an approximation of the Hessian matrix (as is currently the case with the Richardson and Nelder & Mead solvers), it was decided in Tornado-II to create separate experiment types. Evidently, an ExpOptim experiment and its related ExpCI experiment make use of the same ExpObjEval experiment. Given the fact that both ExpOptim and ExpCI experiment can point to an ExpObjEval experiment through referral, it can be easily accomplished to let both ExpOptim and ExpCI point to the same ExpObjEval description.

An ER diagram of the internal representation of the ExpCI experiment type is in Figure 8.23. An XML Schema Definition of the persistent representation of this experiment type can be found in Figure 8.24.

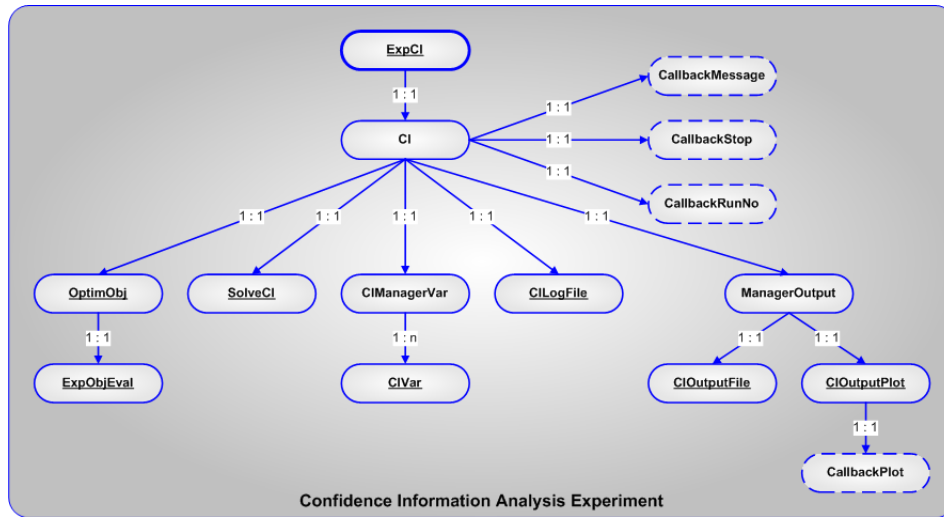


Figure 8.23: ExpCI ER Diagram

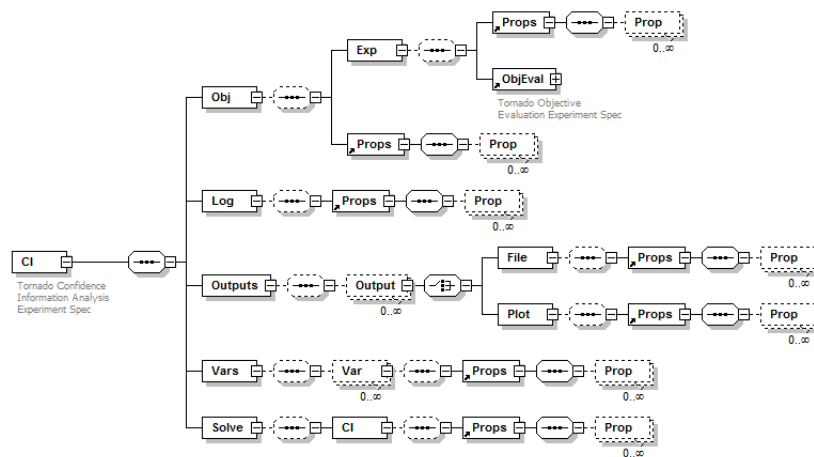


Figure 8.24: ExpCI XML Schema Definition

8.3.8 ExpStats: Statistical Analysis

The ExpStats experiment type performs statistical analysis of a number of time series (*i.e.*, runs). For every time point, defined by a time base that can be specified within the ExpStats experiment, one or more aggregation functions are invoked on values from these time series. Typically, ExpStats is run after an ExpScen or ExpMC experiment, since it allows for performing statistical analysis on the simulated trajectories that were generated as a result of varying parameters, input variables and/or initial conditions through these experiments. In some exotic situations, ExpStats could also be run on simulated trajectories that are a result of ExpOptim and ExpCI experiments.

According to the experiment composition hierarchy depicted in Figure 8.2, ExpStats is a 4-level compound experiment. However, in reality it does not really contain ExpOptim, ExpCI, ExpScen or ExpMC experiments, not through embedding nor through referral. It rather refers to the output files that were generated by these four experiments through a file name descriptor that can be set as a property. The advantage of this approach is that also other series of files (potentially generated by applications that are unrelated to Tornado) can be processed.

As mentioned before, the ExpStats experiment allows for the definition of a time base (*StatsTime*). The reason why it is necessary to be able to define an additional time base at the level of the experiment (in combination with interpolation), instead of using the time points that occur in the input time series, is because the time points for which the input time series provides data are not necessarily equal. In case the time series has been generated with an integration solver with a fixed stepsize, or in case an output communication interval has been specified, time points will be equal. However, in case a variable stepsize integration solver is used and no communication interval is specified, time points will not be equal. Next to a time base, the ExpStats experiment has an objective (*StatsObj*) that governs the process. A set of quantities (*StatsObjQuantity*) determines the state variables, worker variables and/or output variables for which the analysis is to be performed. Output results can be sent to file (*StatsOutputFile*) or to graphical plot channels (*StatsOutputPlot*).

An ER diagram of the internal representation of the ExpStats experiment type is in Figure 8.25. An XML Schema Definition of the persistent representation of this experiment type can be found in Figure 8.26.

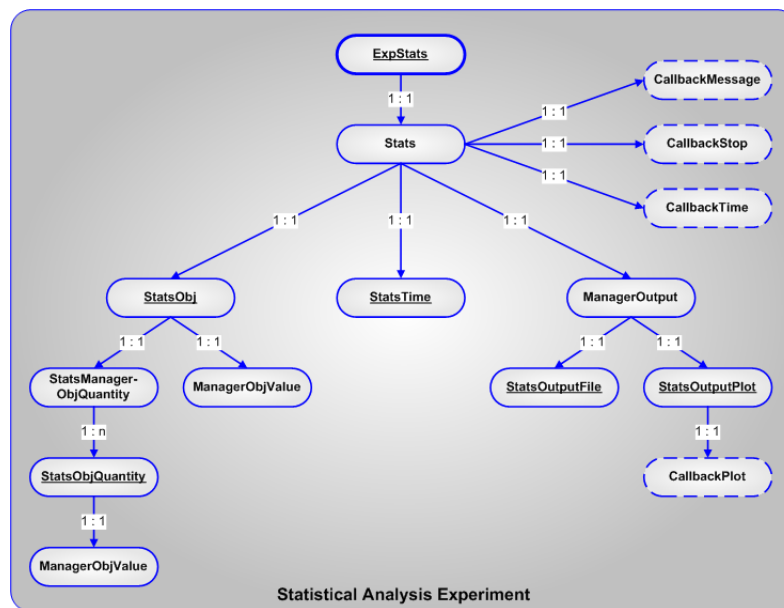


Figure 8.25: ExpStats ER Diagram

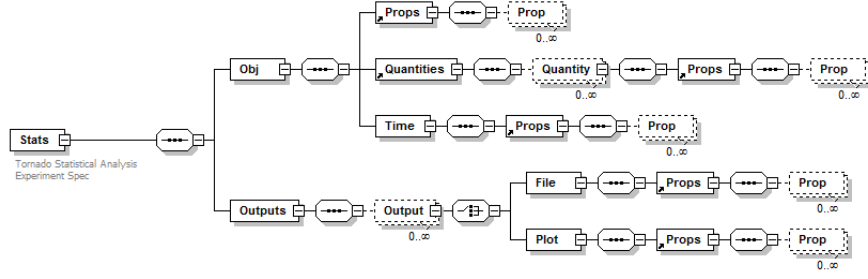


Figure 8.26: ExpStats XML Schema Definition

The objective values that can be computed for each quantity at a certain time point, aggregating all runs, are listed below:

- **Min:** Minimum value
- **Max:** Maximum value
- **Mean:** Mean value
- **StdDev:** Standard deviation
- **Median:** Median value
- **Skewness:** Skewness estimator
- **Kurtosis:** Kurtosis estimator
- **Percentile:** n-th percentile
- **MinRunNo:** Number of the run that yields the minimum value
- **MaxRunNo:** Number of the run that yields the maximum value
- **MeanRunNo:** Number of the run that is closest to the mean value

All of these procedures are implemented through the Common Library. As a result, the implementation of the ExpStats experiment type is very straightforward.

At the moment only a linear time base is available, as in (8.86). t_{start} , t_{stop} and Δ are supplied by the user, therefore $n_t = (t_{stop} - t_{start})/\Delta$.

$$t_k = t_1 + (k - 1) \times \Delta \quad k = 1, \dots, n_t \quad (8.86)$$

8.3.9 ExpEnsemble: Ensemble Simulation

The ExpEnsemble experiment type implements a form of ensemble modelling (Barai and Reich, 1999). The goal of ensemble modelling is to create a better or more reliable model by combining a number of alternative models that model the same system or phenomenon. The ExpEnsemble experiment type in Tornado runs simulations using a number of ExpSimul experiments that are supposed to be based on alternative models for the same system. The simulation results are then aggregated into an ensemble trajectory. The aggregation is done through the computation of a weighted mean. Before the computation of the mean, the trajectories can be modified by applying an offset and/or factor, if desired.

According to the experiment composition hierarchy depicted in Figure 8.2, ExpEnsemble is a 2-level compound experiment since it contains ExpSimul experiments. As a mechanism for containment, only referral is available. Embedding has not been implemented since it would make the ExpEnsemble experiment description too heavy in case a large number of ExpSimul sub-experiments are involved.

The ExpEnsemble experiment allows for the definition of a time base (*EnsembleTime*), for reasons similar to the ExpStats experiment. Simulation progress and status information are communicated to the encapsulating application through callbacks (*CallbackTime* and *CallbackMessage*). A simulation user stop can be incurred through a callback (*CallbackStop*). The ExpEnsemble experiment has a objective entity (*EnsembleObj*) that governs that process and a collection of quantities (*EnsembleObjQuantity*) that determines which ensemble trajectories need to be computed. Output can be sent to a file (*EnsembleOutputFile*) or to graphical output channels (*EnsembleOutputPlot*). Other types of output, such as buffers, are not available at this point but may be added at a later stage.

An ER diagram of the internal representation of the ExpEnsemble experiment type is in Figure 8.27. An XML Schema Definition of the persistent representation of this experiment type can be found in Figure 8.28.

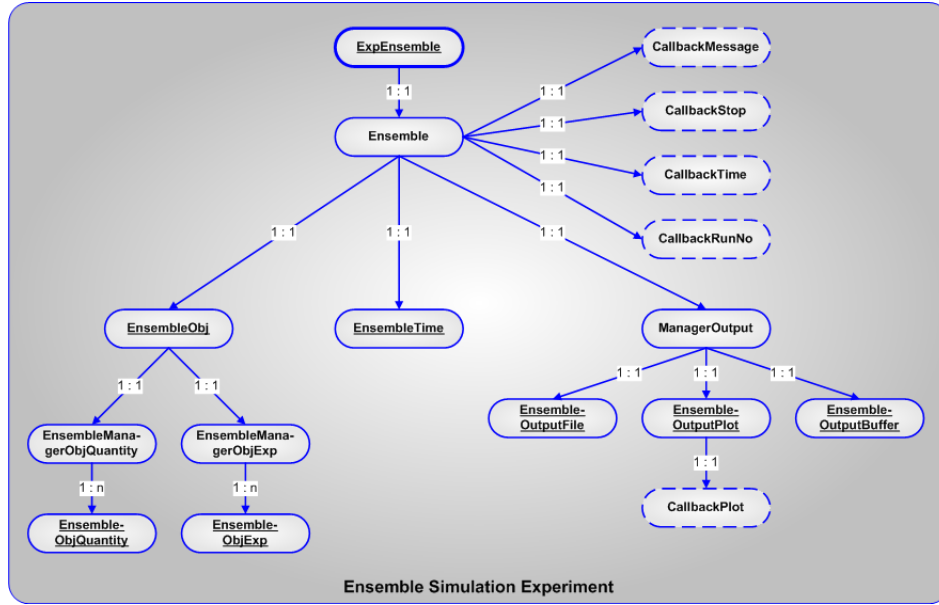


Figure 8.27: ExpEnsemble ER Diagram

As mentioned before, trajectories can be modified on the basis of an offset and a factor before they are used to compute the weighted mean. In fact, next to the mean trajectory, also a minimum and maximum trajectory are computed, as well as two percentile trajectories (*e.g.*, for the 5th and 95th percentiles), to serve as envelopes. This process is respectively represented by (8.87), (8.88), (8.89), (8.90), (8.91) and (8.92). The symbols used in these equations are described in Table 8.6.

$$y'_{i,j}(t_k) = a_j y_i(t_k) + b_j \quad i = 1, \dots, n_q; j = 1, \dots, n_m \quad (8.87)$$

$$y_i^{Mean}(t_k) = \frac{\sum_{j=1}^{n_m} w_j \times y'_{i,j}(t_k)}{\sum_{j=1}^{n_m} w_j} \quad i = 1, \dots, n_q \quad (8.88)$$

$$y_i^{Min}(t_k) = \min_{j=1}^{n_m} y'_{i,j}(t_k) \quad i = 1, \dots, n_q \quad (8.89)$$

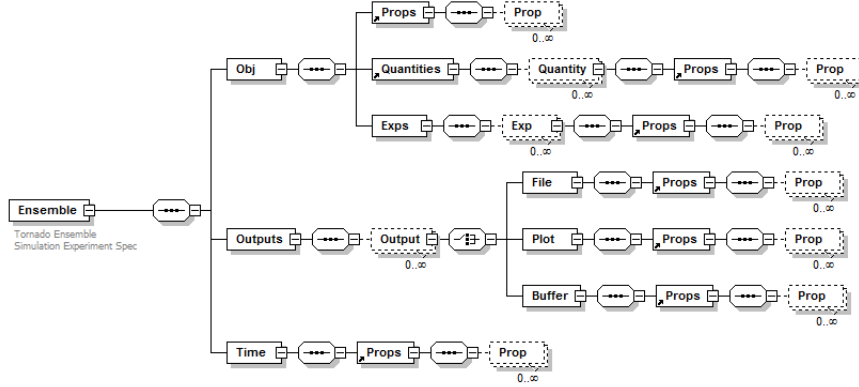


Figure 8.28: ExpEnsemble XML Schema Definition

Table 8.6: ExpEnsemble Symbol Descriptions

Symbol	Description
$y_{i,j}$	Trajectory of quantity i for model j
$y'_{i,j}$	Modified trajectory of quantity i for model j
y_i^{Mean}	Mean trajectory of quantity i
y_i^{Min}	Minimum trajectory of quantity i
y_i^{Max}	Maximum trajectory of quantity i
$y_i^{LowerPercentile(p_l)}$	p_l -th percentile trajectory of quantity i
$y_i^{UpperPercentile(p_u)}$	p_u -th percentile trajectory of quantity i
a_j	Factor for model j
b_j	Offset for model j
w_j	Weight for model j
n_q	Number of quantities
n_m	Number of models
t_k	Time point

$$y_i^{Max}(t_k) = \max_{j=1}^{n_m} y'_{i,j}(t_k) \quad i = 1, \dots, n_q \quad (8.90)$$

$$y_i^{LowerPercentile(p_l)}(t_k) = P_{y,p_l} \quad i = 1, \dots, n_q \quad (8.91)$$

$$y_i^{UpperPercentile(p_u)}(t_k) = P_{y,p_u} \quad i = 1, \dots, n_q \quad (8.92)$$

8.3.10 ExpSeq: Sequential Execution

The ExpSeq experiment type allows for a sequence of experiments to be executed one after the other. ExpSeq allows for each experiment to store or retrieve time series data, and/or to store or retrieve values of state variables in internal buffers. In this way, information can be exchanged between experiments without being forced to make any of this data persistent. Evidently, exchange of data through files remains a possibility.

There are several applications of the ExpSeq experiment. A few are described in the following:

- **Execute ExpStats after ExpScen / ExpMC / ExpOptim / ExpCI:** Allows for statistical analysis of the simulation run data generated by ExpScen / ExpMC / ExpOptim / ExpCI. In this case, transfer of data is done through a series of output files, each containing simulation run data for a number of selected quantities.
- **Execute ExpSimul after ExpSSRoot / ExpSSOptim:** Allows for starting a dynamic simulation from the steady-state of a system. In this case, the state of the model is transferred from the first experiment to the second through an internal buffer.
- **Execute ExpSimul with algebraic model after ExpSimul with dynamic model:** Allows for running postprocessing (implemented as an algebraic model) on the results of a dynamic simulation. In this case, time series data can either be transferred from the first experiment to the next through internal data buffers or through files.
- **Execute ExpSimul after ExpOptim:** Allows for performing a dynamic simulation using the optimal parameter set that was generated by an optimization experiment.

The ExpSeq experiment is a compound experiment since it uses other experiments. However, it is otherwise difficult to classify since it can contain sub-experiments of any type. As a result, the number of levels of processing is undetermined. Sub-experiments can not be embedded in an ExpSeq experiment and are always used through referrals.

Simulation progress and status information are communicated to the encapsulating application through callbacks (*CallbackTime* and *CallbackMessage*). A simulation user stop can be incurred through a callback (*CallbackStop*). In order to provide information on the active simulation run, a run number callback is available (*CallbackRunNo*). The ExpSeq experiment has an objective entity (*SeqObj*) that governs the process and manages a set of ExpSimul sub-experiments (*SeqObjExp*). For each sub-experiment, the quantities that are to be stored in an internal buffer can be given (*SeqObjExpQuantity*).

An ER diagram of the internal representation of the ExpSeq experiment type is in Figure 8.29. An XML Schema Definition of the persistent representation of this experiment type can be found in Figure 8.30.

8.3.11 ExpScenOptim / ExpMCOptim: Optimization Scenario / Monte Carlo Analysis

The ExpScenOptim and ExpMCOptim experiment types are experimental and will not be described here in full. The purpose of these experiments is to run an ExpOptim experiment several times, starting from different initial values.

The problem with some optimization algorithms is that they easily get trapped in a local minimum. Whether an optimization experiment will get trapped in a local minimum is largely determined by the initial values that are used. The assumption therefore is that it might be beneficial to run an optimization several times, starting from different initial values. In the case of ExpScenOptim and ExpMCOptim, the initial value vectors are respectively determined through an ExpScen or ExpMC experiment. It is evident that the ExpScenOptim and ExpMCOptim experiments are brute force experiments that result in a substantial computational load.

8.3.12 ExpScenSSRoot: Steady-state Analysis with Root Finder Scenario Analysis

The ExpScenSSRoot experiment type allows for performing a scenario analysis on a steady-state analysis experiment. Currently only steady-state experiments based on a root finder are supported, but implementing this procedure for optimizer-based steady-state analyses could easily be implemented as well. The purpose of this experiment is to study the effect of different initial values for parameters and/or

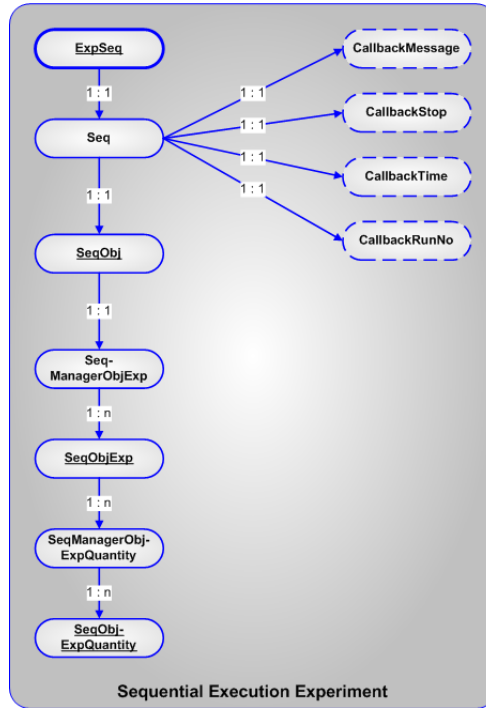


Figure 8.29: ExpSeq ER Diagram

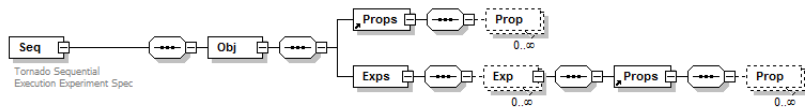


Figure 8.30: ExpSeq XML Schema Definition

derived variables on the steady-state experiment. More specifically, one may be interested in the number of model evaluations that are required to find the steady-state solution, and/or whether the procedure converges.

The ExpScenSSRoot experiment is a 2-level compound experiment, for it contains an ExpSSRoot experiment (*ExpSSRoot*), either through embedding or through referral. The ExpScenSSRoot experiment uses a callback for messages (*CallbackMessage*), a callback to report on new execution runs (*CallbackRunNo*) and a callback to poll for user interrupts (*CallbackStop*). The ExpScenSSRoot process is steered by an objective (*ScenSSRootObj*) that loops through all ExpScenSSRoot variable (*ScenVars*) combinations and runs an ExpSSRoot experiment for each of these. The order in which variable value combinations are used, and the total number of variable value combinations are determined by a pluggable algorithm (*SolveScen*). After the computation of objective values for each ExpSSRoot run, results can be stored in output files (*ScenSSRootOutputFile*) or sent to a graphical output channel (*ScenSSRootOutputPlot*). If desired, buffers could be added at a later stage. Finally, there is also a log file (*ScenLogFile*) that is used for unformatted textual logging information on the progress of the scenario analysis.

An ER diagram of the internal representation of the ExpScenSSRoot experiment type is in Figure 8.31. An XML Schema Definition of the persistent representation of this experiment type can be found in Figure 8.32.

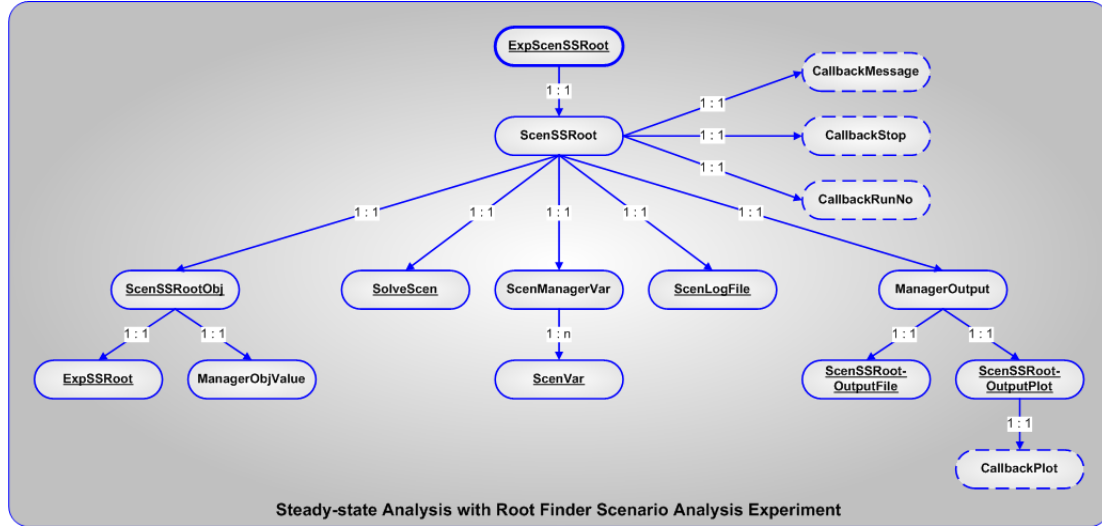


Figure 8.31: ExpScenSSRoot ER Diagram

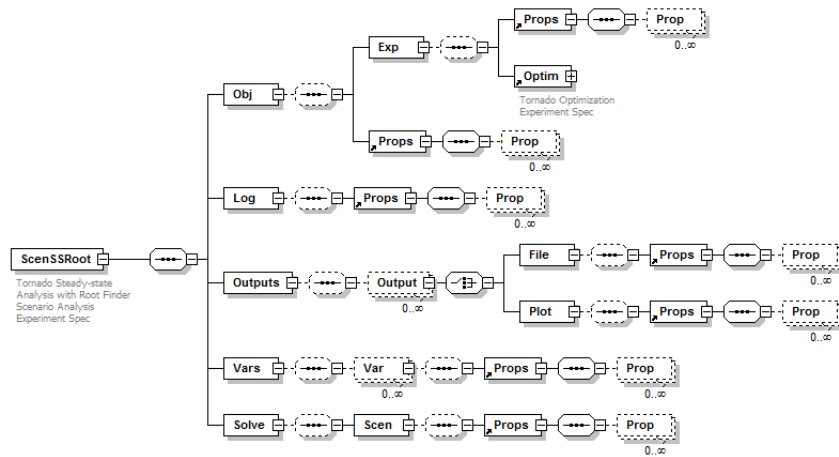


Figure 8.32: ExpScenSSRoot XML Schema Definition

8.4 Generalized Framework for Abstraction and Dynamic Loading of Numerical Solvers

8.4.1 Introduction

In many areas, including modelling and simulation, scientific software typically relies on numerical solvers. Examples of well-known solver types are integrators, optimizers and root finders. Since the advent of digital computing, a *plethora* of solvers have become available for each type of task. The problem however is that in spite of the large number of available solvers for a specific task, it is very rarely the case that one solver can be identified that is truly general, *i.e.*, a solver that is capable of appropriately solving all problems that fall within the boundaries of a certain task. For instance in the case of integration, no solver can be identified that efficiently solves all types of ODE's, DAE's and PDE's. Moreover, even for ODE's it is difficult to find a solver that can handle stiff and non-stiff systems equally well.

A scientific software system with a certain degree of genericity usually allows for performing different tasks (integration, optimization, . . .), which implies that several types of solvers need to be included. In addition, since each solver typically can only efficiently handle one specific class of problems, several instances of each required solver type must be included. Evidently, the latter also implies that at a certain point, choices must be made about which solver to use for a particular problem. In principle, this choice can either be made by the user on the basis of expert knowledge and/or prior experience, or automatically, *i.e.*, through machine-learning techniques based on the analysis of previously collected data (Claeys et al., 2006g, 2007b). The problem of choosing amongst solver alternatives is not the focus of this section however. Instead, it touches on issues that can be encountered when integrating multiple solvers into complex industrial quality software systems, and the ways these issues can be overcome.

8.4.2 Solver Availability

Solver codes for most numerical computation tasks are readily available from literature, lecture notes and on-line repositories (either public or commercial). A well-known book on numerical techniques is (Press et al., 1992). Table 8.7 lists some of the most popular on-line resources.

Table 8.7: Popular Numerical Solver Repositories

Repository	Description	URL
Netlib	Network Library	http://www.netlib.org
GAMS	Guide to Available Mathematical Software	http://gams.nist.gov
ACM TOMS	Transactions on Mathematical Software	http://www.acm.org/toms
NAG	Numerical Algorithms Group	http://www.nag.co.uk
SUNDIALS	Site of Nonlinear and Differential/Algebraic Eq. Solvers	http://www.llnl.gov/CASC/sundials
Hairer	Website of Ernst Hairer at the University of Geneva	http://www.unige.ch/~hairer/software.html

8.4.3 Embedding Solvers into Quality Software: Integration Issues

When integrating numerical solvers into quality software several issues can be encountered, which render this process often much less straightforward than integrating the same solvers into less demanding software such as prototype or research applications. Quality software (either commercial or non-commercial) is typically subject to a number of requirements that are related to stability, extensibility, maintainability and consistency. For instance, quality software will not allow for application crashes due to solver failures, nor will it allow for memory to be depleted due to iterative calls to a solver routine that does not perform proper memory management. One might argue that in modern software frameworks such as J2EE and .NET, issues such as these have become irrelevant. However, numerical computational efficiency is a major factor in scientific software kernels, and hence modern frameworks that rely on code interpretation, byte code compilation and/or garbage collection should preferably not be used. Actually, it would be incorrect to state that byte code compilation or garbage collection as such are hindrances for numerical efficiency. Instead, there are other features that typically come with languages that offer byte code compilation and garbage collection that have a more detrimental effect on performance, *e.g.*, non-rectangular arrays, exact 64-bit floating-point computations and detailed exception-handling semantics. There is however ongoing work, *e.g.*, on high-performance Java, to overcome some of these limitations. In the mean time, one will notice that compiled languages such as FORTRAN, C and C++ remain popular for the development of scientific software kernels. As a result, the problem of integrating numerical solutions in quality software is largely restricted to the world of C/C++ and FORTRAN.

The following gives a non-exhaustive overview of issues that can be encountered when integrating numerical solvers into quality software:

- **Programming language heterogeneity:** A situation that commonly occurs is that the solver code that needs to be integrated into the application is programmed in a language that differs from the language of the encapsulating application. In practice this most often proves to be a minor problem. In case the languages are C and FORTRAN, there is compatibility at link-level by default, meaning that compiled objects written in C can call objects written in FORTRAN and *vice versa*. However, one must keep in mind that arrays are stored *row-major*¹ in C and *column-major* in FORTRAN, some data transformations may therefore be required. Also, C++ has no trouble calling C and FORTRAN. However, in the opposite case compatibility is only ensured when the prototypes of the C++ functions to be called are preceded by the *export* “C” clause.
- **Lack of extensibility and maintainability:** The most direct way of integrating solver codes is through static linkage. Evidently, this implies that each time a new version of the solver code is to be integrated, re-linkage is required. Moreover, in case additional solver codes are to be added, modifications of the application code, re-compilation and re-linkage are required. Clearly, in view of extensibility and maintainability, a mechanism that allows for solvers to be loaded dynamically is required.
- **Function signature heterogeneity:** Inevitably, solvers for different tasks (integration, optimization, *etc.*) will have different signatures. However, since solver codes for the same task may have very different origins, their function signatures are in practice also very different. In order to allow for the development of application code that is independent of the particular solver that is being used, an intermediate abstraction layer is required.
- **Solver setting heterogeneity:** Each solver typically has a number of configuration options or settings. In the case of integration solvers, these settings for instance include stepsizes and tolerances. Solvers that perform similar tasks may not only have a different number of settings, but often settings that have the same meaning are named differently, or settings with the same name have a somewhat different meaning. The confusion that arises from this should be overcome through the introduction of a flexible solver setting querying and updating mechanism.
- **Lack of I/O flexibility:** It is very common for solvers to generate messages during processing. These messages can be classified as error messages, warnings and informational messages. Typically these messages are written directly to the console (*i.e.*, *standard output*). Sometimes a distinction is made between *standard output* for informational messages and warnings and *standard error* for errors. Clearly, this behavior may be very undesirable when a solver is integrated in an application, since in this case output may have to be sent to a log file or a widget that is part of a GUI.
- **Lack of thread-safety:** An issue that is nearly always overlooked by most authors of solver codes is thread-safety. If one wants to keep all deployment options of a solver code open, one should not make any assumptions as to the use of the solver in a single-threaded or multi-threaded context. This specifically means that global variables (*i.e.*, COMMON blocks in FORTRAN) should not be used. Solvers that rely on global data will hamper deployment in types of applications such as Multiple Document Interface (MDI) GUI's and multi-threaded computation servers.
- **Inappropriate memory management:** Numerical solvers usually refrain from the extensive use of dynamically allocated memory in order to maximize efficiency. However, in case dynamically allocated memory is used, it should be properly managed and cleaned up upon exit of the solver

¹http://en.wikipedia.org/wiki/Row-major_order

routines. In practice, solvers often do not (or only partly) clean up dynamically allocated memory in case of solver failures. In applications where a large number of calls to such a solver occur, depletion of memory is to be expected.

8.4.4 The Tornado Solver Framework

Since Tornado has to deal with various solver types (in order to support the respective types of virtual experiments discussed above), and for each type of solver, multiple solver implementations need to be available (in order to cover a number of areas of application that is sufficiently large), a generalized solver framework was developed. The framework is based on dynamic loading and run-time querying of solver plug-ins (*i.e.*, DLL's on Windows and shared objects on Linux). Solver plug-ins are loaded and registered in an internal data structure during startup of the kernel. Afterwards, registered solvers can be used by one or more virtual experiments, possibly concurrently. Solver codes contained in plug-ins are wrapped by an abstraction layer and are equipped with a data structure that allows for run-time querying and updating of solver settings. Figure 8.33 gives an overview of all the solver plug-ins that are currently available for Tornado.

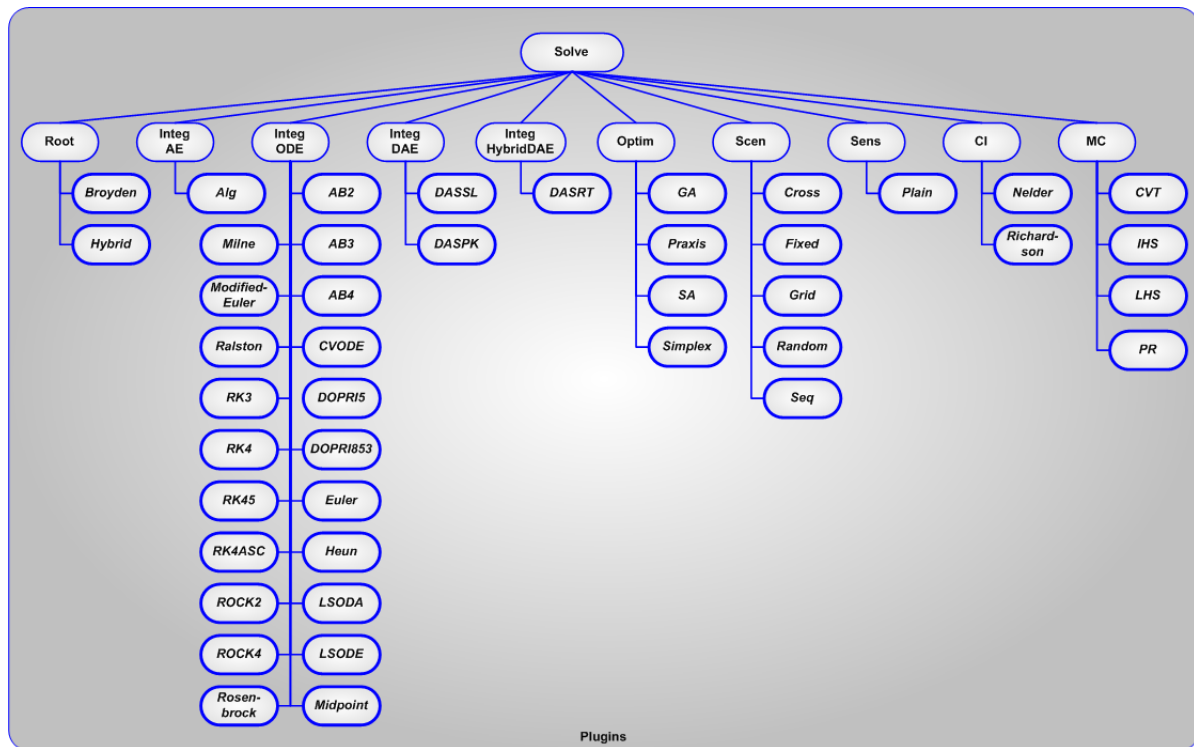


Figure 8.33: Solver Plug-ins

Inheritance hierarchy

Tornado was designed in an object-oriented manner, which implies that the software is based on classes that are structured according to an inheritance hierarchy. Figure 8.34 depicts an excerpt from the solver base class inheritance hierarchy in Tornado. The figure shows that all classes are derived from an abstract interface. From the most general solver base class (which only contains two methods: *Reset()* and *Solve()*), other base classes have been derived that are related to each type of task for which a solver is

required. As the figure is only an excerpt from the complete hierarchy, only base classes for root finding, integration and optimization are shown. In reality, Tornado supports a wider variety of solver types.

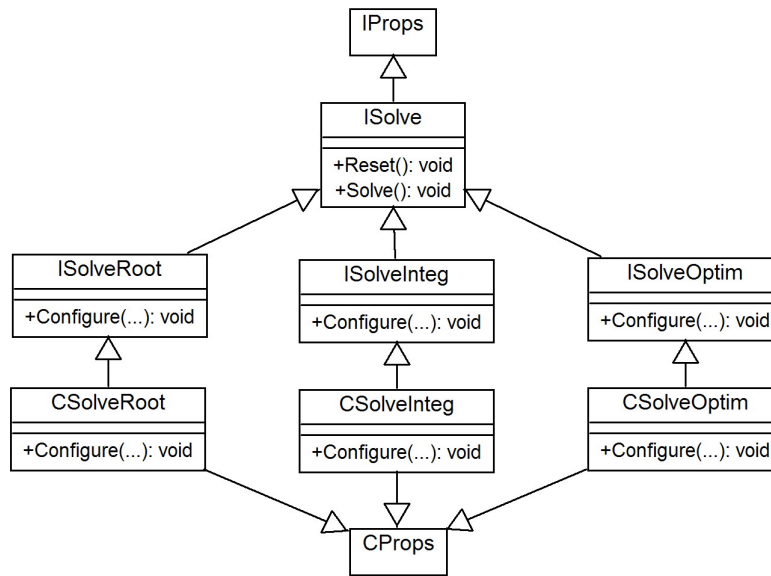


Figure 8.34: Solver Inheritance Hierarchy Excerpt

The figure also shows that *SolveRoot*, *SolveInteg* and *SolveOptim* all contain a method named *Configure()* (next to some other less important methods that are not shown). This method configures the solver with references to a number of entities that are required for the operation of the solver. As an example, Table 8.8 lists the arguments of this method for the integration solver base class.

Table 8.8: Arguments of the *SolveInteg::Configure()* Method

Argument	Description
CallbackMessage	Called when messages are generated
CallbackStop	Called to discover whether processing is to be stopped
CallbackLicense	Called to discover whether the solver is licensed
CallbackTime	Called when the integration time is incremented
Model	Reference to the model to be integrated
Input	Reference to the model's input providers
Output	Reference to the model's output acceptors

Figure 8.34 also shows that abstract interfaces and base classes are derived from a class named *Props*. This class implements the properties mechanism that was discussed in Chapter 7. All solver codes that are to be integrated into Tornado should be part of the Tornado solver hierarchy (*i.e.*, derived from one of the solver base classes). In most cases a thin layer of wrapper code as well as some modifications of the solver code itself are required to this end.

Dynamic Loading

Solver implementations derived from the base classes of the Tornado solver hierarchy are to be wrapped as a dynamically-loadable object (DLL or shared object) before they can be loaded into Tornado. In order to allow for this mechanism to be ported to as many platforms as possible, simplicity has been favored

to the largest extent. In fact, the only two functions that are expected to be exported by the dynamically loadable object are the following:

```
wchar_t* GetID();  
void* Create();
```

The first function should return a string that uniquely identifies the solver that is contained within the dynamically-loadable object. This string should be structured as follows: *Tornado.Solve.<SolverType>.-<SolverName>*. For example, in case of a Runge-Kutta 4 solver an identifier such as *Tornado.Solve.-Integ.RK4* could be used.

The second function is expected to act as a factory (Gamma et al., 2003) for solver instances, *i.e.*, when called it should return a new instance of the solver class. For reasons of simplicity, the most general pointer type (*void**) is used as a return data type. Evidently, after calling the *Create()* method Tornado will have to apply the appropriate casting.

As mentioned before, Tornado will load solver plug-ins at startup. The list of plug-ins that is to be loaded is specified in the main configuration file for Tornado, which is an XML document with a predefined grammar. The following sample configuration file shows that 7 solvers are to be loaded (3 integration solvers, 2 optimization solver and 2 root solvers). The names mentioned are plug-in file names, which could be absolute or relative path names (possibly containing environment variables). At the moment a total of 43 solver plug-ins are available for Tornado (*cf.* Figure 8.33).

```
<Tornado>  
  <Main Version="1.0">  
    <Props/>  
    <Plugins>  
      <Plugin Name="SolveIntegCVODE"/>  
      <Plugin Name="SolveIntegDASSL"/>  
      <Plugin Name="SolveIntegRK4"/>  
      <Plugin Name="SolveOptimPraxis"/>  
      <Plugin Name="SolveOptimSimplex"/>  
      <Plugin Name="SolveRootBroyden"/>  
      <Plugin Name="SolveRootHybrid"/>  
    </Plugins>  
    <Units/>  
  </Main>  
</Tornado>
```

Properties

In Tornado, solver settings (such as tolerances or stepsizes for integration solvers) are not manipulated through specialized method calls, since every solver has its own distinct settings. A more general system is therefore required to be able to manipulate solvers in a plug-and-play fashion. The solution that is implemented by Tornado is based on a map (or dictionary) of so-called *properties* that is to be provided by each solver.

Table 8.9 contains an overview of the properties that are supported by the CVODE integration solver plug-in for Tornado, which originates from the SUNDIALS suite². The first three properties in this table are read-only. These properties provide some information on the characteristics of the solver, which is a first step towards automated solver selection.

²<http://www.llnl.gov/CASC/sundials>

Table 8.9: Properties of the CVODE Integration Solver

Name	Type	Access	Default	Range
ModelTypes	String	R	ODE	-
StepSizeType	String	R	Variable	-
IsStiffSolver	Boolean	R	true	-
MaxNoSteps	Integer	R/W	0	0 - ∞
AbsoluteTolerance	Real	R/W	1e-7	0 - ∞
RelativeTolerance	Real	R/W	1e-7	0 - ∞
LinearMultistepMethod	String	R/W	Adams	Adams;BDF
IterationMethod	String	R/W	Functional	Functional;Newton
LinearSolver	String	R/W	Dense	Dense;Band;Diag;SPGMR
CVBandUpperBandwidth	Integer	R/W	0	0 - ∞
CVBandLowerBandwidth	Integer	R/W	0	0 - ∞
CVSPGMRGSType	String	R/W	ModifiedGS	ModifiedGS;ClassicalGS

Callbacks

In Tornado, solvers interact with their encapsulating application through callbacks. In an object-oriented context such as the Tornado framework, callbacks are implemented through abstract interfaces. These interfaces are to be implemented by the encapsulated application and called by the solver when certain events occur. The idea behind this is that the ultimate decision on how to handle a certain event should not be with the solver, but with the application. The most notable situation in which callbacks are required occurs when the solvers generate a message (info, warning or error). The solver should not directly output this message, *e.g.*, to the console), but rather pass on the message to the application. The application will then take the appropriate action, *e.g.*, display the message in a text widget or pop up an error dialog box.

As is shown in Table 8.8, integration solvers can make use of 4 callbacks in Tornado. Next to the message callback, there is a callback for testing whether the user has requested for processing to be aborted. In addition, there are callbacks for testing if the solver plug-in is appropriately licensed and for notifying the application about the incrementation of the integration time. Other types of solvers (optimizers, root finders, ...) will support a different set of callbacks. A subset of callbacks, consisting of a message and stop callback, will however always be present.

Thread-safety

The Tornado kernel has been conceived as a multi-threaded software system. Most notable example is the fact that Tornado allows for running multiple virtual experiments concurrently. A necessary condition for this is that every virtual experiment has its own local data. Global experiment-related data is therefore strictly forbidden. Consequently, solvers should also refrain from the use of global data (*e.g.*, COMMON blocks in FORTRAN and global variables in C/C++).

A situation that frequently occurs is that solver codes implemented in FORTRAN have to be integrated in Tornado. As mentioned before, FORTRAN-compiled objects can be linked to objects with C-linkage. However, in the case of Tornado another approach has been followed, based on the automatic conversion of the original FORTRAN code to C through the application of the well-known *f2c*³ tool. A fortunate result of the use of *f2c* is that FORTRAN COMMON blocks are not translated to global C variables, but to structs. It is possible to create separate instances of these structs for every experiment instance, hence guaranteeing that experiments will use local data only.

³<http://www.netlib.org/f2c>

Unfortunately, not all solver codes that rely on global data are implemented in FORTRAN. Situations also occur where solver codes implemented in C or C++ that rely on global variables have to be integrated in Tornado. In this case, two options exist. One can either modify the code manually in order to replace the global variables by data that can be instantiated. Depending on the number of global variables, the amount of manual work involved may or may not be acceptable. In case it is not acceptable, another alternative is currently under development in Tornado. It is based on loading a separate copy of the plug-in that contains the solver and its global data, for every experiment that uses it. Since the address spaces of dynamically loaded objects are distinct, the use of only local data is hence again guaranteed. However, there is of course also a downside to this approach. Loading multiple copies of the same dynamically loadable object is much less efficient than creating multiple instances of a solver contained in a plug-in that is only loaded once.

Adding Custom Solvers

Tornado users are free to implement additional solver plug-ins. Solver codes have to be developed in C++ by deriving a class from the appropriate solver base class. In order to be able to build solver binaries, a number of header files and one static library (*TornadoSolve.lib*) are needed. Tornado can be informed about the availability of new solver plug-ins by modifying the Tornado main XML configuration file.

8.5 Managing Graphical Input Provider and Output Acceptor Representations

As Tornado is a computational software kernel, it does not deal with *rendering* of graphical widgets (this is left up to the application that sits on top the Tornado kernel). However, the management of the *specifications* of these widgets has been included. The reason for this is the fact that Tornado has a convenient framework for handling internal and persistent representation of entities such as Experiments and Layouts. Extending this framework towards support for specifications of graphical input providers and output acceptors requires minor effort, while implementing the same functionality time and again in various applications that are built on top of the kernel, is inefficient. One problem with including this type of support into Tornado is that different graphical input providers and output acceptors not only may require application-dependent renderings, but also may have different sets of properties. To tackle this problem, it has been attempted to come to a set of properties that can act as a superset for most situations. In case additional information is to be represented that does not fit in this scheme, an additional property has been foreseen for each sub-entity that can act as an unformatted receptacle for this information.

An ER diagram for Controls, *i.e.*, graphical input providers, can be found in Figure 8.35. In close correspondence with the ER diagram is the XML Schema Definition that is presented in Figure 8.36. An ER diagram for Plots, *i.e.*, graphical output providers, can be found in Figure 8.37. In close correspondence with the ER diagram is the XML Schema Definition that is presented in Figure 8.38.

8.6 Internationalization and Localization

In computing, *internationalization* is a means of adapting computer software for non-native environments, especially other nations and cultures. Internationalization is the process of ensuring that an application is capable of adapting to local requirements, for instance ensuring that the local writing system can be displayed. On the other hand, *localization* is the process of adapting the software to be as familiar as possible to a specific locale, by displaying text in the local language and using local conventions for the display of things such as units of measurement. Due to their length, the terms are sometimes abbreviated to *L10n* and *i18n* respectively. The distinction between internationalization and localization is

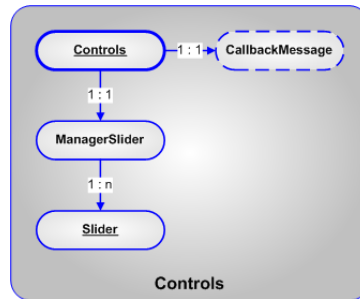


Figure 8.35: Controls ER Diagram

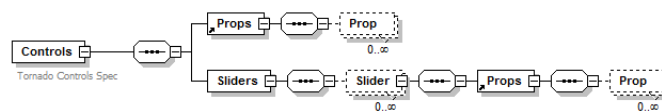


Figure 8.36: Controls XML Schema Definition

subtle but important. Internationalization is the adaptation of products for potential use virtually everywhere, while localization is the addition of special features for use in a specific locale. The processes are complementary, and must be combined to lead to the objective of a system that works globally.

Internationalization and localization have many facets. The following lists a number of aspects of internationalization and localization that are of importance in the scope of Tornado:

- Formatting of numbers (decimal separators, thousand separators, number of decimal digits)
- Use of different units and unit systems (metric units versus imperial units)
- Use of different alphabets
- Translation of text strings

Tornado allows for decimal separators, thousand separators and the number of decimal digits to be configured per output acceptor instance. *In concreto*, this means that for every individual output file, buffer, *etc.* it can be decided whether the decimal separator is to be set to a dot (default) or comma, whether the thousand separator is to be set to a comma (default) or dot, and how many digits should be used for the representation of reals (8 by default). It is important to note that these settings only apply to data items that are made persistent. Internally, Tornado uses a native format to represent numbers and always uses the maximum precision.

For input providers, the situation is similar. Every input file, buffer, *etc.* that is presented to Tornado can use its own settings with respect to decimal and thousand separators. Before using data from input providers, Tornado will apply a conversion to its native internal format. In general it can be stated that Tornado will only perform data conversion and formatting before and after (*i.e.*, never during) internal computational processing, in order not to impede performance.

As will be discussed in Chapter 9, when defining models for use with Tornado, various meta-information items can be specified for each declared quantity. One of these meta-information items specifies the unit of a quantity. Unit information is mostly used for documentation purposes, but can also be adopted to perform unit conversions. In fact, for each virtual experiment a unit system can be set. When a unit system is specified for an experiment, all data items will upon output be converted

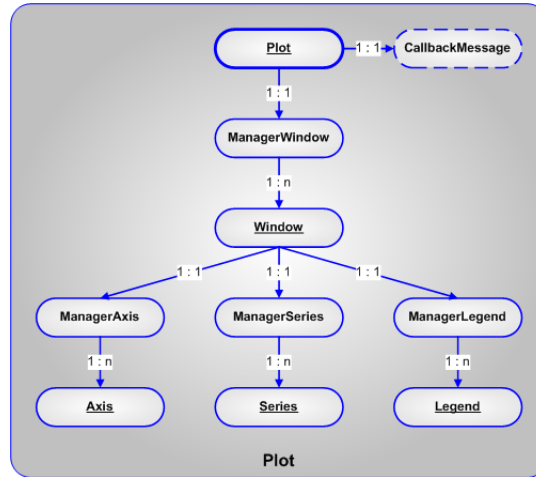


Figure 8.37: Plot ER Diagram

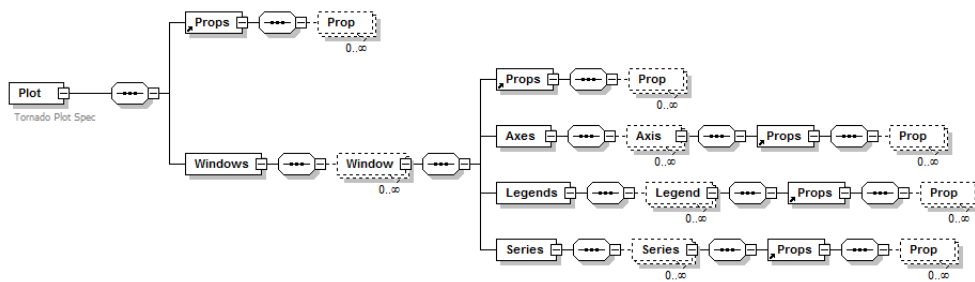


Figure 8.38: Plot XML Schema Definition

to this unit system, except for quantities for which a display unit has been set. Display units can also be specified as meta-information items when defining a model for use with Tornado. When a display unit is set, conversion to this unit will take place upon output, regardless of the unit system specification (*i.e.*, display units overrule unit system specifications). Algorithm 12 further clarifies the process that is followed to determine the conversion that is to be performed. In this algorithm, $GetConversion(x, y)$ returns the conversion that is required to convert x to y ; $GetMetric(x)$ returns the corresponding unit of x in the metric system; $GetImperial(x)$ returns the corresponding unit of x in the imperial system.

Conversion between units can mostly be done on the basis of a factor (*e.g.*, $in \rightarrow cm : y = 2.54x$), however in some cases also an offset is required (*e.g.*, $^{\circ}F \rightarrow ^{\circ}C : y = \frac{5}{9}(x - 32)$). Tornado allows for both types of conversions.

Unit conversion can cause a substantial performance degradation, since it is performed at every output time point during the course of a virtual experiment. Tornado therefore contains provisions to turn off unit conversion if performance needs to be maximized. Note that turning off unit conversion cannot simply be implemented by setting all factors to 1 and offsets to 0. It has been noted that multiplications by 1 that are executed thousands of times cause a performance degradation that is better avoided.

Algorithm 12 Unit Conversion Selection

```

if Unit = "" then
    Conversion := None
else
    if DisplayUnit = "" then
        if UnitSystem = "" then
            Conversion := None
        end if
        if UnitSystem = "Metric" then
            Conversion := GetConversion(Unit, GetMetric(Unit))
        end if
        if UnitSystem = "Imperial" then
            Conversion := GetConversion(Unit, GetImperial(Unit))
        end if
    else
        Conversion := GetConversion(Unit, DisplayUnit)
    end if
end if

```

8.7 Improving Simulation Speed through a priori Exploration of the Solver Setting Space

Finding appropriate integrator solver settings can be a tedious task (especially since environmental scientists and practitioners who use simulation are not necessarily experts in numerical techniques). It is typically a manual trial-and-error process in which solver settings are modified iteratively and simulations are run to investigate the effect. The process continues until settings are found that solve the set of equations at hand with a sufficient level of accuracy, and in a timely fashion. Actually, this process is very similar to the scenario analysis process described in the prequel. It was therefore suggested to extend the standard scenario analysis functionality in Tornado so that not only model parameters, input variables and initial conditions can be varied, but also solver settings.

8.7.1 Description of the Procedure

Thanks to the object-oriented and layered design of Tornado, implementation of solver setting scenario analysis was straightforward. In fact, the scenario analysis experiment only interacts with its embedded simulation experiment through a high-level abstract interface. The most relevant methods in this interface (with respect to scenario analysis) are the following:

```

void SetValue(const std::wstring& FullName,
              double Value);

double GetValue(const std::wstring& FullName);

void Run();

```

In standard scenario analysis, the *FullName* argument in the *Set/GetValue* methods is a fully-qualified model object name. In order to allow for solver setting scenario analysis, the implementation of these methods was modified so that in case the *FullName* argument is structured differently, if it is used to refer to a solver setting rather than a model object. More specifically, when a pattern such as *Solve.<Type>.-<PropName>* instead of *[.<ModelName>]*.<ObjName>* is discovered as a *FullName*, the property of name *<PropName>* in the currently configured solver of type *<Type>* is modified. In the case of

integration, the solver type is *Integ*. Other types of solvers are possible as well (such as optimizers, root finders, ...), but are not relevant in the scope of the current discussion.

The different methods that can be used for varying variables in scenario analysis (manual placement, sampling from distributions and linear/logarithmic spacing) turn out to be very practical in case solver settings need to be varied (*e.g.*, the use of logarithmic spacing for accuracies that are represented by negative powers of 10). Also, the different methods for sorting value combinations (sequential, random, grid, fixed, cross) are equally useful in solver setting scenario analysis as in standard scenario analysis. In case one only wishes to evaluate a limited set of predetermined combinations of settings, the fixed method is appropriate. In case one wants to evaluate an entire grid of combinations, without any preferences with regard to sequence, the sequential method seems most appropriate. The random method (or possibly grid method) could be used to walk through the solver setting space in order to get some initial understanding of how it behaves. Processing can be stopped as soon as this insight has been gained.

One must take into account that in case appropriate solver settings are sought for a certain simulation experiment through scenario analysis, these settings may only apply within a certain neighborhood of the parameters and initial conditions of the model. Strongly differing values may require a new analysis to be performed. If solver settings are sought that are to be applied within the scope of a compound experiment, one must make sure that the parameters and initial conditions of the simulation experiment that is iteratively run within the solver setting scenario analysis, are sufficiently representative for the simulations that are to be run in the standard compound experiment afterwards.

In solver setting scenario analysis, objective functions that are typically applied to standard scenario analysis are not very useful. More interesting in this case is to know the speed and accuracy of simulation. Speed of simulation could in principle be measured through the total simulation time. However, load variations on the machine on which the simulation is run render this metric unreliable in practice. A better approach is to relate simulation speed to the computational complexity of the simulation. A good measure for this is the number of state evaluations of the model. Indeed, the fewer times the state of a model needs to be computed during a simulation, the faster the simulation will run. Since Tornado by default internally computes the number of state evaluations, it was straightforward to make this metric available as a special objective for scenario analysis. It should be noted that next to the model, the solver itself also introduces some computational complexity. In the case of small models, this solver-incurred complexity is non-negligible. However, for the elaborate models that are typically dealt with in the scope of Tornado, the complexity of the model largely outweighs the solver complexity.

Table 8.10 gives an overview of all special objectives that are available for scenario analysis. The *Error* objective simply signals whether there has been an error during the execution of the simulation experiment. The other objectives give the number of evaluations of a specific section within the executable model. The *Initial* and *Final* sections respectively contain initial and final equations (*cf.* Chapter 2). In principle, these sections are executed exactly once, unless there is a bug in the software. The *State* and *Output* sections respectively contain equations that do and do not influence the state of the system. The number of evaluations depends on the nature of the model, the integration solver settings and the simulation time horizon. The *Initial*, *Final*, *State* and *Output* sections are discussed further in Chapter 9.

Table 8.10: Special Objectives that are available for Scenario Analysis

Item	Description
Error	1 in case an error has occurred, 0 otherwise
NoComputeInitials	Number of evaluations of the <i>ComputeInitial</i> routine
NoComputeStates	Number of evaluations of the <i>ComputeState</i> routine
NoComputeOutputs	Number of evaluations of the <i>ComputeOutput</i> routine
NoComputeFinals	Number of evaluations of the <i>ComputeFinal</i> routine

For measuring the simulation accuracy, Tornado also has some useful functionality available: it allows for computing measures of similarity between a simulated trajectory and a reference trajectory (either by computing the mean difference (MeanDiff), or the maximum difference (MaxDiff)). This functionality has also been made available as an objective (as was described in the scope of the ExpObj-Eval experiment), so that trajectories that are simulated through solver settings scenario analysis can be compared with a reference trajectory with the desired accuracy, established beforehand.

Figure 8.39 shows the overall flow of the suggested procedure. First, a solver setting scenario analysis is performed for a simulation experiment with parameters and initial conditions that are representative for the compound experiment that follows. From the results obtained (number of state evaluations, and difference with regard to reference trajectories), an appropriate solver configuration is selected, which is then applied to the simulation experiment that is to be run in the scope of a standard compound experiment. Next follows the compound experiment (typically a standard scenario analysis or Monte Carlo analysis) that will iteratively run this simulation experiment, and should benefit from the improved solver settings that were found through the first step of the procedure.

During both steps of the procedure, simulation sub-experiments can either be run sequentially or concurrently, in case a distributed execution environment is available (to be discussed in Chapter 14).

In order to further clarify both steps of the procedure, a functional comparison of standard and solver setting scenario analysis is presented in Table 8.11.

Table 8.11: Comparison between Standard and Solver Setting Scenario Analysis

Item	Standard	Solver Setting
Variables	Model parameters, input variables and initial conditions	Solver settings
Objective	Aggregation functions MeanDiff, MaxDiff	NoComputeStates MeanDiff, MaxDiff
Solver	Sequential, Random, Grid, Fixed	Sequential, Random, Grid, Fixed
Distributed Execution	Available	Available

8.7.2 Application to the Lux Case

Recently, an integrated model was built for the Sûre river in Luxembourg, and two of its tributaries (Solvi et al., 2006) (see also Chapter 2). The overall model consists of some 12,000 (mainly coupled) parameters, 4,400 algebraic variables and 414 ODE's. For this model, a Monte Carlo analysis was set up that takes 1,000 shots from a 32-dimensional parameter space. Of the 32 parameters involved, 30 vary according to a uniform distribution; the remaining 2 are triangularly distributed.

For simulating the model, an integration solver had to be chosen amongst the wide variety of solvers available in Tornado (approx. 20 integration solvers available at this moment). In a first instance, a conservative approach was followed which lead to the selection of a variable stepsize Runge-Kutta 4 solver. The solver was shown to work well for all 1,000 runs of the Monte Carlo analysis. With this solver, a simulation run takes on average 53 s on a reference machine (HP DL145 Dual Opteron 242 1.6GHz, 4GB RAM, 40GB HD) and results in an average total of 392,356 state evaluations per run.

In general, advanced solvers such as CVODE, which is part of the SUNDIALS⁴ suite, often yield better performance than the more basic Runge-Kutta solver that was chosen at first. Unfortunately, a

⁴<http://www.llnl.gov/CASC/sundials>

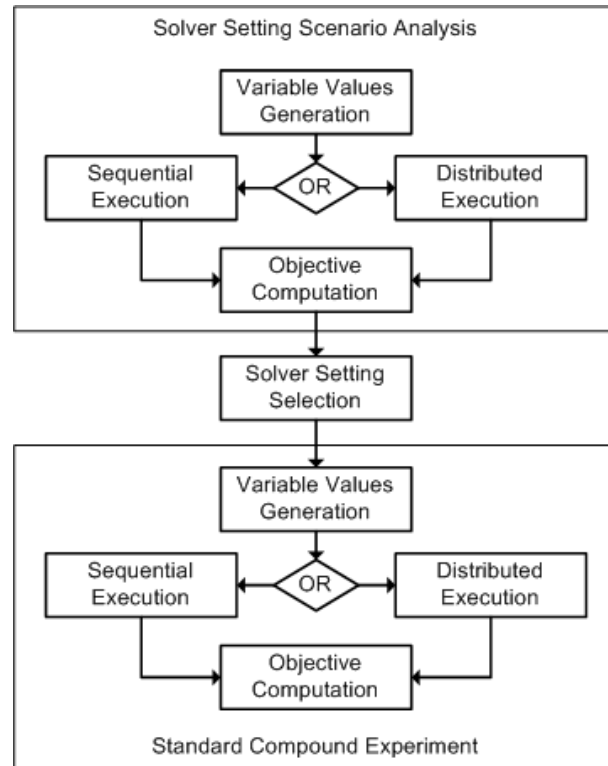


Figure 8.39: Solver Setting Scenario Analysis Procedure

number of simulation runs with manually configured CVODE settings initially lead to the belief that in this case CVODE would not be appropriate. However, CVODE has many combinations of solver settings, and it is very unpractical to test each of these manually. Therefore, the above-mentioned approach based on solver setting scenario analysis was applied. The CVODE solver settings and values that were chosen as variables for the analysis are listed in Table 8.12:

Table 8.12: Solver Settings for the Lux Case

Name	Values
AbsoluteTolerance	1e-3, 1e-4, 1e-5, 1e-6, 1e-7, 1e-8
RelativeTolerance	1e-3, 1e-4, 1e-5, 1e-6, 1e-7, 1e-8
IterationMethod	Functional (0), Newton (1)
LinearMultistepMethod	Adams (0), BDF (1)
LinearSolver	Dense (0), Diag (2), SPGMR (3)

It should be noted that *IterationMethod*, *LinearMultistepMethod* and *LinearSolver* are not represented by a real value, but by an enumerated value (a string in Tornado). Since scenario analysis cannot deal with enumerated values, these have to be mapped to an integer value (as is illustrated by the values between parenthesis in Table 8.12).

The values for *AbsoluteTolerance* and *RelativeTolerance* could in principle be specified through the manual placement method, but in this case they were specified through logarithmic spacing, as illustrated in Table 8.13. Since for *IterationMethod* and *LinearMultistepMethod* only 2 choices exist, these were specified through the manual placement method. This also applies to *LinearSolver*, for in this case the values (0, 2 and 3) are unrelated.

Table 8.13: Properties of *AbsoluteTolerance* and *RelativeTolerance* Scenario Analysis Variables

Name	Value
DistributionMethod	Logarithmic
Lower Bound	1e-8
Upper Bound	1e-3
Spacing	10

As can easily be seen from Table 8.12, the total number of solver setting combinations and hence the number of simulations to be run is $6 \times 6 \times 2 \times 2 \times 3 = 432$. A number of simulations as high as this evidently is only advisable in case the standard compound experiment that follows also consists of a high number of simulation runs, and in addition the expected speed-up for one simulation is substantial. In case these conditions are not fulfilled, the number of *a priori* simulations may have to be reduced in order to avoid overall performance degradation instead of the improvement that is hoped for.

During the scenario analysis, the average number of state evaluations as well as the total number of failures of the CVODE algorithm (due to stepsize underflow) were determined for each *IterationMethod* / *LinearMultistepMethod* / *LinearSolver* pattern, as is shown in Table 8.14. From this table follows that 1-0-0, 1-0-2, 1-0-3, 1-1-0 and 1-1-2 are unreliable since these settings sometimes result in CVODE failures. For the remaining settings, the difference with respect to reference trajectories was studied (*cf.* the average value of Theil's Inequality Coefficient in the table). From this, it was concluded that the 0-1-* patterns lead to appropriate solver settings for the simulation at hand, for all *AbsoluteTolerance* and *RelativeTolerance* settings considered. The 0-0-* patterns (which have a more or less comparable number of state evaluations) are less favorable since in this case the difference with respect to the reference trajectories is higher. On the basis of the number of state evaluations, one might be lead to believe that also the 1-1-3 pattern is a good option. However, in this case the resulting trajectories differ considerably from the desired reference trajectories (*cf.* the average TIC value, which is much higher than in the other cases). This indicates that this pattern is actually unstable, although no explicit CVODE failures have occurred.

Table 8.14: Total Number of Failures, Average Number of State Evaluations and Average TIC Value per Pattern

Pattern	#Failures	Avg(#StateEvals)	Avg(TIC)
0-0-0	0	150,128	0.00152
0-0-2	0	150,128	0.00152
0-0-3	0	150,128	0.00152
0-1-0	0	142,943	0.00098
0-1-2	0	142,943	0.00098
0-1-3	0	142,943	0.00098
1-0-0	4	55,879	N/A
1-0-2	26	29,835	N/A
1-0-3	2	167,988	N/A
1-1-0	6	6,052	N/A
1-1-2	27	8,782	N/A
1-1-3	0	1,594	0.23049

In view of the preceding analysis, it was decided for this application to retain the 0-1-0 pattern in combination with a *RelativeTolerance* and *AbsoluteTolerance* set to 1e-6. As can be seen from Figure 8.40, the number of state evaluations for these tolerances is more or less situated in the middle of the

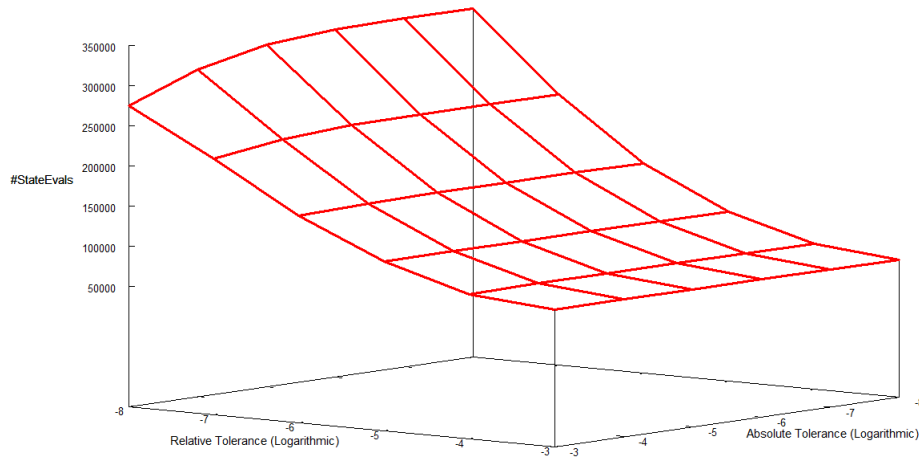


Figure 8.40: Number of *ComputeState* Evaluations with Respect to Absolute and Relative Tolerance for the 0-1-0 Pattern

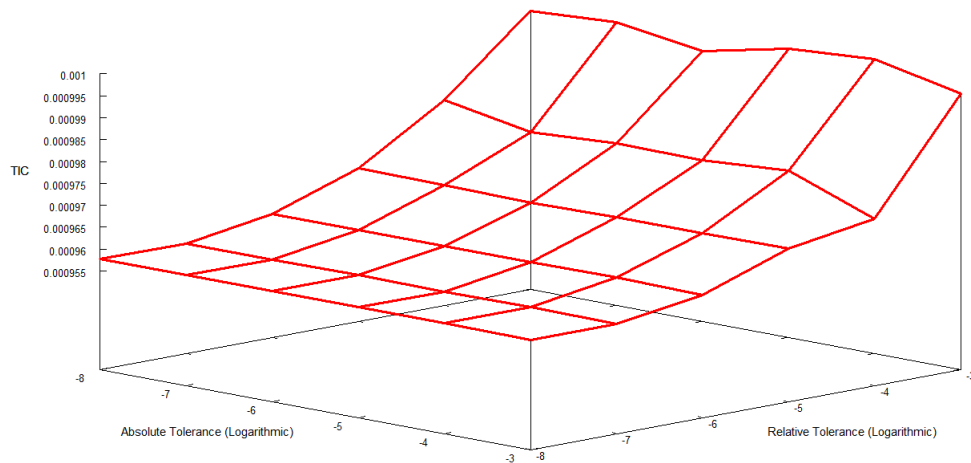


Figure 8.41: TIC with Respect to Absolute and Relative Tolerance for the 0-1-0 Pattern

surface depicted in the figure. As can also be seen from this figure, the number of state evaluations is quite sensitive to the relative tolerance. However, only for small relative tolerances, the absolute tolerance has some impact on the number of state evaluations. Figure 8.41 shows a similar behavior for the TIC.

Using the new solver settings, the average simulation time per run could be reduced from 53 to 33 s. The number of state evaluations could be more than halved (from 392,356 to 153,490). This implies that for sequential execution, the total simulation time for 1,000 runs could be reduced by $1,000 \times 20$ s, or approximately 5.5 hours. Compared to the approximate 2.5 hours that was spent on the *a priori* solver setting exploration, this results in a net performance gain of 3 hours. This clearly shows that because of the amplification of performance improvements (or degradations) that result from compound virtual experiments, it is beneficial to invest time into the exploration of the solver setting space *a priori*.

9

Modelling Environment

9.1 Introduction

The Tornado modelling environment is aimed at the creation and management of high-level model descriptions, and the generation of binary executable models. Historically, the modelling language that has been used for Tornado is MSL. Recently however, also partial support for Modelica has been included. This chapter first discusses both modelling languages. Subsequently, the executable model format and the model compilation process are discussed. Finally, also some information is provided on concepts such as layouts and model libraries.

For the MSL and Modelica languages, the discussion is very superficial. Detailed descriptions on the syntax and semantics of these languages can be respectively obtained from (Amerlinck et al., 2007; Vanhooren et al., 2003) and (Fritzson, 2004). Only those aspects of both languages will be highlighted that are of importance for the remainder of this manuscript.

Parts of this chapter have previously been published in (Claeys et al., 2006b), (Claeys et al., 2006d) and (Claeys et al., 2007a).

9.2 Modelling Languages

9.2.1 MSL

MSL (Model Specification Language) (Vangheluwe, 2000; Vanhooren et al., 2003) is a high-level, declarative, object-oriented modelling language that was developed by Hans Vangheluwe in 1997 on the basis of the ideas resulting from the 1993 ESPRIT Basic Research Working Group 8467 on “Simulation for the Future: New Concepts, Tools and Applications” (Vangheluwe et al., 1996). Actually, to be precise, only versions 3.0 and 3.1 of the MSL language can be regarded as high-level, declarative, object-oriented modelling languages. MSL version 1.0 was based on a column-based, spreadsheet-like format, which has also been adopted by languages such as NMF (Neutral Model Format) (Sahlin et al., 1996), and MSL version 2.0 was an experimental language based on nested lists that has never been applied to practical cases.

MSL version 3.* was originally intended to serve as a multi-formalism modelling language (*cf.* Chapter 4). In order to make the description of multiple formalisms possible, it is based on the concept of

(recursive) **records** containing **attributes**. These attributes are basically key-value pairs, where the key is an identifier and the value can be of several types, including record types. The ODE/AE formalism that is needed for Tornado is in principle only one of several formalisms that could potentially be described through the language, however in practice it is the only formalism that has received a full implementation.

Interestingly, MSL allows for the description of new formalisms within the language itself. These formalism descriptions are realized on the basis of primitive atomic constructs that are bootstrapped upon activation of a model library. In general, the MSL model compilation process first requires for the bootstrapping of primitive constructs to be carried out and is subsequently followed by the processing of formalism descriptions and the processing of the actual high-level model descriptions.

Being an object-oriented language, MSL relies on concepts such as **types**, **classes**, **objects**, **inheritance** and **composition**, which are shortly described below.

Types

MSL supports *Boolean*, *Integer*, *Real*, *Char* and *String* as basic types. From these basic types, other types can be constructed through **type signatures** or through **type subsumption** and **type extension**. Using type signatures one can create **type aliases**, **enumeration types**, **vector types**, **record types**, **union types** and **powerset types** (sets are unsorted collections). Through type subsumption (*i.e.*, specialization, subtyping), a new type can be created that restricts the set of possible values with respect to the original type. Finally, through type extension (*i.e.*, generalization), a new type is created by extending the record structure describing the original type with additional fields. Clearly, type subsumption and type extension form the basis for the MSL inheritance mechanism.

Classes

In MSL, types and classes are concepts that only have subtle differences. A class is in essence a type with default values. Classes can be constructed through type signatures, or through subsumption or extension. In contrast to types, classes can be provided with **annotations** next to regular attributes. Annotations are key-value pairs that contain meta-information. This type of information is not relevant to the model compilation process and is hence not interpreted by the model compiler. Instead, meta-information is copied as-is to the executable model code that is generated by the model compiler. Typically, annotations are used to specify information on the graphical representation of models.

Objects

Objects are instances of a class. An object binds the fields of a type or class record structure to concrete values. The last object declaration that is discovered by the model compiler when processing a model library is the one for which executable code will be generated.

Models

In MSL, ODE/AE models are described by a record that has a number of attributes predefined by the specification of the ODE/AE formalism. The attributes are the following:

- **Comments:** String that can be used to provide textual documentation for the model
- **Independent:** Set of independent variables
- **Interface:** Set of interface variables (input variables and/or output variables)

- **Parameters:** Set of parameters
- **State:** Set of worker and/or state variables
- **Initial:** Set of initial equations
- **Equations:** Set of state equations
- **Terminal:** Set of terminal (final) equations

Quantities

The term **quantity** is the collective noun that is used in MSL to refer to parameters, independent variables, input variables, output variables, state variables and worker variables. In MSL, all types of quantities are declared in a similar manner through their name, type, description and a list of attributes. The attributes of quantities are the following:

- **Value:** Real that represents the default value of a parameter or the initial condition of a state variable
- **Unit:** String that represents the unit of the quantity
- **Causality:** Value of an enumeration type that specifies whether the quantity is an input or output variable
- **LowerBound:** Real that represents the lower bound of the range of validity
- **UpperBound:** Real that represents the upper bound of the range of validity
- **Group:** String that represents the group membership of a quantity. Since the number of quantities for a certain model description can become quite large (especially in the case of parameters), grouping may be desirable. Applications can use the group attribute to present the overall set of quantities in a structured manner, and hence allow for easy navigation.
- **Fixed:** Boolean that specifies whether the quantity's value is allowed to change. Since MSL does not allow for constants to be declared as such, this property can be set for a parameter to obtain the same effect.
- **Hidden:** Boolean that specifies whether applications are to allow for a user to access a quantity. In case a quantity is hidden, it is considered to be internal to the system.
- **PDF:** String that represents the Probability Density Function (PDF) for a quantity. This attribute can be used by applications to sample values from distributions in order to perform Monte Carlo simulations. The availability of the PDF attribute is a first step towards the representation of stochastic models.

Equations

MSL allows for equations to be specified in the usual manner, *i.e.*, as a left-hand side (LHS) and right-hand side (RHS) separated by an equals sign. In the case of explicit equations, the LHS will be a single variable (possibly derived). For implicit equations, the LHS is a general expression. The RHS may always be a general expression. Table 9.1 lists the operators that can be used in equation expressions. Next to these operators, all functions from the C standard library as well as C functions with external implementations may be used. Also, a *IF-THEN-ELSE* construct can be used in the RHS of equations. Finally, a *FOREACH* construct allows for a vector-based equation pattern to be expanded into a set of scalar equations, and a *SUMOVER* construct allows for computing a sum over the elements of a vector.

Table 9.1: Operators in MSL

Operator	Description
<i>Arithmetic</i>	
$a + b$	Plus
$a - b$	Minus
$a * b$	Multiply
a / b	Divide
$a \% b$	Remainder after integer division
<i>Relational</i>	
$a == b$	Equal
$a != b$	Not equal
$a > b$	Larger than
$a >= b$	Larger than or equal
$a < b$	Less than
$a <= b$	Less than or equal
<i>Logical</i>	
$a b$	Or
$a \& \& b$	And
$!a$	Not

Composition

For constructing coupled models MSL uses a *connect* construct, similar to other modelling languages. Through the *connect* construct inputs and outputs of sub-models are connected in order to create connection graphs. However, next to the *connect* construct, MSL also has a *control* construct. The latter is to be used when the control signal of a controller (*e.g.*, PI, PID, On/Off, *etc.*) is to be connected to a parameter of another model. The reason why a special construct is needed in these cases is related to the fact that a parameter is defined as a quantity that remains constant during the course of a simulation. In case a controller would modify a parameter during the course of a simulation, this definition would be violated. In order to solve this problem the *control* construct is used to instruct the model compiler to transform a parameter into an input variable when it is controlled.

One might wonder if it would not be easier to model parameters that will potentially be involved in control actions as input variables from the onset. In principle the answer is affirmative, however there are two problems: firstly, it is not always easy to determine upfront which parameters will eventually be controlled, and secondly, modelling quantities that are parameters in the minds of modellers as input variables will appear unnatural to these modellers.

Preprocessor

The MSL language as such does not have a proper mechanism for encapsulation, structuring and re-use of model libraries. Basically, a MSL model library is a text file that contains type declarations, class declarations and - in case executable code is to be generated - at least one object declaration. In order to allow for the re-use of already existing model libraries for the development of new model libraries, a standard C preprocessor¹ is used in practice. A preprocessor allows for the use of directives such as *include*, *define* and *ifdef* to conditionally or unconditionally perform text replacements or carry out file inclusions.

In practice, the C preprocessor also plays a major role in the management of type and class templates.

¹http://en.wikipedia.org/wiki/C_preprocessor

These are descriptions that contain one or more textual placeholders that are to be replaced by actual content *before* the model compiler process takes place.

Model Libraries

Since its inception, several model libraries have been developed for MSL. Most of these are related to water quality processes, including treatment plants, sewer networks and river systems. The collection of water quality models that is available through MSL model libraries is larger than through any other declarative, object-oriented model library. An extensive number of models is shipped by default with the WEST product. Other models are available from the distributor of WEST on a commercial basis. Still others are custom developments made by MSL users that are only used within a limited scope. Table 9.2 gives an overview of the most important MSL libraries that are currently available.

Table 9.2: MSL Model Libraries

Name	Description
WWTP	Biological wastewater treatment modelling ASM1, ASM2, ASM2d, ASM3 (With and without temperature correction) Sensor models Controller models Biofilm models
RWQM	River water quality modelling
KOSIM	Sewer water quality modelling
SD	System dynamics modelling
PBM	Population balance modelling

Example

Listing 9.1 provides a MSL description of a simple two-step model for fat crystallization in chocolate. The model is due to (Foubert et al., 2005) and was also presented in Chapter 2. The model has 6 parameters, 6 state variables (2 of which are derived) and no interface variables. The initial equation section contains one equation and the state equation section has 6 equations. The terminal equation section is empty and has therefore been omitted. The $pow(x,y)$ function which computes x^y is implemented through the C standard library and is readily available in MSL code, since the C standard library is linked by default to executable model code resulting from the MSL compiler.

Listing 9.1: MSL Description of a Fat Crystallization Model

```
#include "generic.msl"

CLASS FoubertClass SPECIALISES PhysicalDAEModelType :=
{
  comments <- "Foubert_two-step_model_for_fat_crystallization";

  independent <-
  {
    OBJ t : Time;
  };

  parameters <-
  {
    OBJ K : Real := { : value <- 3; };
    OBJ a1 : Real := { : value <- 1; };
    OBJ a2 : Real := { : value <- 1; };
    OBJ b : Real := { : value <- 3; };
    OBJ n : Real := { : value <- 4; };
  };
}
```



```

    OBJ tind : Real := {: value <- 0.5; :};
};

interface <-
{
};

state <-
{
    OBJ f1 : Real := {: value <- 0; :};
    OBJ f2 : Real := {: value <- 0; :};
    OBJ h1 : Real := {: value <- 1; :};
    OBJ h2 : Real := {: value <- 0; :};
    OBJ r1 : Real := {: value <- 0; :};
    OBJ r2 : Real := {: value <- 0; :};
};

initial <-
{
    state.h2 = pow((1 + ((pow(0.99, (1 - parameters.n)) - 1) /
        exp(((parameters.n - 1) * (parameters.K * parameters.tind))))),
        (1 / (1 - parameters.n)));
};

equations <-
{
    state.r1 = (parameters.b * -state.h1);
    state.r2 = (parameters.K * (pow(state.h2, parameters.n) - state.h2));
    DERIV(state.h1, [independent.t]) = (state.r1 - state.r2);
    DERIV(state.h2, [independent.t]) = state.r2;
    state.f1 = (parameters.a1 - (parameters.a1 * state.h1));
    state.f2 = (parameters.a2 - (parameters.a2 * state.h2));
};
:};

OBJ Foubert : FoubertClass;

```

Critiques

When observing the current state of the MSL language, a number of critiques can be formulated that are related to the readability and expressiveness of the language:

- **Readability:** The fact that MSL was originally intended as a multi-formalism language has had an impact on its syntax. The number of tokens (such as “:=”, “<-”, “{:” and “:}”) that have a specific meaning is substantial. As a result, it is in practice quite difficult to write down a MSL model from scratch unless one has a template or reference manual available. Another issue is the fact that sections for the declaration of quantities (*parameters*, *independent*, *interface*, *state*) create their own namespaces (to a certain extent). As a result, one has to fully qualify identifiers when they are used in equations (e.g., *parameters.a1* instead of merely *a1*). This has a negative impact on readability and conciseness. On the other hand, one might also argue that these prefixes have a certain benefit of providing additional information on an identifier. However, following this approach further, one might also be interested in including type information as a prefix. This would lead to a mechanism comparable to Hungarian notation (see also Chapter 6), which is generally believed to be detrimental to readability, except for highly technical, low-level code.
- **Expressiveness:** Although MSL has been shown to be sufficiently powerful to represent water quality models (Vanhooren et al., 2003), it lacks expressiveness to be a viable candidate for multi-domain modelling. Issues include the following:
 - **Hybrid systems:** MSL does not contain any constructs that specifically allow for modelling discrete aspects of continuous systems. Since water quality processes usually do not have

- many discrete aspects, and moreover, discrete events occur with low frequencies, this rarely poses a problem for these models. However, in other domains (such as electronics, mechanics, *etc.*) extensive support for hybrid system modelling is a necessity in order to obtain meaningful simulation results.
- **DDE's:** MSL does not have any high-level constructs to model Delay-Differential Equations (DDE's). In case variable values for a past time point are required, they must be retrieved from a buffer that is implemented externally (in C).
 - **Array slicing:** Array slicing is an operation that extracts certain elements from an array and packages them as another array, possibly with a different number of indices (or dimensions) and different index ranges. One typical example is extracting a row (or a column) from a matrix to be used as a vector. In MSL, array slicing is only partially available through the *FOREACH* construct.
 - **Algorithmic code:** MSL allows for a declarative description of mathematical equations. This implies that no information is supplied about how to compute those equations. The model developer is not even required to enter the equations in a particular sequence, since sorting of equations is handled by the model compiler. On the one hand, declarative modelling offers convenience to a user. However, on the other hand it excludes the modelling of specific types of behavior since certain algorithms cannot be modelled conveniently in a declarative manner (for instance sorting algorithms are much easier to implement in an algorithmic way). In such cases, MSL allows for algorithms to be implemented in a procedural manner through external C code. Although this solution works in practice, it is not desirable from the point of view of readability since in this way a part of the model is coded in a declarative manner and resides in a MSL model library, and another part is coded in a procedural manner in C (or another language that supports C-linkage) and resides in an external source code base.
 - **Packaging and namespaces:** As mentioned earlier, the MSL language does not have proper constructs for encapsulation, structuring and re-use of model libraries. In order to circumvent this problem, a preprocessor is used. Although the preprocessor approach is very powerful if used wisely, it is also error-prone. For convenient management of large model libraries a proper packaging and namespace approach is desirable.
 - **Class parameters:** MSL does not have a class parameter mechanism. In order to allow for re-use of class templates one therefore has no other option but to rely on (non type-safe) textual replacements on the basis of a preprocessor.
 - **Matrix computations:** MSL does not directly support any matrix-based computations (such as addition, subtraction, multiplication, *etc.*). This limitation can however be circumvented through explicit expansion to a number of scalar operations, or by calling an external function with C-linkage. Evidently, this causes for inconvenience.

9.2.2 Modelica

Modelica (Fritzson, 2004) is similar to MSL in the sense that it is high-level, declarative and object-oriented. In fact, both MSL and Modelica were designed according to the ideas resulting from the 1993 ESPRIT Basic Research Working Group 8467 on “Simulation for the Future: New Concepts, Tools and Applications” (Vangheluwe et al., 1996).

The design of the Modelica language is the result of an international effort. The main objective was to facilitate the exchange of models and model libraries amongst researchers and practitioners. The design approach of Modelica builds on non-causal modelling with ODE and DAE equations and the use of object-oriented constructs to facilitate the re-use of modelling knowledge. At the time the design of Modelica started (1996), an extensive number of languages with similar goals existed. Examples include

ASCEND (Piela et al., 1991), Dymola (Elmqvist et al., 1996), gPROMS (Barton and Pantelides, 1994), NMF (Sahlin et al., 1996), ObjectMath (Fritzson et al., 1995), Omola (Mattsson et al., 1993), SIDOPS+ (Breunese and Broenink, 1997), Smile (Kloas et al., 1995), ULM (Jeandel et al., 1996) and VHDL-AMS (IEEE, 1997). The aim of Modelica was to unify the concepts of these languages, in order to introduce a common syntax and semantics (Mattsson et al., 1998), *i.e.*, a *lingua franca* for mathematical modelling.

Version 1 of the Modelica language definition was released in September 1997. Since then, several minor and major updates have been published. In September 2007, version 3 was presented.

The development of Modelica is governed by the Modelica Association², which is a non-profit, non-governmental organization with the aim to develop and promote the Modelica language for modelling, simulation and programming of physical and technical systems and processes. The Modelica Association owns and administrates incorporeal rights related to Modelica, including but not limited to trademarks, the Modelica Language Specification, Modelica Standard Libraries, *etc.*, which should be generally available for the promotion of industrial development and research. The Modelica Association currently has 8 institutional members and approximately 60 individual members.

Classes

The class concept is fundamental to Modelica and is used for a number of different purposes. Almost anything in Modelica is a class. However, in order to make Modelica code easier to read and maintain, special keywords have been introduced for specific uses of the class concept. The keywords *model*, *connector*, *record*, *block* and *type* can be used instead of *class* under specific conditions, hence these special classes are named *restricted* classes:

- The **model** restricted class is the most commonly used type of class for modelling purposes. Its semantics are almost identical to the general class concept in Modelica. The only restriction is that a model may not be used in connections.
- A **record** is a restricted class for specifying data without behavior. No equations are allowed in the definition of a record or in any of its elements. A record may also not be used in connections.
- The **type** restricted class is typically used to introduce new type names using short class definitions, even though short class definitions can be used to introduce new names also for other kinds of classes. A type class may be only an extension to one of the predefined types, a record class, or an array of some type.
- The **connector** restricted class is typically used for connectors, *i.e.*, ports for communicating data between objects. Therefore no equations are allowed in the definition of a connector or within any of its elements.
- A **block** is a restricted class for which the causality is known for each of its variables. Blocks are typically used for signal-related modelling since in that case the data-flow direction is always known. A block may also not be used in connections.

Other types of classes with additional restrictions with regard to the situations under which they can be used are:

- The **function** concept in Modelica corresponds to mathematical functions without external side-effects, and can be regarded as a restricted class with some enhancements.
- A **package** is a restricted and enhanced class that is primarily used to manage namespaces and to organize Modelica code.

²<http://www.modelica.org>

Models

In Modelica, models are constructed and structured in a way similar to MSL. New models can be created through inheritance or through composition. A model structure consist of a section containing quantity and sub-model declarations, and sections for initial and state equations. Finally, also algorithmic sections can be added to models.

Quantities

Also in a way similar to MSL, quantities in Modelica have a number of attributes of which the most important are the following:

- **Unit:** String that represents the unit of the quantity
- **Min:** Value that represents the lower bound of the range of validity
- **Max:** Value that represents the upper bound of the range of validity
- **Start:** Initial value
- **Fixed:** Indicates that the quantity is not allowed to change

The types of quantities that are distinguished by Modelica are constants, parameters, input variables, output variables, worker variables and state variables. Modelica always implicitly assumes that the independent variable is time.

Model Libraries

The number of libraries that is available for Modelica (both commercially and non-commercially) is extensive. Table 9.3 gives an overview of the most important libraries that are in use to date.

Example

Listing 9.2 provides a Modelica description of the same two-step fat crystallization model as in Listing 9.1. One will notice that the Modelica description is structurally similar to the MSL description. However, the Modelica description is more concise and more readable.

Listing 9.2: Modelica Description of a Fat Crystallization Model

```
model Foubert "Foubert two-step model for fat crystallization"
  parameter Real a1 = 1;
  parameter Real a2 = 1;
  parameter Real b = 3;
  parameter Real K = 3;
  parameter Real n = 4;
  parameter Real tind = 0.5;

  Real f1;
  Real f2;
  Real r1;
  Real r2;

  Real h1(start = 1.0);
  Real h2;

initial equation
```

Table 9.3: Modelica Model Libraries

Name	Description
<i>Standard Libraries</i>	
Modelica Standard Library	Basic mechanics (1-D / 3-D), electronics, thermodynamics and fluids
Modelica Magnetic	Electromagnetic devices with lumped magnetic networks
Modelica Fluid	1-D thermo-fluid flow in networks of pipes
<i>Free Libraries</i>	
Wastewater	Biological wastewater treatment: ASM1, ASM2d, ASM3
ObjectStab	Power systems voltage and transient simulation
ATplus	Building simulation and control
SPICELib	Contains some of the capabilities of the electric circuit simulator PSPICE
SystemDynamics	Modelling according to the principles of system dynamics of J. Forrester
TechThermo	Technical thermodynamics
FuzzyControl	Fuzzy control
ThermoPower	Thermal power plants
BondLib	Bond graph modelling
ExtendedPetriNets	Modelling Petri nets and state transition diagrams
FuelCellLib	Fuel cells
OSSFluidFlow	Quasi steady-state fluid pipe flow
SPOT	Power systems, both in transient and steady-state mode
ModelicaDEVS	Discrete event modelling using the DEVS formalism
MultiBondLib	Modelling with multi-bond graphs
<i>Commercial Libraries</i>	
PowerTrain	Vehicle power trains and planetary gearboxes with speed and torque dependent losses
SmartElectricDrives	Hybrid electric vehicles and new alternative concepts with electrical auxiliaries
VehicleDynamics	Vehicle dynamics
AirConditioning	Air conditioning systems
HyLib	Hydraulic components
PneuLib	Pneumatic components
CombiPlant	Combined cycle power plants and heat recovery steam generators
HydroPlant	Hydro power plant components

```

h2 = ( 1 + ( (0.99^(1 - n) - 1) / exp( (n - 1) * K * tind) ) ) ^ ( 1 / (1 - n) );

equation

r1 = b * -h1;
r2 = K * ((h2^n) - h2);
der(h1) = r1 - r2;
der(h2) = r2;
f1 = a1 - a1*h1;
f2 = a2 - a2*h2;

end Foubert;

```

Appreciation and Critique

Modelica is generally appreciated as one of most advanced languages for physical modelling currently available. It combines powerful modelling concepts and constructs with conciseness and easy readability. More specifically, it has a solution for the shortcomings of MSL that were discussed earlier: hybrid system modelling is supported, DDE's can be modelled through the *delay()* function, array slicing is supported, special *algorithm* sections allow for algorithmic code to be added to otherwise declarative model

descriptions, packaging and namespaces are available, class parameters can be implemented through the *replaceable* and *redeclare* modifiers and basic matrix operations are directly supported by the language.

Unfortunately, the main advantage of the language, *i.e.*, the fact that it is an open standard that is supported by several tools and products, may also be perceived as its greatest disadvantage. The level at which the language is supported varies from tool to tool. Models can therefore not always be successfully exchanged between tools, especially in case advanced constructs are used. For instance, the OpenModelica implementation, which is especially attractive for researchers since it is freely available, does not yet support a number of constructs that are of particular importance for water quality modelling. The most important feature in this respect that is defined by the Modelica language, but has not yet been implemented in OpenModelica is the ability to declare vectors over an enumeration type (in order to be able to use elements of the enumeration type as vector indices, instead of integers). Since most water quality models are based on component vectors with named elements, this feature is essential.

In December 2007, the Open Source Modelica Consortium (OSMC) was created. OSMC is a non-profit, non-governmental organization that aims to further extend and promote the development and usage of the OpenModelica open source implementation of the Modelica language, and its related tools and libraries. Within the limitations of its available resources, OSMC will provide support and maintenance of OpenModelica, support its publication on the web, and coordinate contributions to OpenModelica. It is expected that the creation of OSMC will be beneficial for leveraging OpenModelica so that it becomes a reference implementation for the full Modelica Standard Specification.

9.3 Executable Models

High-level declarative models as described in languages such as MSL and Modelica cannot be directly used in the scope of virtual experiments. They need to be transformed by a model compiler to an executable model format based on a general-purpose programming language, and subsequently be compiled and linked by a regular compiler and linker. Executable models need to be efficient, both in terms of speed and memory consumption, and should not contain any constructs that cannot be directly executed by a computer. In other words, executable models should be fully procedural and need to be optimized for performance. Also, hierarchical relationships (inheritance as well as composition) should no longer be present, *i.e.*, executable models should preferably be **flat** in order to facilitate further optimizations. In this section, the nature of the executable model format that is used by Tornado will be discussed, as well as the experiences that have led to the current format.

In the case of high-level modelling languages, there have been several attempts at standardizing and unifying model representations across software environments. Evidently, Modelica is a prime example of these efforts. However, in the case of executable model representations only few such efforts have been undertaken, consequently executable models can generally not be exchanged between software environments. The main reason for this is that the executable model format and the virtual experiment executor that uses it, are usually strongly intertwined. It is difficult to standardize the executable model format without also imposing certain conditions and mechanisms on the executor.

One effort to come to a standardized executable model format is **DSblock** (Dynamic System block) (Grubel et al., 1994; Otter and Elmqvist, 1994). This representation was originally FORTRAN-based but was later also ported to C. During the primal phase of the design of Modelica it was suggested to use DSblock as a standard mechanism for executable model representations. At that stage, the Dymola product actually used DSblock internally. However, in later stages the original standard was abandoned. At this moment, most tools that are based on Modelica use their own executable model format, which may or may not be loosely based on the original DSblock standard.

Another executable model format that could be considered a standard is actually a *de facto* standard that is imposed by MATLAB/Simulink and is named **S-function**. An S-function is a computer language

description of a Simulink block. S-functions can be written in MATLAB, C, C++, Ada, or FORTRAN. C, C++, Ada, and Fortran S-functions are compiled as MEX-files using the *mex* utility. As with other MEX-files, they are dynamically linked into MATLAB when needed. S-functions use a special calling syntax that enables them to interact with Simulink equation solvers. This interaction is very similar to the interaction that takes place between the solvers and built-in Simulink blocks. The form of an S-function is very general and can accommodate continuous, discrete, and hybrid systems. In order to compile and link S-functions, access to the MATLAB/Simulink SDK is needed, which is why S-functions, albeit being a *de facto* standard, cannot be considered a *neutral* executable model format.

The first step in the design of an executable model format for Tornado was to decide on the information that needs to be captured. In contrast to some other related software environments, Tornado adheres to the principle of strictly separated modelling and experimentation environments. This implies that executable models need to be self-describing, since they are the only piece of information that is transferred from the modelling environment to the experimentation environment. As a result, next to the computational information, executable models in Tornado need to contain information on how the original composition hierarchy relates to its flat, executable counterpart. Interaction between the executable model and the experiment executor's solver occurs through the model's computational information, however interaction with applications that sit on top of Tornado occurs through the latter. For instance, a user wants to specify a value for the *Kla* parameter of an activated sludge tank by using that parameters fully qualified, hierarchical name, *e.g.*, *.ASU.Kla*, instead of through an index into a long vector of scalars.

Figure 9.1 presents the flow of the executable model generation process. From high-level modelling languages such as MSL or Modelica, executable model code is generated by a model compiler. Possible executable model formats are DSblock, S-functions and the Tornado format. A general-purpose language compiler and a linker is used to generate a stand-alone executable or a dynamically-loadable library, which is then used by the virtual experiment executor.

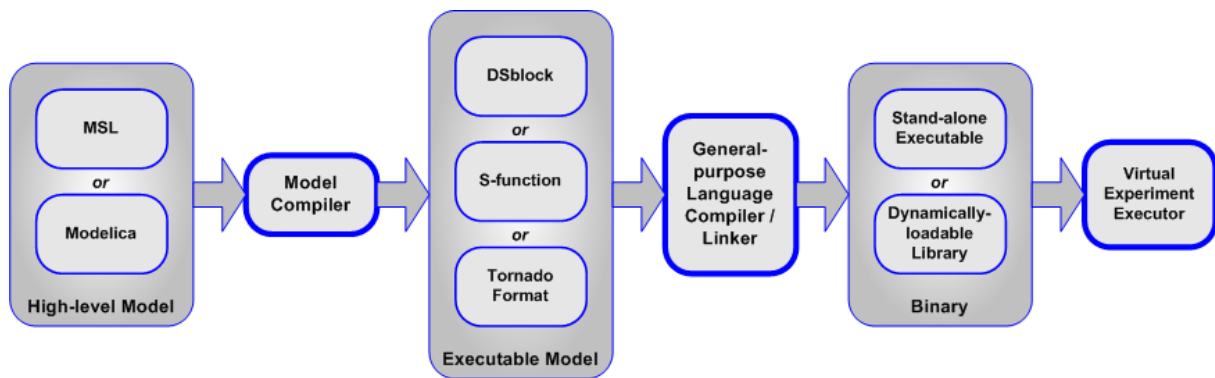


Figure 9.1: Flow of the Executable Model Generation Process

9.3.1 Symbolic Information

For Tornado-I it was decided to use C++ as a language for the representation of executable models, both for the **computational** information (*i.e.*, the equations), as for the **symbolic** information (*i.e.*, the relationship between the original hierarchical model and its flat counterpart). Unfortunately, experience has shown that this approach leads to several problems, especially in the case of complex models. Since in water quality management, complex models have in recent years become a rule rather than an exception, the Tornado-I executable model format had to be substantially modified in the scope of Tornado-II.

The problems that were encountered with the representation of both representing symbolic and computational model information in C++ are the following:

- **Speed of compilation of symbolic information:** Computational information is largely string-based. It deals with names and attributes of parameters, variables and sub-models and their relationships to elements of the executable model. C++ compilers tend to be quite slow when large amounts of string literals need to be processed. The compilation times for large executable models (> 300,000 AST nodes, *cf.* Chapter 3) represented in the Tornado-I format were therefore inconveniently long (minutes to hours on an average workstation), and generally unfavorable for the scalability of the model compilation approach.
- **Programming interface compatibility:** Experience has shown that changes to the representation of symbolic model information are often required, *e.g.*, attributes may be added or removed. Since in Tornado-I symbolic information is presented in a compiled binary form to the virtual experiment executor, changes to the interface quickly lead to incompatibilities. As a result, old binary executable models can no longer be used each time the representation changes. Evidently, this is highly undesirable.
- **Linkage incompatibility of C++ code:** A major issue is the fact that C++ code compiled with different compilers is not compatible at linkage level, in contrast to C code. So, one cannot simply compile a C++ module with one compiler, and then link it to another C++ module that has been compiled with another compiler. As a result of this, the C++ compiler that is used to compile executable models for Tornado-I, determines the compiler that can be used to compile the experiment executor. Conversely, when an experiment executor is supplied to a user, that user has no choice but to use the same compiler to compile executable models. In practice, this limitation proves to be very inconvenient. Especially since different C++ compilers do not have the same characteristics, *e.g.*, some compilers are better at generating meaningful error messages and enforcing standards compliance, while others are good at generating efficient binary code. Ideally, one therefore would like to have the opportunity to use different compilers for different purposes. The linkage incompatibility of C++ code makes this impossible.

Another aspect that illustrates the importance of being able to use one's compiler of choice is purchase price. Commercial compilers can be quite expensive, being able to use a low-priced or free compiler is therefore an important asset for researchers. However, it must be pointed out that this issue has recently lost some of its importance since Microsoft has made its Visual C++ compiler available free of charge (as part of the Visual Studio Express Edition). Borland C++ 5.5 is already available free of charge since quite a number of years.

- **Debugging:** The process of generating the correct relationships between the original hierarchical model representation and the executable model is quite error-prone. During the model compiler development process it is frequently required to explore the symbolic information that was generated in order to investigate problems. Performing these manipulations on binary code has proven to be difficult and inconvenient. Actually, in certain situations the errors in the symbolic model information might be such that an application crash occurs upon instantiation. In these cases it is even more difficult to investigate the problem.

In view of the problems with the Tornado-I format, it was decided to make a number of drastic changes to the executable model format as it is used in the scope of Tornado-II. Firstly, it was realized that symbolic information and computational information are essentially different in the sense that the first is only to be read once by the experiment executor, after which the executor can build up its own efficient internal representation for future use. It is therefore not strictly necessary to provide for the utmost efficiency while loading the symbolic information, and hence a binary representation is not required. It was therefore decided to use an open, text-based representation, more specifically on the basis of XML. Advantages are many:

- XML can be generated much faster than C++ code (containing symbolic information) can be compiled.
- When using a permissive XML parser, changes to the representation of symbolic information can be easily dealt with and hence do not necessarily need to lead to incompatibilities.
- Errors in the XML representation of symbolic information can be easily discovered since XML is a text-based, open format. In principle, any text editor can be used to investigate the information.
- In the context of internationalization, all strings related to an executable model can easily be translated by modifying the XML file containing the symbolic information.

For water quality models, the size of XML files containing symbolic information typically ranges from 100 kB to 10 Mb. On an average workstation, processing large symbolic model information files takes a number of seconds at most. The time spent on loading these files is therefore negligible with respect to the time gained thanks to the fact that there is no more need for the compilation of code, which could easily take minutes to hours.

A slight disadvantage of the use of XML is that an executable model is now composed of two parts: a binary part containing computational information, and an XML-based part containing symbolic information. This causes for some heterogeneity and the fact that care has to be taken to always jointly distribute both files.

9.3.2 Computational Information

The computational information of the Tornado-II format consists of **data containers** and **event routines**. The data containers are arrays (with standardized names) in which scalar data items (that are produced during flattening) are aligned. A distinction is made between the following types of data items:

- **Params**: Parameters
- **IndepVars**: Independent variables
- **InputVars**: Input variables
- **OutputVars**: Output variables
- **AlgVars**: Algebraic (worker) variables
- **DerVars**: Derived (state) variables
- **Derivatives**: Derivatives of state variables (*i.e.*, left-hand sides of differential equations)
- **Previous**: Values of a number of selected variables at the previous major integration time point
- **Residues**: Left-hand sides of implicit equations

As the number of data items of complex models can be very large, these arrays can easily hold 10,000 items or more. For water quality models, especially the *Params* and *AlgVars* arrays can be of considerable size.

Event routines are functions (with standardized names) that perform the actual computations. Hence, these contain the executable counterparts of the declarative equations of the original high-level model. The most important event routines are the following:

- **ComputeInitial:** Computes parameters and/or initial conditions of derived variables on the basis of other parameters. Is only executed once at the beginning of each simulation.
- **ComputeState:** Computes all variables that are needed to compute the right-hand sides of differential equations. Is executed at each minor integration time point. A minor time point is a time point at which an integration solver performs intermediate computations that are to be used in the computation of the values of derived variables for the next major time point.
- **ComputeOutput:** Computes variables that do not contribute to the state of the system. Is executed at each major integration time point.
- **ComputeFinal:** Computes variables for which only the final value is required. Is only executed once at the end of each simulation.
- **Update:** Copies the current values of a number of selected variables to the *Previous* array.
- **CheckBounds:** Checks the current values of all variables against their range of validity.

It is the model compiler back-end's responsibility to ensure that the equations in each section are sorted, *i.e.*, that no variable is used before it has been computed. Figure 9.2 shows the execution sequence of these routines, represented as a deterministic finite state machine. One might argue that the execution of the *ComputeOutput* routine at each major time point is not strictly needed. Indeed, in principle the execution of *ComputeOutput* is only required when output to output files, output buffers, *etc.* is desired, *i.e.*, at output communication intervals. These communication time points might be wider spaced than major integration time points, hence allowing for *ComputeOutput* to be executed less frequently. However, one should take into account that multiple output acceptors can be associated with a simulation, and that all of these output acceptors might have different communication intervals (both in terms of size as in terms of spacing method (linear, logarithmic)). Determining the overall list of time points for which output is desired is therefore non-trivial. Also, output calculator variables can also be associated with a simulation. This type of output acceptor can be used as any regular output variable, which implies that it can be used as a quantity in the computation of an objective function. In this case, "output" (*i.e.*, the computation of the calculator variable) is desired at every time point. For all of these reasons, it has been decided to always execute *ComputeOutput* unconditionally at every time point. The impact of this overzealous execution frequency in case the desired output pattern is non-dense remains to be investigated.

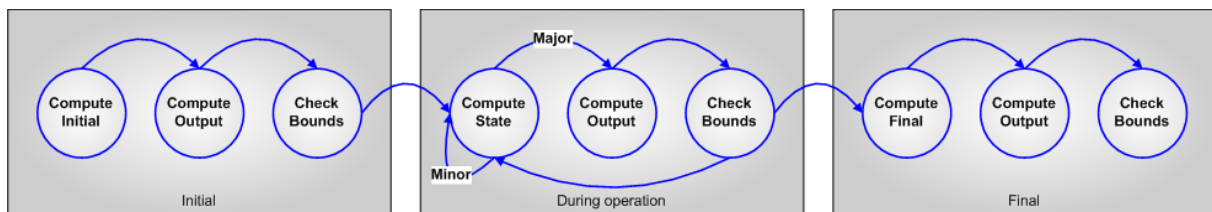


Figure 9.2: Compute Event Routines

Figure 9.3 contains an informal ER diagram of the internal representation of an executable model. Basically, an executable model (*Model*) is comprised of a symbolic model description (*SymbModel*), a computational model description (*ExecModel*) and a description of the links between both (*Links*). The computational information consists of data container arrays (*ParamValues*, *IndepVarValues*, *InputVarValues*, *OutputVarValues*, *AlgVarValues*, *DerVarValues*, *Derivatives*, *Previous* and *Residues*) and event routines (*ComputeInitial*, *ComputeState*, *ComputeOutput*, *ComputeFinal*, *Update* and *CheckBounds*). The symbolic information is hierarchical and mimics the composition hierarchy of the high-level model. The

symbolic representation of the model (*SymbModel*) contains lists of symbolic representations of parameters, independent variables, input variables, output variables, algebraic variables and derived variables (respectively *SymbParam*, *SymbIndepVar*, *SymbInputVar*, *SymbOutputVar*, *SymbAlgVar* and *SymbDerVar*). Next to these, a symbolic model representation also contains a list of symbolic sub-model representations (*SymbModel*). Each symbolic description consists of a name, a description and a number of attributes. Each symbolic object is also registered in executable model symbol tables using the object's fully qualified name as a key. In this way, efficient lookups of symbolic objects on the basis of their fully qualified name can be guaranteed. Two types of symbol tables exist: specialized tables that only contain objects of a certain type (e.g., only parameters or models), and a general table that contains all objects, irrespective of their type. In case one only disposes of the name of an object, and does not know its type, the general table should be used. Otherwise, one should refer to one of the specialized tables. Fully qualified object names are also used in links. These links basically map a fully qualified object name to a position in a specific data container.

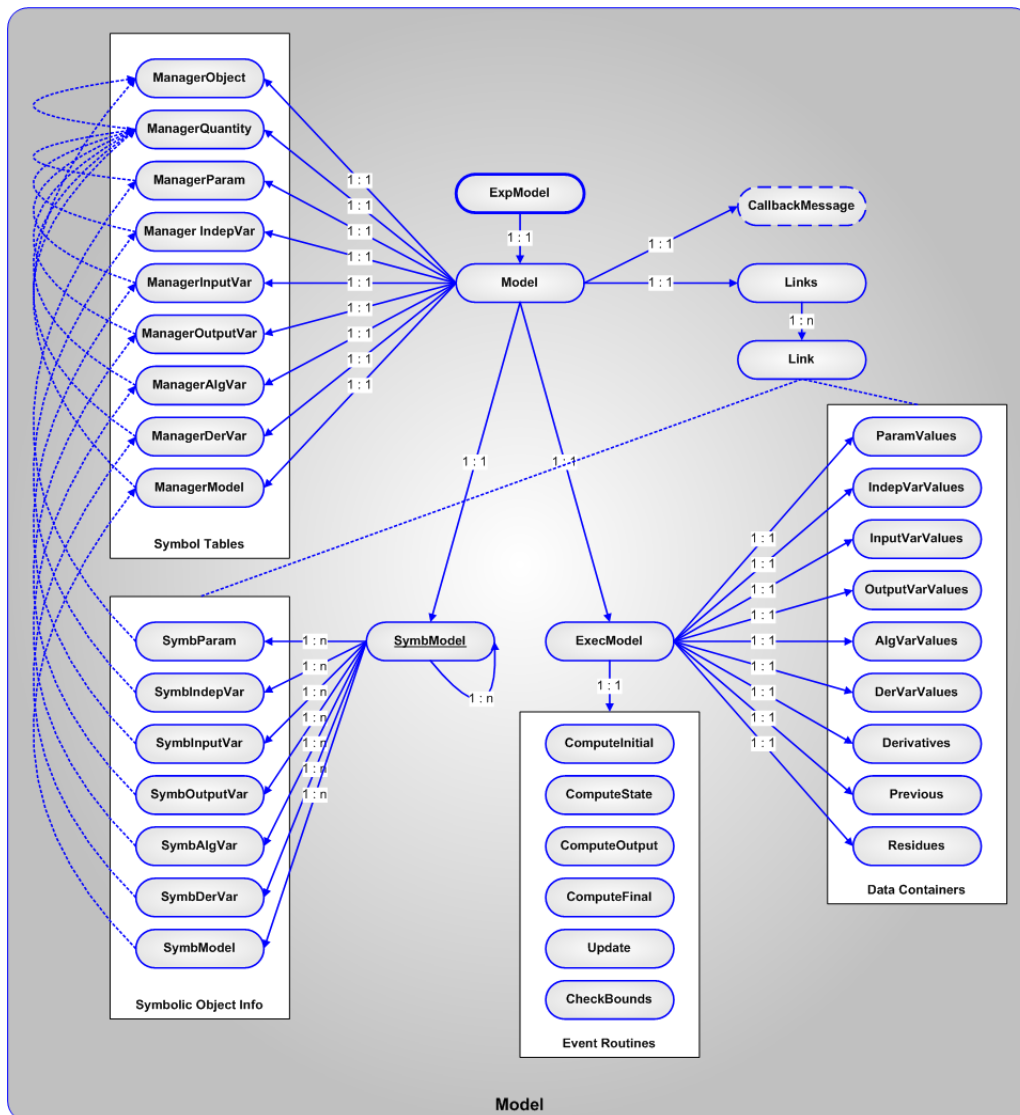


Figure 9.3: Executable Model ER Diagram

Since computational information needs to be as efficient as possible, a representation in a general-purpose programming is also used in the scope of Tornado-II. However, instead of C++ (as in Tornado-I),

the language that is adopted is C. Given the fact that the use of C++ leads to the problems mentioned earlier, and since there is no need to represent complex datatypes in the scope of computational information, C has proven to be a good option. Thanks to the use of C, the user can use his C compiler of choice to compile computational information, irrespective of the C++ compiler that was used to compile the virtual experiment executor. As in the case of C++ compilers, each C compiler has its own strengths and weaknesses. Some compilers are better at issuing messages, ensuring standards compliance and generally guarding against potential problems and pitfalls, while others are better at generating efficient code. In certain cases it might therefore be beneficial to have several C compilers at one's disposal. The following compilers have thus far been tested on Tornado-II computational information:

- (win32) Microsoft Visual C/C++ (MSVC) 6.0, 7.1 and 8.0³
- (win32) Borland C/C++ (BCC) 5.5⁴
- (win32) LCC⁵
- (win32) INTEL C/C++ 9.0⁶
- (win32) Open Watcom C/C++ 1.4⁷
- (win32) GNU C/C++ 3.*⁸
- (linux) GNU C/C++ 3.*⁹

In practice, Open Watcom has proven to be unusable since the compilation times for complex models are prohibitively long. All other compilers are usable, but have their own specific advantages and disadvantages. Borland C++ 5.5 for instance is able to compile code quickly, but the resulting binaries are not efficient. INTEL C++ 9.0 on the other hand has a slow compilation process, but generates highly efficient binaries. In general, MSVC 7.1 and 8.0 have been found to yield the best compromise between efficiency of the compilation process and efficiency of generated code.

Table 9.4 provides some information on the application (on the Windows platform) of several C compilers to a closed-loop model of the Galindo WWTP plant in San Sebastian, Spain. This model is of relatively high complexity (approximately 300 state variables, 6000 parameters and 4000 worker variables). For a workstation with an INTEL Pentium M 1600 MHz processor and 512 Mb of RAM, the time to compile the model (with optimization enabled), the time to execute a simulation, the size of the executable model binary, and the compiler availability are given. As can be seen from the results, the fastest execution time is given by the INTEL compiler, however at the expense of a compilation time that is substantially longer than the compiler that provides the second fastest code, *i.e.*, Microsoft Visual C/C++ 7.1/8.0.

One might argue with the statement that Microsoft Visual C/C++ yields the best compromise between compilation and execution times, since Borland C/C++ and LCC are able to compile many times faster than the Microsoft compilers. However, one should keep in mind that the compilation process is a one-time operation, whereas the models might be run many times. For instance, in case an ExpScen or ExpMC experiment is run with 100 shots, the time lost on compilation is almost instantly compensated by faster execution times.

³<http://www.microsoft.com>

⁴<http://www.codegear.com>

⁵<http://www.cs.virginia.edu/~lcc-win32>

⁶<http://www.intel.com>

⁷<http://www.openwatcom.org>

⁸<http://www.mingw.org>

⁹<http://www.gnu.org/software/gcc>

A compiler that is typically popular amongst researchers is the GNU C compiler (GCC). In general, GCC has two major advantages: the fact that it is freely available, and the fact that it supports the same input syntax on all supported platforms (*cf.* portability of code). Apart from that, it is not very efficient, as can be seen from Table 9.4. Actually, we have experienced that on the Linux platform, a version of Tornado compiled for Windows with the MSVC compiler and run through the WINE¹⁰ emulator actually runs faster (approximately 10 to 20%) than a version that is “natively” compiled for Linux with the GCC compiler.

Table 9.4: Performance of C Compilers for the Galindo-CL Model (Windows platform)

Compiler	Execute (s)	Compile (s)	Model size (KByte)	Availability
Borland C/C++ 5.5	752	1	352	Free
LCC	689	2	291	Free
GNU C/C++ 3.*	592	23	221	Free
Microsoft Visual C/C++ 6.0	561	32	232	Commercial
Microsoft Visual C/C++ 7.1	503	42	228	Commercial
Microsoft Visual C/C++ 8.0	502	41	227	Free (Visual Studio Express Ed.)
INTEL C++ 9.0	501	58	226	Commercial

Given the different characteristics of C compilers, one could for instance imagine a situation where Borland C++ is used during the model development process (because it allows for quickly generating binary models), while Microsoft Visual C++ is used during the model deployment process (because it allows for fast execution). However, there is no real need for involving two different compilers, since one can also simply switch off the optimization options of the Microsoft compiler (*e.g.*, use */O1* instead of */O2*). This will lead to compilation and execution times that are similar to the Borland compiler (with full optimization switched on).

One issue regarding executable models that remains to be discussed is the manner in which they are attached to the virtual experiment executor. In other software environments that rely on the generation of executable models such as Dymola and OpenModelica, a stand-alone simulator executable is built for each executable model. More specifically, the binary model code is statically linked to a solver library and the resulting executable is run as an external process by the encapsulating software environment. The advantage of this approach is that the simulator executable is a self-contained program that - if desired - can be run individually, without an encapsulating context. However, from our perspective, disadvantages are many. Firstly, run-time communication between the simulation executable and the encapsulating environment is not easy to accomplish. Consequently, simulation results are often captured in a data file and are only visualized or further processed *a posteriori*, *i.e.*, after the simulation executable has stopped working. In many cases, this is undesirable as on-line rather than off-line visualization is required. Another result of the fact that communication between stand-alone executables and an encapsulating environment is not easy to accomplish is that run-time control (stopping, pausing and starting simulations) is difficult. One last disadvantage of the stand-alone executable approach is that each executable model is linked against the same solver library. This leads to a set of executable model binaries that each partly contain the same information, causing an unnecessary consumption of disk space.

In view of the disadvantages related to stand-alone executables, the approach that is taken by Tornado is based on dynamic instead of static linkage of executable models. Executable models are compiled and linked into a DLL on Windows and a shared object on Linux. Upon request these dynamically-linked objects are loaded by the virtual experiment executor at run-time. Processing of simulation results as they are generated, and start/stop control of virtual experiments is easy. Also, no unnecessary disk space

¹⁰<http://www.winehq.org>

is consumed.

The low-level executable model format that is used by Tornado is named **MSL-EXEC** (EXECutable representation of Model Specification Languages). In order to make a clear distinction with the high-level MSL language, the latter is also referred to as **MSL-USER** (USER-oriented representation of Model Specification Languages).

9.4 Model Compilers

As mentioned before, Tornado relies on a high-level, declarative, object-oriented modelling language. Through these languages, complex models can be constructed using techniques such as inheritance (*i.e.*, the process of deriving models from base models by inheriting properties) and composition (*i.e.*, the process of creating coupled models by connecting inputs and outputs from other coupled and/or atomic models). The process of converting high-level models to executable code is called **model compilation**, and typically consists of two phases, implemented by a front-end and a back-end:

During the first phase (model compiler **front-end**), the textual representation of the high-level model is converted into an internal representation consisting of an **abstract syntax tree** (AST) and a **symbol table** (ST). For this conversion, a **lexical analyzer** (for splitting up the input text stream into individual tokens) and a **parser** (for matching the sequence of tokens against a set of grammatical rules) are adopted (Aho et al., 1986). Subsequently, the AST is further manipulated in order to perform **flattening** of the model's inheritance and composition hierarchies, as well as of the complex data types that the model uses. The resulting representation will consist of only one model with coalesced declaration and equation sections, in contrast to the original situation in which a top-level model was built upon base models through inheritance, and upon sub-models through composition. As an effect of the flattening of complex data types, the resulting model will only rely on scalar data types, which offers increased potential for code optimization.

The second phase (model compiler **back-end**) of the model compilation process is aimed at the generation of optimized executable code from the flattened model. As discussed in the previous section, C was chosen as a target language for executable models, for reasons of efficiency and flexibility. The code generated by the model compiler is then further processed by a regular C compiler and linked against a number of support libraries. In the case of Tornado, the resulting binary code is dynamically loaded into the virtual experiment executor.

In the scope of Tornado-I, the MSL language was designed and a full MSL model compiler (front-end and back-end) was developed that generates executable model code in the Tornado-I format. For Tornado-II, this model compiler was modified in order to generate Tornado-II executable models. Also, a model compiler back-end for the Modelica language was developed, which generates Tornado-II executable models. The Modelica model compiler back-end operates on the basis of flat Modelica models as generated by the OpenModelica model compiler front-end. The following discusses these Tornado model compiler front-ends and back-ends in further detail.

9.4.1 MSL: Introduction

In the following two sections, the most relevant components of the MSL 3.* model compiler (as depicted in Figure 9.4) are discussed. It should be noted that the initial version of the MSL 3.* model compiler was designed and implemented in 1997–1998 by Hans Vangheluwe and Bhama Sridharan (Vangheluwe, 2000). This effort is therefore not claimed as a result of the PhD work described in this dissertation, in contrast to several major improvements to this compiler with respect to performance and stability that were realized at a later stage. Also the generation of Tornado-II executable models and the replacement of LEDA by STL (to be discussed below) are claimed as results.

As mentioned above, the MSL model compiler is usually used in conjunction with a preprocessor. The compiler is implemented as a stand-alone program that was named *tmsl*.

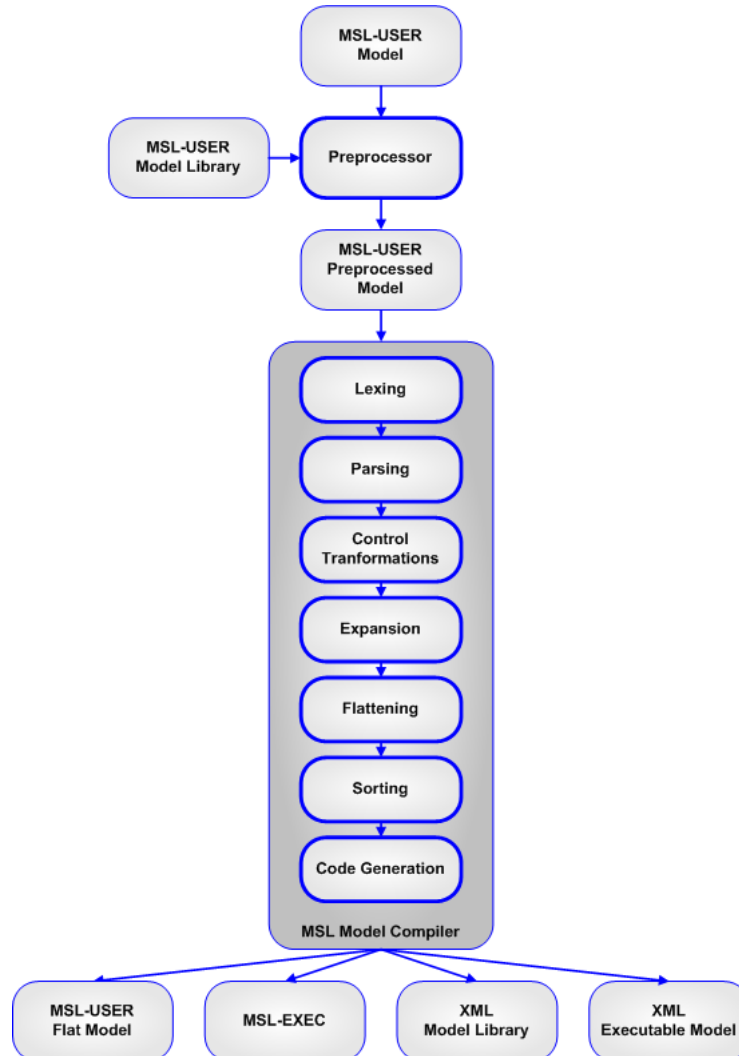


Figure 9.4: MSL Model Compiler

9.4.2 MSL: Front-end

The MSL model compiler front-end was originally developed in the scope of Tornado-I and later re-used for Tornado-II. Development was done in C++. The front-end only supports one input format, *i.e.*, MSL-USER. The process that is executed consists of the following stages:

- Lexical and grammatical analysis
- Control transformations
- Expansion
- Flattening
- Sorting

Lexical and Grammatical Analysis

The front-end respectively uses *flex* and *bison* for lexical and grammatical analysis of high-level model text input. *flex*¹¹ is an improved clone (developed by GNU) of the *lex* tool (Levine et al., 1995), which has been a part of most Unix systems since the 1970's. *lex* is a lexical analyzer generator, *i.e.*, it converts a high-level description of a number of tokens into source code for a lexical analyzer that breaks up input text into a sequence of these tokens. The high-level token description is actually a mixture of regular expressions (Aho et al., 1986) describing the tokens, and source code statements that need to be executed when these tokens are encountered in the input text stream. The lexical analyzer source code that is generated by *lex* and *flex* is C-based.

The *bison*¹² program is an improved clone (developed by GNU) of the *yacc* (Yet Another Compiler Compiler) tool (Levine et al., 1995), which has also been a part of most Unix systems since the 1970's. *yacc* is a parser generator, *i.e.*, it converts a high-level description of a grammar into source code for a grammatical analyzer that matches the token stream (that is generated by a *lex*-generated lexical analyzer) against a language grammar. The high-level grammar description is actually a mixture of production rules (Aho et al., 1986) describing the various constructs supported by the language, and source code statements that need to be executed when these rules are matched against the stream of tokens. The grammatical analyzer source code that is generated by *yacc* and *bison* is C-based.

The symbol table and abstract syntax tree that are built up during the parsing process are complex data structures that were originally implemented on the basis of LEDA¹³ (Library of Efficient Data Types and Algorithms) (Näher and Mehlhorn, 1990). The symbol table is basically a dictionary (*i.e.*, an associative array) that maps identifiers to records containing attribute-value pairs. The abstract syntax tree is a n-ary tree where each node contains a record that contains information on the type of the node and a number of type-dependent attribute-value pairs.

In the scope of Tornado-II, the implementations of ST and AST were substantially modified. The LEDA library was removed and STL was used instead. At the time of Tornado-I, STL could not be used because it was not yet sufficiently stable and efficient, hence LEDA had to be involved. Later however, STL was substantially improved, and as a result no more need existed to rely on LEDA, for all LEDA functionality that is used in the Tornado MSL compiler is also available in STL. An additional reason for moving away from LEDA is the fact that it became commercial in February 1st, 2001. Given the little additional benefit of the use of LEDA over STL, the purchase of license fees for LEDA could not be justified.

Control Transformations

The first transformation that is applied to the AST and ST that were built up during parsing is to handle MSL *control* constructs. As discussed before (*cf.* Section 9.2.1), *control* constructs are used in MSL in case a model parameter (which by definition cannot change during the course of a simulation) is to be modified by a controller signal. The goal of the control transformation process is to convert controlled parameters to input variables, after which the *control* construct can be converted into a regular *connect* construct.

Expansion

During the expansion processes, complex data structures (typically vectors and matrices) are converted into collections of scalar variables. Also, *FOREACH* and *SUMOVER* constructs are respectively con-

¹¹<http://www.gnu.org/software/flex>

¹²<http://www.gnu.org/software/bison>

¹³<http://www.algorithmic-solutions.com/enleda.htm>

verted into a set of scalar equations, and an equation that computes a sum over vector elements. After expansion, all symbol table entries and all equations are scalar.

Flattening

During flattening, the inheritance hierarchy and composition hierarchy of the model are resolved. Inheritance relationships are removed by recursively copying the description of base models into the derived model's description. The composition structure of a coupled model is removed by name unification (making local sub-model names globally unique, *i.e.*, by using the unique name of the sub-model as a prefix) and concatenation of sub-model equations. The concatenation of sub-model equations involves extension of the coupled model's set of equations with the sub-models' equations sets. *connect* constructs are at this stage converted to equations of the form $a.u = b.y$, where u is an input variable of sub-model a and y is an output variable of sub-model b .

Sorting

During the sorting process, the original (unsorted) set of equations is converted to a (sorted) sequence of assignments. Sorting is done on the basis of a dependency graph where each node represents a variable. Edges indicate which other variable values need to be available before a specific variable can be computed. This dependency graph can easily be constructed by walking through the flattened model's equation set. For each algebraic equation, edges are to be created going from the equation's LHS variable to each of the RHS variables. It is not necessary to involve differential equations in this process since these have derivatives as a LHS ($der(x)$). Derivatives are used by the integration solver to compute an approximated new value at the next time point. Since the new value only comes available at the next time point, it does not influence sorting for the current time point.

9.4.3 MSL: Back-end

The MSL model compiler's back-end is capable of generating information in four different formats:

- **MSL-USER**
- **MSL-EXEC**
- **XMLModelLib**: Model Library XML
- **XMLExec**: Executable Model XML

MSL-USER

When MSL-USER code generation is selected, an MSL representation of the flattened model is generated. This mode is especially useful for obtaining insight in the model compiler's front-end manipulations and for tracking potential problems.

MSL-EXEC

The MSL-EXEC code generation option is the most frequently used. In the scope of Tornado-I it was used to generate C++ code representing the flattened model's symbolic and computational information. In Tornado-II it generates C code (*.c) that represents the models computational information and an XML file (*.SymbModel.xml) that contains a description of the model's symbolic information as well as the relationship between this information and the computational information's data containers.

In order to be able to dynamically load executable model code into a virtual experiment executor it first needs to be compiled and linked with a regular C compiler/linker. Tornado provides for an entity named **model builder** that facilitates this process. The model builder runs the compilation and linkage process behind the scenes, with appropriate compiler and linker options, using the user's C compiler of choice. Resulting from this process is a DLL on Windows or a shared object on Linux, as is depicted in Figure 9.5 (left). In this Figure, *tbuild* is a command-line tool implementing a model builder, while *texec* is a command-line virtual experiment executor (refer to Chapter 11 for a description of the Tornado command-line suite).

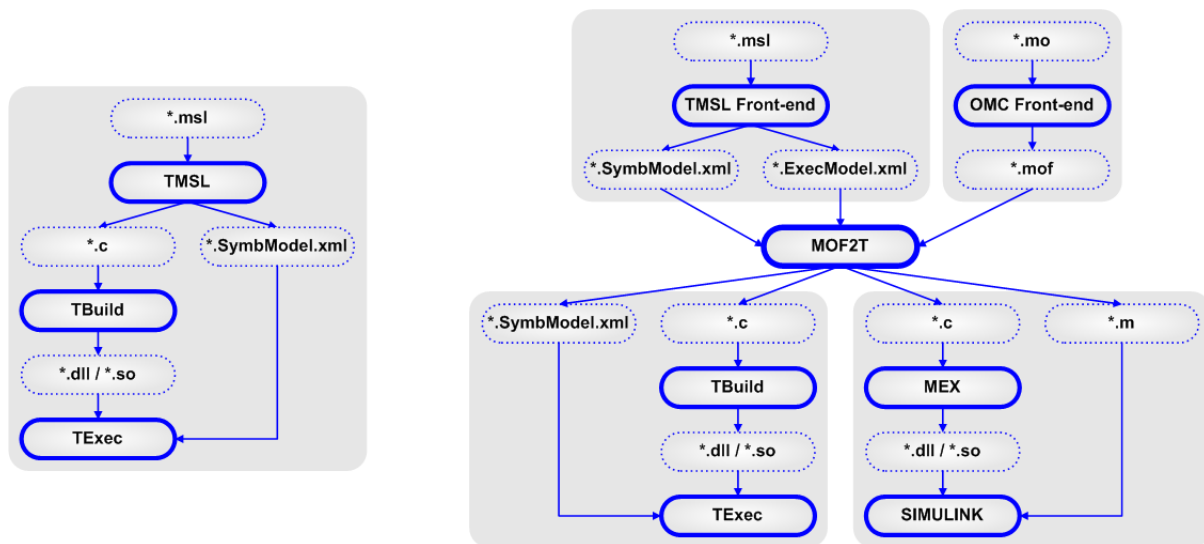


Figure 9.5: Relationship between *tmsl*, *omc* and *mof2t*

Model Library XML

In this mode, an XML representation of a model library is generated. The XML representation is not complete, in the sense that only information about types, models, sub-models and quantities is generated. Equations are not represented in this type of information. The Model Library XML representation can be regarded as a static, easily-queryable representation of the symbolic information of a model library. It can for instance be used by graphical layout editors to present collections of already existing models that a user can choose from to build new coupled models.

Executable Model XML

The last type of information that can be generated from the MSL model compiler is an XML representation of a flattened model's computational information (**.ExecModel.xml*). It is the same information as contained in **.c*, albeit expressed as XML and therefore more suitable for various types of post-processing.

9.4.4 Modelica: Front-end

Modelica is a language of substantial complexity. So far, no implementation has managed to entirely support the language as specified in the latest Modelica Specification¹⁴ document, published by the

¹⁴<http://www.modelica.org/documents/ModelicaSpec30.pdf>

Modelica Association. The implementation that has the longest track record with regard to the adoption of Modelica is Dynasim’s Dymola, but also this tool does not implement every facet of the Modelica Specification.

Although in the scope of Tornado-II it was soon realized that Modelica could fulfill many of the requirements stated in Chapter 4, the implementation of a comprehensive Modelica model compiler has not been attempted, given the time and effort that this would require. Moreover, one may wonder if the implementation of a comprehensive Modelica model compiler is at all desirable, given the substantial number of compilers for this language that are already available. Indeed, model compilers consist of a front-end and a back-end. Whereas the back-end is usually fairly specific as it is intended for the generation of a particular type of executable code, the front-end is quite general and could therefore potentially be re-used.

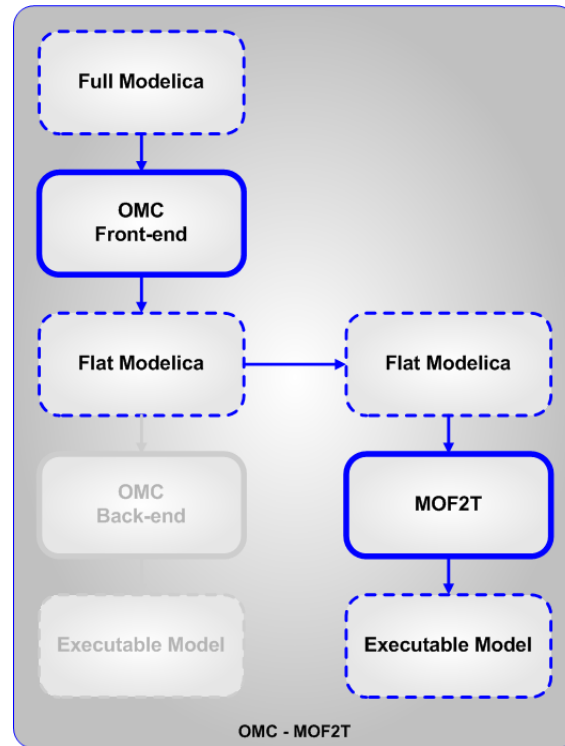
On the basis of several (including practical) considerations, it was decided to adopt the OpenModelica model compiler front-end in the scope of Tornado-II, and to complement it with a back-end that performs MSL-EXEC code generation. The OpenModelica compiler (*omc*) is an executable that is typically run from the command-line. By default (*i.e.*, when no additional command-line switches are used) *omc* will generate a representation of a flattened model in a language that is informally known as “flat Modelica”. It should be noted that at this point, there is no official definition of “flat Modelica” by the Modelica Association or any other party. We therefore consider flat Modelica to be the form of Modelica that is free of inheritance hierarchies, composition hierarchies and complex data structures that is generated by the OpenModelica compiler when code generation is not switched on. Actually, the generation of code for the OpenModelica simulator is only activated when a specific switch is applied to the *omc* command. This behavior is quite convenient with regard to the use of custom back-ends for code generation. These back-ends can be implemented as self-contained programs that simply read in the flat Modelica generated by *omc* and hence generate their own custom executable code. This is also the approach that was followed for the Tornado Modelica back-end, which was named *mof2t* as it converts flat Modelica (Modelica Flat) to executable code for Tornado.

Figure 9.6 illustrates the relationship between *omc* and *mof2t*. On the left, the typical *omc* situation is shown, whereas on the right it is shown that *mof2t* acts as a replacement for the original *omc* code generator.

9.4.5 Modelica: Back-end

mof2t currently only supports a subset of flat Modelica and is only capable of handling quantities of type *Real*. On the other hand, a number of constructs, attributes and functions are allowed by the input grammar that are inspired by MSL and are not available in the “original” flat Modelica. In this sense, *mof2t* can be regarded as a hybrid model compiler back-end of some sorts. The following are the most important items that are supported by *mof2t* and are not available through the *omc* back-end:

- **Final equations section:** Equations that are only to be evaluated at the end of a simulation can be grouped in a final equation section. This feature can only be used when writing flat Modelica models by hand, since it is not supported by *omc*.
- **Group and PDF attributes:** For quantities, attributes are available to specify the group to which the quantity belongs (to allow for easier browsing in applications), and the probability density function that is associated with the quantity (for documentation purposes and to be used as a default for some virtual experiments that rely on PDF’s). These attributes can also only be used when writing flat Modelica models by hand, since it is not supported by *omc*. We believe it might be useful to modify the implementation of *omc* in such a way that non-standard attributes (*i.e.*, attributes not standardized by the Modelica Specification) are simply copied as-is to the flat Modelica output, instead of discarded as is currently the case. We do not wish to advocate the

Figure 9.6: Relationship between *omc* and *mof2t*

extension of the Modelica Specification in order to add the group and PDF attributes to the set of standard attributes.

- **Delay function:** The *delay()* function allows for modelling DDE's. It is defined by the Modelica Specification, but is not implemented in *omc*. In *mof2t* it is available because of the importance of the ability to model DDE's in the scope of the water quality domain.
- **Previous function:** The *previous()* function returns the value of a quantity at the previous major simulation time point. This is a low-level function with a behavior that is integration solver dependent. It has been made available to provide for compatibility with MSL models that use this feature to break algebraic loops.
- **Reference function:** The *ref()* function returns a pointer to the variable to which it is applied. It is required to be able to pass references to vectors and matrices as arguments to functions. The reason this is necessary is because *mof2t* only supports functions with real-valued arguments.

The *mof2t* stand-alone executable is conceived in such a way that it can not only be used in conjunction with *omc*, but also in unison with *tmsl*, and as a separate tool that stands by itself. The topmost part of Figure 9.5 (right) shows that Modelica input files (*.mo) are processed by the *omc* front-end in order to generate flattened Modelica output (*.mof), which is then further processed by *mof2t*. Since *mof2t* has a number of features that are not present in the original *tmsl* back-end, it has also been made possible to use the *tmsl* front-end with *mof2t*. In this case, a representation of the *tmsl* AST (at least the part that is related to equations) is saved in XML format (*.ExecModel.xml). This computational model information is passed to *mof2t*, along with the symbolic model information (*.SymbModel.xml). So, *mof2t* supports two input grammars: a slightly modified version of flat Modelica, and an XML-based specification describing the computational and symbolic information of a flattened model.

As can be seen from the lower part of Figure 9.5 (right), *mof2t* supports two targets for code generation. For Tornado, C code can be generated and subsequently converted to a dynamically loadable library with *tbuild*. However, it is also possible to generate C code that expresses the flattened model as a so-called S-function for MATLAB/Simulink. In this case, the C code that is generated by *mof2t* is to be further processed by MATLAB's *mex* utility in order to create a dynamically loadable library that can be attached to a Simulink user-defined function block. The sequel will focus on the first type of code generation, although most of the principles that will be discussed are also applicable to the second type.

mof2t was developed in C++, using STL for complex datatypes and algorithms. For lexical and grammatical analysis, *flex* and *bison* were used respectively. Table 9.5 lists the types of nodes that were used to represent the AST. As can be seen from the table, all nodes have a fixed number of children (actually, many node types are binary). The advantage of the approach that was followed is that the implementation of node types and the algorithms that act upon them is straightforward. However, using this approach AST's quickly become complex as the model grows. Nodes with a variable number of children would be better in this respect. However, since at this stage of the development of *mof2t*, the easy implementation of algorithms is more important than the achievement of the utmost efficiency, the approach based on a fixed number of children was decided upon.

Table 9.6 gives an overview of the functions that are directly supported by *mof2t*. Other, external functions can also be used, but these have to be implemented by a library with C-linkage that is linked to the executable model code that is generated by *mof2t*. For each of the directly supported functions listed in the table, it is specified whether an implementation is provided through the standard C run-time library, through the MSL run-time library or through the generated executable model code itself. The MSL run-time library is a library that is linked by default to all executable code that is generated by *tmsl* and *mof2t*. Historically, it was specifically developed for *tmsl* (hence the name), but through a careful design of *mof2t* it has been made possible to re-use it in the scope of the latter as well.

Listing 9.3 contains a formal description of the version of flat Modelica that is supported by *mof2t*. The description is provided in Backus-Naur Form (BNF)¹⁵. The BackusNaur form is a meta-syntax used to express context-free grammars: that is, a formal way to describe formal languages. BNF is widely used as a notation for the grammars of computer programming languages, instruction sets and communication protocols, as well as a notation for representing parts of natural language grammars. There are many extensions of and variants on BNF, but in this case the original version is used.

Listing 9.3: BNF Description of the *mof2t* Flat Modelica Grammar

```

<fclass> ::= "fclass" <id> <desc> <declarations> <initial-equations-section>
           <equations-section> <final-equations-section> "end" <id> ";"

<declarations> ::= | <declaration> <declarations>

<declaration> ::= <prefix> <type> <id> <attribute-section> <expr-section> <desc> ";";

<attribute-section> ::= | "(" <attributes> ")"

<attributes> ::= <attribute>
                | <attribute> "," <attributes>

<attribute> ::= <id> "=" <expr>
                | <id> "=" <string>

<expr-section> ::= | "=" <expr>

<prefix> ::=
    | "constant"
    | "parameter"
    | "input"
    | "output"

```

¹⁵http://en.wikipedia.org/wiki/Backus-Naur_form

```

<type> ::= <id>

<initial_equations_section> ::= | "initial" "equation" <equations>

<final_equations_section> ::= | "final" "equation" <equations>

<equations_section> ::= | "equation" <equations>

<equations> ::= | <equation> <equations>

<equation> ::=
    <id> "=" <expr> ";"
    | "der" "(" <id> ")" "=" <expr> ";"

<expr> ::=
    <simple_expr>
    | "if" <expr> "then" <expr> <elseifs> "else" <expr>

<elseifs> ::= | <elseif> <elseifs>

<elseif> ::= "elseif" <expr> "then" <expr>

<simple_expr> ::= <logical_expr>

<logical_expr> ::=
    <logical_term>
    | <logical_term> "||" <logical_term>

<logical_term> ::=
    <logical_factor>
    | <logical_factor> "&&" <logical_factor>

<logical_factor> ::=
    <relation>
    | "!" <relation>

<relation> ::=
    <arithmetic_expr>
    | <arithmetic_expr> "<" <arithmetic_expr>
    | <arithmetic_expr> "<=" <arithmetic_expr>
    | <arithmetic_expr> ">" <arithmetic_expr>
    | <arithmetic_expr> ">=" <arithmetic_expr>
    | <arithmetic_expr> "==" <arithmetic_expr>
    | <arithmetic_expr> "!=" <arithmetic_expr>

<arithmetic_expr> ::= <terms>

<terms> ::=
    <term>
    | <terms> "+" <term>
    | <terms> "-" <term>

<term> ::= <factors>

<factors> ::=
    <factor>
    | <factor> "*" <factors>
    | <factor> "/" <factors>

<factor> ::=
    <primary>
    | <primary> "^" <primary>
    | "+" <primary>
    | "-" <primary>

<primary> :
    <number>
    | id
    | <function_call>
    | "(" <expr> ")"

<function_call> ::= <id> "(" <args> ")"

<args> ::=
    <expr>
    | <string>
    | <expr> "," <args>
    | <string> "," <args>

<desc> ::= | <string>

```

mof2t is capable of generating output in a variety of formats. Some of these formats pertain to executable model code, others are intended for debugging or to obtain insight into the model structure and/or complexity. The formats that are currently supported, are the following:

- **MOF**: Generates flat Modelica. In case flat Modelica was used as input for the model compiler, this mode will allow for verifying the correctness of the model compiler. In case an XML representation of a flat model, generated by *tmsl* was used as input, the mode allows for a conversion of flat MSL to flat Modelica.
- **MO**: Generates a model description that semantically describes a flattened model, however using the syntax rules of proper Modelica.
- **MSL-USER**: Generates an MSL-USER description of the flattened model. This mode can therefore be used to convert flat Modelica to flat MSL-USER.
- **MSL-EXEC**: Generates executable code in Tornado-II format consisting of computational information (C) and symbolic information (XML)
- **SFunction**: Generates executable code formatted as an S-function for use with MATLAB/Simulink
- **XML**: Generates XML descriptions containing computational information and symbolic information for the flattened model.
- **MathML**: Generates a MathML representation of the flattened model's equations. MathML¹⁶ is an XML-based mathematical markup language.
- **Complexity**: Generates an unformatted report on the complexity of the flattened model. As a metric for the complexity, the number of instances of each type of AST node is used.
- **GraphViz**: Generates a description of the AST in the GraphViz' dot language. GraphViz¹⁷ is open source graph visualization software.

Constant Folding

mof2t applies a technique known as constant folding to equations. This technique allows for computing expressions that are based on constant items at compile-time, rather than at run-time. Evidently, this is beneficial for run-time performance. Constant expressions are made up of literals and quantities that are of variability 0 (*i.e.*, constants and parameters/variables for which the Fixed attribute was enabled), *e.g.*, $\text{sqrt}(5 * \pi + 1.234)$. The effect of constant folding is very model-dependent, but is usually fairly limited. Actually, it can be assumed that the C compiler that is used to compile the executable model code, will perform constant folding of its own, hereby partly alleviated the need to perform model compiler based constant folding.

Equiv Substitution

As a result of the coupling of sub-models, flattened models will typically contain many equiv's, *i.e.*, equations of type $u = y$, where u is an input variable and y is an output variable. Although these equations are trivial, their sheer number can lead to a substantial performance degradation, since they are to be evaluated at each integration time point (the number of integration time points can easily be as high as 10^6 or more). Through symbolic substitution of u by y in all equations, the equation $u = y$ can be dropped,

¹⁶<http://www.w3.org/Math>

¹⁷<http://www.graphviz.org>

and superfluous evaluations can be avoided. Of course, u can only be removed from the computational information of an executable model, in the symbolic information it must be retained, for a user who wants to use this variable to perform analyses must be able to do so. This is accomplished by modifying the linkage information that is part of a model's symbolic information. Instead of u and y being linked to different data containers, equiv substitution will link them to one and the same data container.

The performance gain that can be accomplished by equiv substitution is determined by two factors: the number of equiv's versus the total number of equations, and the complexity of the non-equiv equations. In case the number of equiv's is high, or the complexity of non-equiv equations is low, equiv substitution has been shown to lead to a performance increase of 5 to 30%. In case there are only few equiv equations, or in case the non-equiv equations are of high complexity (*e.g.*, containing trigonometric functions or power operations), the effect of equiv substitution will only be marginal. In fact, experiments have shown that in these cases equiv substitution can even lead to a performance degradation. The reason for this is that the C compiler (that is invoked by *tbuild* on the code generated by *mof2t*) performs a large number of optimizations itself. In some cases, equiv substitution will restrict the number of optimizations that the C compiler can perform, resulting in a slight degradation of performance. For this reason, equiv substitution has been made optional in *mof2t*. It should be noted that irrespective of the effect on performance, equiv substitution is also useful to reduce the size of executable model code. Since C compilers have internal limits with regard of the size of the code they are able to compile, equiv substitution may help to resolve problems in this respect.

Lifting

In MSL and Modelica users have the ability to distinguish initial equations from state equations (by placing them either in a so-called initial or state section). In MSL, final equations can also be specified. At first glance, this information is useful for the model compiler to place flattened equations in the appropriate event routine (*cf.* Section 9.3.2). However, it is not sufficient. First of all, MSL nor Modelica allow for equations to be labeled as output equations. Reason for this is that finding out which equations are output equations is non-trivial and error-prone, it is therefore better to leave it to the model compiler. Secondly, there might be equations that do not vary during the course of a simulation, but nonetheless have not been identified by the user as initial equations. In order for these two issues to be overcome, there are two types of lifting (*i.e.*, taking an equation out of one section and moving it to another) that have been implemented:

- **Lifting of initial equations:** By inspecting the right-hand sides of equations in the state section, the variability of equations can be determined. Equations of variability 0 only rely on constants and will always yield the same result. They can therefore be computed at compile-time and removed from the system. Equations of variability 1 rely on constants and parameters and will remain constant during the course of a simulation. They can therefore be lifted from the state section and moved to the initial section. The remaining equations vary during the course of a simulation and are therefore of variability 2.
- **Lifting of output equations:** When investigating the relationships between equations remaining in the state section after lifting of initial equations, it can be determined which equations actually contribute to the state of the system and which do not. In order to do this, a dependency graph can be built in which each node represents an equation. It then suffices to walk through this graph starting at each node representing a differential equation. Each node visited in this way is marked. The nodes that remain unmarked pertain to output equations. These equations can therefore be lifted from the state section and moved to the output section.

In practice, the number of equations that can be identified as initial equations by the model compiler (next to the ones that were already identified as such by the user) is often limited. Lifting of initial

equations will therefore usually not yield a significant performance gain. However, the number of output equations is mostly significant. In addition, these equations tend to be relatively complex. As a result, the fact that these equations only have to be computed at major time points (instead of at every minor time point if they were to remain in the state section) yields significant performance improvements, *e.g.*, 20%. As can be seen from Figure 9.2, *ComputeOutput* is only executed at major time points, *i.e.*, less frequently than the *ComputeState* section, which is executed at every minor time point. The more intermediate, *i.e.*, minor, time points an integration algorithm uses, the more the integration process will benefit from lifting of output equations. For example, for first and second order algorithms such as Euler and Heun the benefit will be non-existent to marginal. For higher-order algorithms such as Runge-Kutta 4, it will be more noticeable.

Bounds Checking

As a result of the various manipulations performed by a model compiler (flattening, lifting, ...), the resulting executable code is often not recognizable anymore to the user. Consequently, care must be taken to detect potential problems at an early stage, and emit warnings or error messages that can easily be understood by the user. One way to improve model safety (*e.g.*, divergence due to numerical problems such as inappropriate integrator tolerance settings) is to check each variable against its lower bound and upper bound during the course of a simulation (both MSL and Modelica allow for lower bounds and upper bounds to be specified as meta-information attributes). A possible way to tackle this issue is by adding a generic bounds checking module to the simulator engine. This approach however has proven to be prohibitively slow (up to 50% performance degradation), since it requires extensive run-time querying of model meta-data. Another approach is to have the model compiler generate code specifically for the bounds that are to be checked for the model at hand. This approach allows for several optimizations since lower bounds set to $-\infty$ and upper bounds set to $+\infty$ do not need to be checked. Hence, for these variables no code needs to be generated. The event routine that performs bounds checking was named *CheckBounds* and is called at every major integration time point, as can be seen from Figure 9.2.

There are several possible ways to handle a bounds violation. One possibility is simply to halt execution, *i.e.*, stop the simulation and emit an error message. Another possibility is to continue the simulation by setting the variable to the value of the bound that was exceeded. In this way the assumed biological behavior can be mimicked. A third alternative is to emit a warning message and continue the simulation with the variable value in exceedance. In case of the latter, a flag is to be maintained that indicates the bound's violation state, in order to avoid the same message to be issued multiple times. Figure 9.7 depicts a state machine that describes *mof2t*'s implementation of this mechanism.

Tests have shown that automatically generated bounds checking code typically only incurs a performance degradation of about 5%, which is 10 times less than performing bounds checking through the simulation engine.

Code Instrumentation

Mathematical expressions frequently contain operations that are only defined within a certain domain with respect to their arguments. A division for instance should not have zero as a denominator. A power operation such as x^y is not defined when x is negative and y is not an integral value (this would result in a complex number).

The problem with complex models is that in case a division-by-zero or other domain error occurs, it is very difficult to track down where this problem originates. At best, execution will be halted and one will receive an error message indicating that a domain error has occurred at a certain simulation time point. What one would like to see in these cases however, is a human-readable representation of the expression that has triggered the run-time error. This can be accomplished by performing code instrumentation,

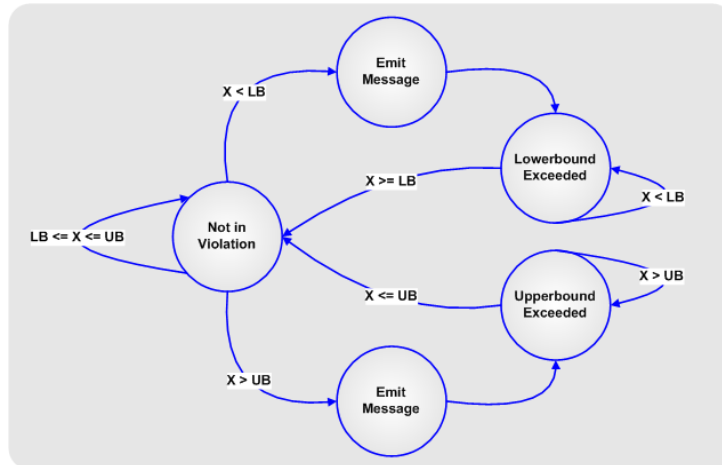


Figure 9.7: Bounds Violation Handling Mechanism implemented in *mof2t*

i.e., replacing potentially hazardous operations by macro's (preprocessor defines). In these macro's, the arguments that might lead to problems are first checked against their domain of validity. In the absence of problems, the original operation is performed. In case of a problem however, a meaningful message is issued that is constructed on the basis of a human-readable representation of the expression in which the operation occurs. This representation (or rather a reference to it) is passed as an additional argument to the macro.

Table 9.7 gives an overview of a number of operations that have limited domains of validity for their arguments. The table also lists the macro's that are generated by the model compiler as a replacement. Finally, the table shows the implementation of the various macro's. Since the executable model code is generated by the model compiler from its internal AST, it is no problem also to generate the necessary human-readable expression representations. The reason why references (ID's) are passed to the macro's (rather than the expression strings themselves) is related to the compilation process of the generated C code. With several C compilers, the compilation process is severely slowed down when string literals occur in function calls. It is therefore better to store all strings in a lookup table and pass a reference to a location in this table to each function.

Symbolic Differentiation

The computation of derivatives of equation systems plays an important role in many numerical procedures. Several techniques for computing these derivatives exist. In (Tolsma and Barton, 1998) an overview is given of the ways derivatives can be evaluated: hand-coding, finite difference approximations, Reverse Polish Notation (RPN) evaluation, symbolic differentiation and automatic differentiation. In this article it is concluded that in an interpretive environment, the automatic differentiation approach has many advantages over the other techniques. Interpretive environments exist in most tools that are renown for formula management, such as REDUCE (Hearn, 1987), MACSYMA (Pavelle and Wang, 1985) and Maple (Hutton, 1995). However, in the scope of Tornado executable model code is being generated by a model compiler. Symbolic differentiation performed by the model compiler and resulting in an executable representation of the Jacobian of the equation system is therefore a compelling option.

Since equations are represented as a rooted tree in model compilers such as *mof2t*, symbolic differentiation can be implemented by traversing the tree and applying the simple rules of differentiation recursively. In *mof2t* this technique has recently been implemented in an experimental procedure. The procedure has thus far only been tested on simple equation systems and further improvement is required

in order for it to be successfully applicable to larger equation systems. One of the remaining issues is the fact that simplification of equations (reduction to a canonical form) is not yet available. In fact, this issue is not only relevant in the scope of symbolic differentiation, but also for other procedures in *mof2t*.

Results

Table 9.8 contains various results that were obtained by applying the above-mentioned techniques to a number of cases. A description of these cases can be found in Chapter 2. All models were implemented in MSL and subsequently processed by the *tmsl* front-end and the *mof2t* back-end. Tests were run on the Windows platform using a machine equipped with an INTEL Pentium M 1600 MHz processor and 512 Mb of RAM. As a C compiler, Microsoft Visual C++ 7.1 was used. Below is additional information on the items listed in the table:

- Items **1–7** provide information on the simulation experiment’s integration solver settings, simulation time horizon and input provider and output acceptor settings. Each of these have an effect on the speed of simulation, and are therefore given in the table as background information (they do not influence the model compilation process as such, nor the effects of it).
- Items **8–11** provide insight in the complexity of the model. Noteworthy are the fact that for large models, the model compiler’s AST becomes a very complex data structure. Also, *mof2t* is able to detect unused variables and parameters. These are reported as warnings, and can optionally be removed. Removal of unused quantities will reduce the size of the executable model, but will not have any effect on the speed of execution.
- Items **12–15** and **16–19** give the number of equations in the initial, state, output and final sections, respectively at the beginning and at the end of *mof2t*’s processing. In general, only few additional equations can be lifted from the state to the initial section. However, the number of equations that can be lifted to the output section is mostly substantial. In each case, the following relations hold:

$$\begin{aligned} \#Initial_{start} + \#State_{start} - \#Equiv &= \#Initial_{end} + \#State_{end} + \#Output_{end} \\ \#Output_{start} &= 0 \\ \#Final_{start} &= \#Final_{end} \end{aligned} \tag{9.1}$$

- Items **20–21** provide some information on the size of the executable model code after C compilation. The label “unsafe” refers to the fact that these figures pertain to executable models for which no bounds checking nor code instrumentation was activated. Activation leads to larger binary models.
- Items **22–24** give an indication of the duration of the compilation process itself. The *-p* flag that is applied to *tbuild* indicates the C compiler that is to be used. The *-n* flag switches off optimization in the C compiler (*i.e.*, results in the use of */O1* instead of */O2*).
- Items **25–27** respectively give the simulation time of non-optimized code (with constant folding, equiv substitution and lifting disabled), the simulation time of optimized code (with constant folding, equiv substitution and lifting enabled), and the speedup of the optimized code versus the non-optimized code.
- Items **28–30** respectively give the simulation time of optimized code with bounds checking enabled, the simulation time of optimized code with bounds checking *and* code instrumentation enabled, and the speedup of code with both safety measures enabled versus non-optimized code without safety measures. It can be seen that “safe” optimized code is slower than code without any

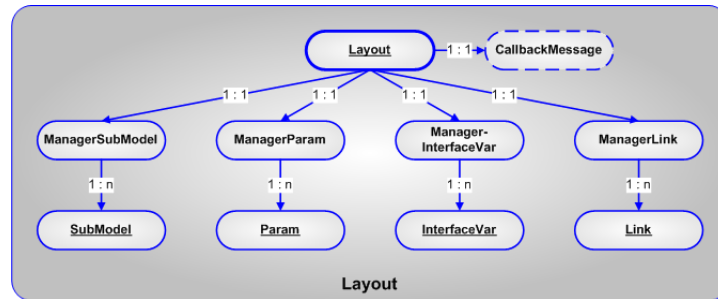


Figure 9.8: Layout ER Diagram

mof2t processing, but the slowdown is compensated to some extent by the types of optimization that were introduced. The overall slowdown therefore remains limited to a maximum of approximately 25%.

- Finally, items **31–32** provide a comparison with the simulation speed of the same cases implemented in WEST-3, a simulation tool that is currently commercially available. The WEST-3 MSL model compiler does not offer constant folding, equiv substitution, lifting, bounds checking nor code instrumentation. However, bounds checking is available through the WEST-3 simulation engine (hence the reference to “partially unsafe unoptimized code” in the table). In all cases, *mof2t*-optimized code with both safety measures switched on and simulated through the Tornado engine, is substantially faster than the same model simulated in a less safe manner in WEST-3. For large models, the speedup of Tornado is more pronounced than for small models. Higher scalability can therefore be assumed.

9.5 Layouts

In Tornado, a Layout is an entity that represents a coupled model’s connection graph. Tornado’s responsibility is only to maintain internal and XML-based persistent representations of layouts, graphical renditions are left up to applications that are built on top of the kernel. A coupled model consists of a number of sub-models (*SubModel*) and a number of links (*Link*) that connect sub-model interface variables. A coupled model has parameters (*Param*) and interface variables (*InterfaceVar*) of its own. Connections between coupled model parameters and interface variables and sub-model parameters and interfaces variables are also represented as links. Sub-models, links, parameters and interface variables are entities in their own right, *i.e.*, they support the dynamic property management mechanism.

An ER diagram of the internal representation of Layouts is in Figure 9.8. An XML schema definition of the persistent representation of Layouts can be found in Figure 9.9.

9.6 ModelLib: Model Library

In Tornado, a ModelLib is an entity that contains a static representation of the symbolic information contained in a library of high-level models. A ModelLib is a collection of models (*Model*), each containing collections of parameters (*Param*), interface variables (*InterfaceVar*), state variables (*StateVar*) and sub-models (*SubModel*). Models, parameters, interface variables, state variables and sub-models are entities and thus support the property mechanism. A ModelLib’s persistent representation is XML-based.

An ER diagram of the internal representation of ModelLibs is in Figure 9.10. An XML schema definition of the persistent representation of ModelLibs can be found in Figure 9.11.

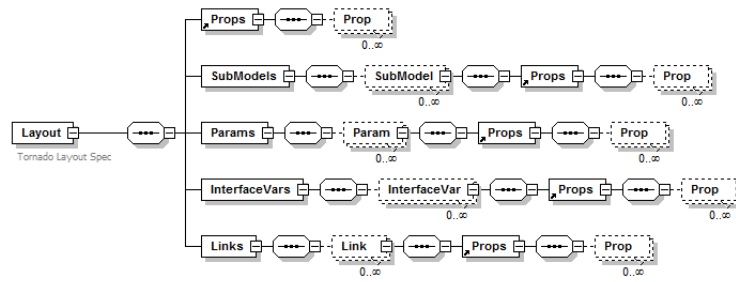


Figure 9.9: Layout XML Schema Definition

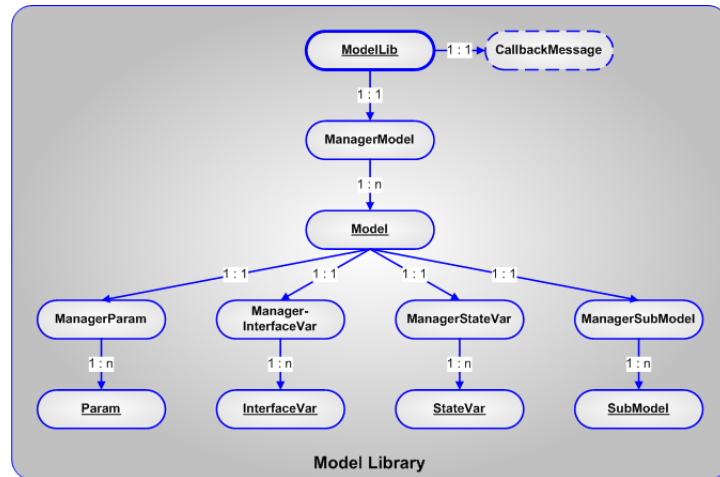


Figure 9.10: ModelLib ER Diagram

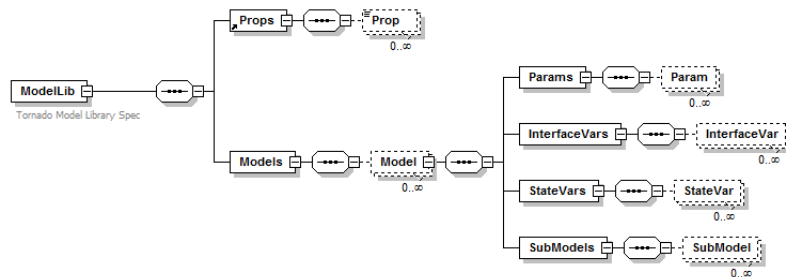


Figure 9.11: ModelLib XML Schema Definition

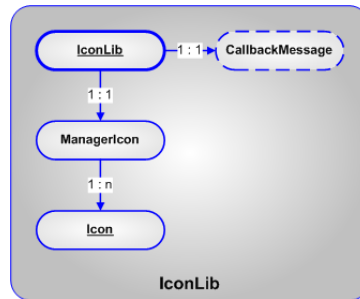


Figure 9.12: IconLib ER Diagram

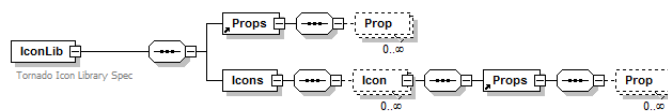


Figure 9.13: IconLib XML Schema Definition

A ModelLib representation is static in the sense that it can only be queried and not modified. Normally, a ModelLib representation is generated from a model compiler and is subsequently typically used by applications for querying during layout editing operations.

9.7 IconLib: Icon Library

In Tornado, an IconLib entity is a collection of icons that can be used to represent models in a graphical rendition of a coupled model's layout. As for layouts, Tornado only manages the internal and XML-based persistent representation of IconLib's. Graphical renditions of the icons themselves are left up to encapsulating applications.

An ER diagram of the internal representation of IconLibs is in Figure 9.12. An XML schema definition of the persistent representation of IconLibs can be found in Figure 9.13.

Table 9.5: *mof2t* AST Node Types

Node Type	Description	Children
EMPTY	Empty	
FCLASS	Flat class	Name [ID], Description [STRING], Declarations [SEQ_DECLARATIONS], Initial equations [SEQ_EQUATIONS], State equations [SEQ_EQUATION], Output equations [SEQ_EQUATION], Final equations [SEQ_EQUATION]
SEQ_DECLARATION	Sequence of declarations	Declaration [DECLARATION], Declarations [SEQ_DECLARATION]
SEQ_ATTRIBUTE	Sequence of attributes	Attribute [ATTRIBUTE], Attributes [SEQ_ATTRIBUTE]
SEQ_EQUATION	Sequence of equations	Equation [EQUATION], Equations [SEQ_EQUATION]
SEQ_ARG	Sequence of arguments	Argument [ARG], Arguments [SEQ_ARG]
SEQ_ELSEIF	Sequence of ElseIf's	ElseIf [ELSEIF], ElseIf's [SEQ_ELSEIF]
DECLARATION	Declaration	Prefix [ID], Type [ID], Name [ID], Attributes [SEQ_ATTRIBUTE], Description [STRING], Expression [EXPR]
ATTRIBUTE	Attribute	Name [STRING], Expression [EXPR]
EQUATION	Equation	IsDiffEq [BOOLEAN], LHS [ID], RHS [EXPR]
IF	If	Condition [EXPR], If-part [EXPR], ElseIf's [SEQ_ELSEIF], Else-part [EXPR]
ELSEIF	ElseIf	Condition [EXPR], Then-part [EXPR]
OR	Or	Arg1 [EXPR], Arg2 [EXPR]
AND	And	Arg1 [EXPR], Arg2 [EXPR]
NOT	Not	Arg [EXPR]
LT	Less than	Arg1 [EXPR], Arg2 [EXPR]
LE	Less than or equal	Arg1 [EXPR], Arg2 [EXPR]
GT	Larger than	Arg1 [EXPR], Arg2 [EXPR]
GE	Larger than or equal	Arg1 [EXPR], Arg2 [EXPR]
EQ	Equal to	Arg1 [EXPR], Arg2 [EXPR]
NEQ	Not equal to	Arg1 [EXPR], Arg2 [EXPR]
UPLUS	Unary plus	Arg [EXPR]
UMINUS	Unary minus	Arg [EXPR]
PLUS	Plus	Arg1 [EXPR], Arg2 [EXPR]
MINUS	Minus	Arg1 [EXPR], Arg2 [EXPR]
MUL	Multiply	Arg1 [EXPR], Arg2 [EXPR]
DIV	Divide	Arg1 [EXPR], Arg2 [EXPR]
POW	Power	Arg1 [EXPR], Arg2 [EXPR]
FUNC	Function	Name [ID], Args [SEQ_ARG]
NUMBER	Number	
BOOLEAN	Boolean	
STRING	String	

Table 9.6: *mof2t* Functions

Function	Description	Implemented through
abs(x)	Returns absolute value of x	C run-time library
acos(x)	Returns arc cosine of x	C run-time library
assert(x, Message)	If x is false, emit message and stop model evaluation	Code generation
asin(x)	Returns arc sine of x	C run-time library
atan(x)	Returns arc tangent x	C run-time library
atan2(x, y)	Returns arc tangent of x / y; the signs of both arguments are used to determine the quadrant of the result	C run-time library
ceil(x)	Returns smallest integral value not less than x	C run-time library
cos(x)	Returns cosine of x	C run-time library
cosh(x)	Returns hyperbolic cosine of x	C run-time library
der(x)	Derivative of x	Code generation
delay(x, Delta [, MaxDelta])	Returns x(t - Delta); the MaxDelta argument can be used to specify the maximum size of the time window	MSL run-time library
exp(x)	Returns base-e exponential of x	C run-time library
fabs(x)	Returns absolute value of x	C run-time library
floor(x)	Returns largest integral value not larger than x	C run-time library
max(x, y)	Returns maximum of x and y	MSL run-time library
min(x, y)	Returns minimum of x and y	MSL run-time library
log(x)	Returns natural logarithmic of x	C run-time library
log10(x)	Returns base-10 logarithmic of x	C run-time library
pow(x, y)	Returns x to the power of y	C run-time library
previous(x)	Returns value of x at previous major integration time point	Code generation
ref(x)	Returns pointer to x	Code generation
reinit(x, y)	Reinitialize x with y	Code generation
sign(x)	Returns +1 if x > 0; -1 if x < 0; and 0 otherwise	MSL run-time library
sin(x)	Returns sine of x	C run-time library
sinh(x)	Returns hyperbolic sine of x	C run-time library
sqrt(x)	Returns square root of x	C run-time library
tan(x)	Returns tangent of x	C run-time library
tanh(x)	Returns hyperbolic tangent of x	C run-time library
terminate(Message)	Prints message and stops model evaluation	MSL run-time library

Table 9.7: Code Instrumentations

Original	Replacement	Implementation of Replacement
x/y	<code>_DIV_(x, y, StringID)</code>	<code>y == 0 ? DivisionByZero(StringID) : x/y</code>
acos(x)	<code>_ACOS_(x, StringID)</code>	<code>(x < -1) (x > 1) ? DomainError(StringID) : acos(x)</code>
asin(x)	<code>_ASIN_(x, StringID)</code>	<code>(x < -1) (x > 1) ? DomainError(StringID) : asin(x)</code>
log(x)	<code>_LOG_(x, StringID)</code>	<code>x <= 0 ? DomainError(StringID) : log(x)</code>
log10(x)	<code>_LOG10_(x, StringID)</code>	<code>x <= 0 ? DomainError(StringID) : log10(x)</code>
pow(x, y)	<code>_POW_(x, y, StringID)</code>	<code>(x < 0) && ((y - (int)y) != 0) ? DomainError(StringID) : pow(x, y)</code>
sqrt(x)	<code>_SQRT_(x, StringID)</code>	<code>x < 0 ? DomainError(StringID) : sqrt(x)</code>

Table 9.8: Results of the Application of *mf2t* to Various WWTP Models

	ASU	BSM1_OL	BSM1_CL	Orbal	Galindo_OL	Galindo_CL
1 Integration solver	RK4	RK45	RK45	RK45	RK45	RK45
2 Stepsize / tolerance	1E-04	1E-06	1E-06	1E-06	1E-05	1E-05
3 Simulation time horizon	7d	28d	28d	25d	400d	400d
4 Output file communication interval	0.01d	15min	15min	0.01d	15min	15min
5 Number of quantities sent to output file	28	43	45	23	27	28
6 Input interpolation	No	Yes	Yes	Yes	No	No
7 Output interpolation	No	No	No	No	No	No
8 Number of nodes in AST	38,883	64,328	64,491	241,955	250,448	262,217
9 Number of unused quantities	118	129	155	243	278	373
10 Number of equiv's	172	219	253	623	489	614
11 Number of differential equations	30	178	110	250	270	275
12 Number of equations in initial section at start	148	170	175	2,433	1,410	1,405
13 Number of equations in state section at start	722	1,289	1,331	3,551	3,600	3,953
14 Number of equations in output section at start	0	0	0	0	0	0
15 Number of equations in final section at start	0	0	0	0	0	0
16 Number of equations in initial section at end	149	171	176	2,434	1,553	1,408
17 Number of equations in state section at end	418	929	943	2,137	2,630	2,998
18 Number of equations in output section at end	131	140	134	790	338	338
19 Number of equations in final section at end	0	0	0	0	0	0
20 Size of executable model DLI (unsafe optimized code)	36 KB	72 KB	92 KB	276 KB	248 KB	252 KB
21 Size of symbolic model XML (unsafe optimized code)	323 KB	551 KB	560 KB	2,604 KB	2,315 KB	2,402 KB
22 <i>mf2t</i> (unsafe optimized code)	1 s	3 s	3 s	110 s	75 s	80 s
23 <i>tbuild</i> -p win32-msvc7.1 (unsafe optimized code)	1 s	2 s	2 s	49 s	34 s	35 s
24 <i>tbuild</i> -n -p win32-msvc7.1 (unsafe optimized code)	1 s	1 s	1 s	1 s	1 s	1 s
25 Simulation time of non-optimized code	5 s	15 s	48 s	110 s	315 s	465 s
26 Simulation time of optimized code	4.5 s	14 s	43 s	86 s	220 s	426 s
27 Speedup of unsafe optimized code vs. unsafe non-optimized code	+10%	+7%	+10%	+22%	+30%	+8%
28 Simulation time of optimized code with bounds checking	5 s	14 s	45 s	94 s	235 s	454 s
29 Simulation time of optimized code with bounds checking and instrumentation	5 s	16 s	48 s	106 s	296 s	518 s
30 Speedup of safe optimized code vs. unsafe non-optimized code	-0%	-14%	-7%	-13%	-26%	-14%
31 Simulation time of partially unsafe unoptimized WEST-3 code	8.5 s	40 s	112 s	687 s	1,531 s	2,261 s
32 Speedup of safe optimized Tomado code vs. partially unsafe unoptimized WEST-3 code	+41%	+60%	+57%	+85%	+81%	+77%

Part III

Deployment

10

Application Programming Interfaces

10.1 Introduction

Essential to the ability to deploy a kernel such as Tornado in a wide variety of contexts, is the availability of an elaborate and extensible set of Application Programming Interfaces (API's). These should allow for Tornado to be adopted on a large number of development platforms, hopefully including the application programmer's platform of choice.

Figure 10.1 gives an overview of the API's that are currently available for Tornado. The figure consists of three levels: at the bottom level kernel components are shown, the middle level shows API components, and the top level shows application components. The API's that are in the figure are the following:

- **C++ API:** Tornado C++ API based on an import library and a dynamically-loadable library
- **C API:** Tornado C API based on an import library and a dynamically-loadable library
- **.NET API:** Tornado (generic) .NET API
- **OpenMI API:** Tornado OpenMI .NET API
- **MEX API:** Tornado MEX wrapper for use with MATLAB/Simulink
- **JNI API:** Tornado wrapper for use with Java through JNI technology

All of these API's are situated on top of the native C++ API of the Tornado kernel. This native API is based on a large number of static libraries and header files and is directly used by a limited set of applications that are closely related to Tornado. In general however, application developers should use the alternative C++ API, since it is based on an import library and a dynamically-loadable library. Advantages of the latter are the fact that binary code can be shared amongst applications and that the implementation of dynamically-loadable libraries can be changed without recompiling the applications (as long as the interface remains the same).

In the remainder of this chapter, a distinction is made between two types of API's:

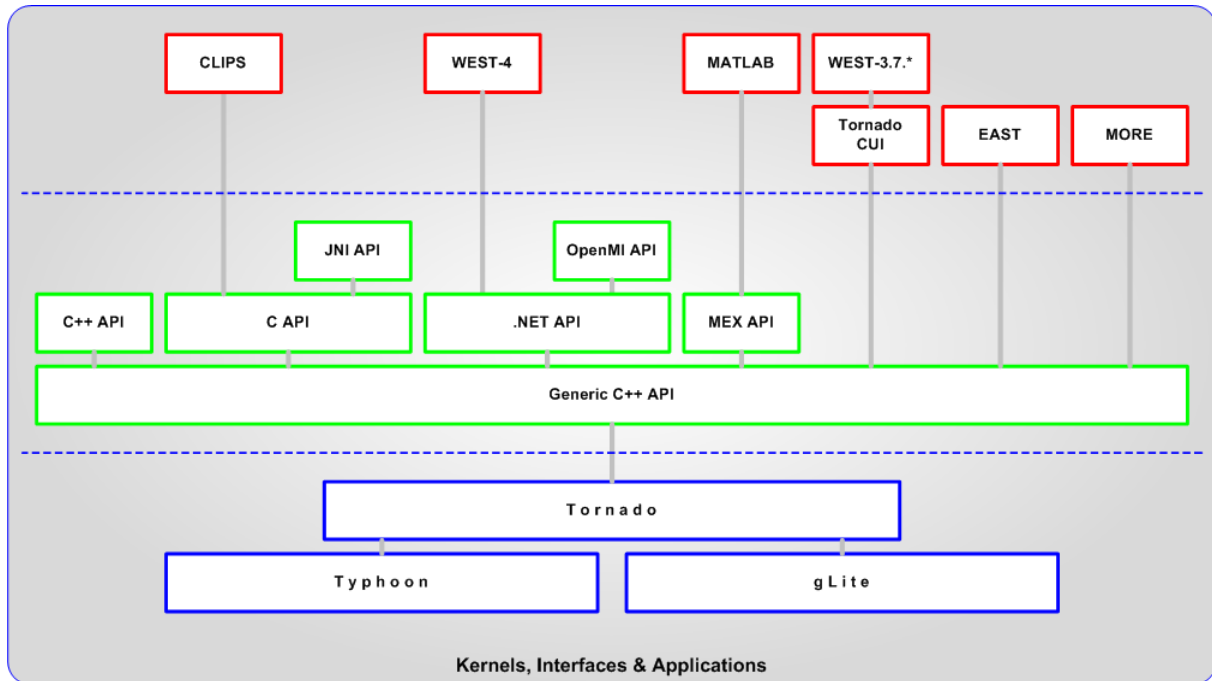


Figure 10.1: Tornado Application Programming Interfaces

- **Comprehensive API's:** Allow for all of the Tornado functionality to be used from application programs.
- **Restricted API's:** Only allow for a restricted subset of Tornado functionality to be used from application programs. Typically, the restricted subset is limited to loading, initializing and executing virtual experiments.

10.2 Comprehensive API's

10.2.1 C++

The first of two comprehensive API's for Tornado is the C++ API. In order to use this API from application programs, the following files are required:

- *TornadoCPP.lib*: Import library
- *TornadoCPP.dll* (Windows) or *TornadoCPP.so* (Linux): Dynamically-loadable library
- *TornadoCPP.h*: TornadoCPP header file
- *I*.h*: Set of header files with declarations of abstract interface classes

The C++ API provides access to Tornado through an API consisting of merely three functions, as is shown in Listing 10.1. The *TInitialize* function creates an instance of the Tornado kernel and initializes it with a message callback, as well as with the name of an XML file that contains settings for the main entity, and a boolean that indicates whether numerical solver plug-ins are to be loaded. The *TGet* function returns a pointer to the kernel that was created through *TInitialize*. Finally, the *TFinalize* function disposes of the kernel that was created through *TInitialize*. An application program will first call *TInitialize*

and will then retrieve a pointer to the Tornado main entity through *TGet*. On the basis of this pointer, the application program will perform its processing. Finally, it will call *TFinalize* to perform clean-up. As mentioned above, one needs to dispose of several header files containing abstract interface declarations (*I*.h*) in order to be able to use the Tornado C++ API. These declarations describe *Tornado::ITornado* and all its dependencies. Due to the extensiveness of this description, it will not be given here.

Listing 10.1: Tornado C++ API (Excerpt)

```
// Instantiate and initialize Tornado kernel
void TInitialize(Common::ICallbackMessage* pCallbackMessage ,
                const std::wstring& MainFileName ,
                bool LoadPlugins);

// Return pointer to Tornado main entity
Tornado::ITornado* TGet();

// Dispose of kernel
void TFinalize();
```

Listing 10.2 shows an example program that loads and executes a virtual experiment on the basis of the Tornado C++ API. The program implements all required callbacks through one implementation class (*CCallbacks*, which is derived from 5 callback interfaces: *ICallbackMessage*, *ICallbackStop*, *ICallbackTime*, *ICallbackRunNo* and *ICallbackPlot*). For the *ICallbackTime*, *ICallbackRunNo* and *ICallbackPlot* interfaces only a dummy implementation is provided. For the *ICallbackMessage* interface, implementation is such that messages are written to the console. The implementation of the *ICallbackStop* interface always returns the same flag, providing for non-interruptable execution of the experiment.

After retrieving a pointer to the Tornado main entity through the *TGet* function, the program starts its actual processing. First, a reference to the factory component of the main entity (*pFactory*) is retrieved. Second, this factory is used to load the persistent representation of an experiment. While loading the experiment, an internal representation is built up, a reference to which is returned (*pExp*). Using this reference, the experiment is initialized and run. During the execution of the experiment, the callbacks implemented through the *CCallbacks* class are called at various instances.

Note that the program shown in Listing 10.2 is generic, in the sense that it can load and execute any type of experiment. The experiment that is handled could be a simple simulation experiment, but also a more complex optimization or scenario analysis.

Listing 10.2: Execution of an Experiment through the Tornado C++ API

```
#include "TornadoCPP/TornadoCPP.h"

#include "Common/String/Convert.h"
#include "Common/Ex/Streaming.h"

using namespace std;
using namespace Common;
using namespace Tornado;

class CCallbacks : virtual public ICallbackMessage ,
                  virtual public ICallbackStop ,
                  virtual public ICallbackTime ,
                  virtual public ICallbackRunNo ,
                  virtual public ICallbackPlot
{
    // ICallbackMessage

    virtual void Info(const wstring& Message)
    {
        wcout << L"I..." << Message << endl;
    }

    virtual void Warning(const wstring& Message)
    {
```

```

    wcout << L"W_L" << Message << endl;
}

virtual void Error(const wstring& Message)
{
    wcout << L"E_L" << Message << endl;
}

// ICallbackStop

virtual bool GetStop() const
{
    return false;
}

// ICallbackTime

virtual void SetTime(double Time)
{
}

// ICallbackRunNo

virtual void SetRunNo(const wstring& RunNo)
{
}

// ICallbackPlot

virtual void SetItems(const wstring& ID,
                     const vector<wstring>& Items)
{
}

virtual void SetNextRun(const wstring& ID,
                       const wstring& Info = L"")
{
}

virtual void SetValues(const wstring& ID,
                      const vector<double>& Values)
{
}
};

int main(int argc,
        char** argv)
{
    try
    {
        // Create callbacks

        CCallbacks Callbacks;

        // Initialize Tornado

        TInitialize(&Callbacks, L"Tornado.Main.xml", true);

        ITornado* pTornado = TGet();
        IFactory* pFactory = pTornado->GetFactory();

        // Load experiment

        auto_ptr<IExp> pExp(pFactory->ExpLoad(&Callbacks,
                                           &Callbacks,
                                           &Callbacks,
                                           &Callbacks,
                                           L"PredatorPrey/PredatorPrey.Simul.Exp.xml"));

        // Initialize & run experiment
    }
}

```

```

    pExp->Initialize ();
    pExp->Run ();

    // Finalize Tornado

    TFinalize ();
}
catch (const CEx& Ex)
{
    wcerr << String::ToWString(Ex) << endl;
    return EXIT_FAILURE;
}
catch (const exception& Ex)
{
    wcerr << String::StringToWString(Ex.what()) << endl;
    return EXIT_FAILURE;
}

return EXIT_SUCCESS;
}

```

Listing 10.3 illustrates a number of other aspects of the use of the Tornado C++ API. In this case a new ExpOptim experiment is created before it is executed. The program uses the same callback implementation class as was presented in the previous example. The source code of this class is therefore not reproduced in Listing 10.3. ExpOptim is a 3-level experiment, more specifically, as was mentioned in Chapter 8, it can be represented as *ExpOptim[ExpObjEval[ExpSimul[Model]]]*. Before the ExpOptim experiment can be created, an ExpObjEval experiment therefore needs to be created first. The ExpObjEval experiment is created by the *ExpCreateObjEval* main factory method, on the basis of an already existing ExpSimul experiment that is specified through a file name. After configuring the ExpObjEval experiment's objective, quantities and input file, the ExpOptim experiment is created by the *ExpCreateOptim* main factory method. After configuring the variables and optimization solver for this experiment, it is initialized, run and finally saved to a file.

Listing 10.3: Creation and Execution of an ExpOptim Experiment through the Tornado C++ API

```

int main(int argc,
        char** argv)
{
    try
    {
        // Create callbacks

        CCallbacks Callbacks;

        // Initialize Tornado

        TInitialize(&Callbacks, L"Tornado.Main.xml", true);

        ITornado* pTornado = TGet();
        IFactory* pFactory = pTornado->GetFactory();

        // Create ObjEval experiment

        auto_ptr<IExpObjEval> pExpObjEval(pFactory->ExpCreateObjEval(&Callbacks,
                                                                    &Callbacks,
                                                                    &Callbacks,
                                                                    &Callbacks,
                                                                    L"OUR.Simul.Exp.xml",
                                                                    L"."));

        pExpObjEval->ObjSetProp(L"EnableStorage", L"false");
        pExpObjEval->ObjSetProp(L"OutputFileName", L"OUR.ObjEval.Simul.{}.out.txt");

        pExpObjEval->ObjQuantityAdd(L".OUR.out");
        pExpObjEval->ObjQuantitySetProp(L".OUR.out", L"EnableMeanDiff", L"true");
        pExpObjEval->ObjQuantitySetProp(L".OUR.out", L"DiffCriterion", L"AbsSquared");
    }
}

```

```

pExpObjEval->ObjQuantitySetProp(L".OUR_out", L"DiffTimeWeighted", L"false");

pExpObjEval->InputsSetEnabled(true);

pExpObjEval->InputFileAdd(L"File");
pExpObjEval->InputFileSetName(L"File", L"OUR.Optim.in.txt");
pExpObjEval->InputFileSetEnabled(L"File", true);

// Create ExpOptim experiment

IExpOptim* pExpOptim = pFactory->ExpCreateOptim(&Callbacks,
                                                &Callbacks,
                                                &Callbacks,
                                                &Callbacks,
                                                pExpObjEval.release(),
                                                L".");

pExpOptim->VarAdd(L".Ks");
pExpOptim->VarSetProp(L".Ks", L"Initial Value", L"2");
pExpOptim->VarSetProp(L".Ks", L"LowerBound", L"0");
pExpOptim->VarSetProp(L".Ks", L"Scaling", L"2");
pExpOptim->VarSetProp(L".Ks", L"StepSize", L"1");
pExpOptim->VarSetProp(L".Ks", L"UpperBound", L"10");

pExpOptim->VarAdd(L".mumax");
pExpOptim->VarSetProp(L".mumax", L"Initial Value", L"0.00085");
pExpOptim->VarSetProp(L".mumax", L"LowerBound", L"0");
pExpOptim->VarSetProp(L".mumax", L"Scaling", L"0.00085");
pExpOptim->VarSetProp(L".mumax", L"StepSize", L"0.1");
pExpOptim->VarSetProp(L".mumax", L"UpperBound", L"10");

pExpOptim->VarAdd(L".s");
pExpOptim->VarSetProp(L".s", L"Initial Value", L"35");
pExpOptim->VarSetProp(L".s", L"LowerBound", L"0");
pExpOptim->VarSetProp(L".s", L"Scaling", L"35");
pExpOptim->VarSetProp(L".s", L"StepSize", L"2.5");
pExpOptim->VarSetProp(L".s", L"UpperBound", L"100");

pExpOptim->VarAdd(L".tau");
pExpOptim->VarSetProp(L".tau", L"Initial Value", L"2.5");
pExpOptim->VarSetProp(L".tau", L"LowerBound", L"0");
pExpOptim->VarSetProp(L".tau", L"Scaling", L"2.5");
pExpOptim->VarSetProp(L".tau", L"StepSize", L"1");
pExpOptim->VarSetProp(L".tau", L"UpperBound", L"10");

pExpOptim->SolveOptimSetMethod(L"Praxis");
pExpOptim->SolveOptimSetProp(L"Accuracy", L"1e-005");
pExpOptim->SolveOptimSetProp(L"Constrained", L"false");
pExpOptim->SolveOptimSetProp(L"Covariance", L"true");
pExpOptim->SolveOptimSetProp(L"IllConditioned", L"false");
pExpOptim->SolveOptimSetProp(L"MaxNoEvals", L"1000");
pExpOptim->SolveOptimSetProp(L"MaxNoStops", L"2");
pExpOptim->SolveOptimSetProp(L"MaxStepSize", L"10");
pExpOptim->SolveOptimSetProp(L"PrintLevel", L"3");
pExpOptim->SolveOptimSetProp(L"ScaleBound", L"1");

// Initialize & run ExpOptim experiment

pExpOptim->Initialize();
pExpOptim->Run();

// Save ExpOptim experiment

pExpOptim->ExpSave(L"OUR.Optim.Exp.xml");

// Finalize Tornado

TFinalize();
}
catch (const CEx& Ex)

```



```
{
    wcerr << String::ToString(Ex) << endl;
    return EXIT_FAILURE;
}
catch (const exception& Ex)
{
    wcerr << String::ToString(Ex.what()) << endl;
    return EXIT_FAILURE;
}

return EXIT_SUCCESS;
}
```

10.2.2 .NET

When the Java framework, which was first introduced in 1995, had gained a respectable level of popularity, Microsoft decided to launch a competing framework based on similar principles. This framework was named .NET¹ and was first distributed in November 2000 (v1.0 Beta 1). The most important component of the .NET Framework is the Common Language Infrastructure (CLI). The purpose of the CLI is to provide a language-agnostic platform for application development and execution, including, but not limited to, components for exception handling, garbage collection, security, and interoperability. Microsoft's implementation of the CLI is called the Common Language Run-time (CLR), which is composed of the following primary parts:

- Common Type System (CTS)
- Common Language Specification (CLS)
- Just-In-Time Compiler (JIT)
- Virtual Execution System (VES)

Generally speaking, the CLI allows for several programming languages to be used within the same framework, using the same type system. The set of supported languages is substantial and includes C#, Visual Basic .NET, Managed C++, J#, Python, Ruby and several others. In contrast to many other Microsoft developments of the past, the .NET framework has been designed with platform-independence and a certain level of openness in mind. In August 2000, Microsoft, Hewlett-Packard, and INTEL worked to standardize the CLI and the C# programming language. By December 2001, both were ratified ECMA standards (ECMA 335 and ECMA 334). ISO followed in April 2003 (ISO/IEC 23271 and ISO/IEC 23270).

The CLI and C# have many similarities to Sun's Java Virtual Machine (JVM) and the Java language. Both are based on a virtual machine model that hides the details of the computer hardware on which their programs run. Both use their own intermediate byte-code, *i.e.*, Java byte-code in the case of Java and Common Intermediate Language (CIL) in the case of .NET. On the .NET platform, the intermediate code is always compiled JIT; with Java the byte-code is either interpreted, compiled in advance, or compiled JIT. Both provide extensive class libraries that address many common programming requirements, and both address many security issues. The namespaces provided in the .NET Framework closely resemble the platform packages in the Java Enterprise Edition API specification, both in style and invocation.

.NET in its complete form is currently only fully available on Windows platforms and partially available on Linux and Mac, whereas Java is fully available on nearly all platforms. The .NET platform was built from the ground up to support multiple programming languages while targeting Microsoft Windows; the Java platform was from the onset built to support only the Java language on many operating

¹http://en.wikipedia.org/wiki/.NET_Framework

system platforms. Software components for .NET are distributed using self-contained and self-describing binary files named assemblies.

Figure 10.2 gives a high-level overview of the CLI: multiple programming languages are compiled to the same intermediate language, the CLR then processes the intermediate code in a JIT fashion in order to generate platform-specific binary code.

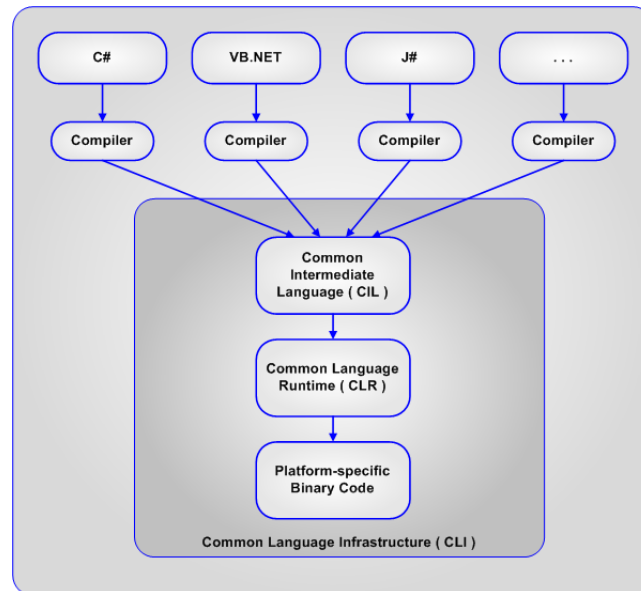


Figure 10.2: The .NET Framework

TornadoNET is a comprehensive API that was developed around the Tornado kernel. It allows for the full Tornado functionality to be adopted in the context of the .NET framework. The classes and methods of the TornadoNET API are very similar to those provided by the C++ API. Application developers who are familiar with the C++ API will therefore quickly learn how to use the .NET API and *vice versa*.

The implementation of the TornadoNET API was done in Managed C++, which is a .NET-specific version of the C++ language that allows for managed data types to be used. Managed data types are data types for which automatic garbage collection is available. The interesting feature of Managed C++ is that it can call Unmanaged (*i.e.*, plain) C++ directly, through a mechanism that is marketed by Microsoft as IJW (It Just Works). In addition to being able to call plain C++ directly, Managed C++ is a regular .NET language, in the sense that it uses the same type system as other .NET languages and that it can be compiled to the same type of intermediate language. Managed C++ is therefore an ideal vehicle for making the native C++ code of the Tornado kernel available in the context of .NET. In fact, the availability of a language such as Managed C++ is the prime reason why a comprehensive Tornado API for was developed .NET rather than for Java.

The TornadoNET implementation is straightforward and provides for bi-directional translation of calls, callbacks (events) and exceptions that go from the Tornado C++ level to the encapsulating .NET application and *vice versa*. Figure 10.3 shows that incoming .NET method calls are simply passed on to the corresponding C++ method. In the case of primitive method arguments (integers, reals, booleans) no translation of data types is required. However, in the case of non-primitive data types (vectors, strings) a conversion of data types is provided by TornadoNET. Table 10.1 gives an overview of the data types that are used in the scope of TornadoNET and how these are respectively implemented in C++ and .NET.

In .NET the concept of callback functions does not exist as such, an event-oriented mechanism has to be used instead. TornadoNET therefore provides for a translation of C++ callbacks to .NET events. Data

Table 10.1: Mapping between C++ and .NET Data Types in the Tornado .NET API

Description	C++	.NET
Boolean	bool	bool
Long integer	long	long
Double real	double	double
Wide-character string	std::wstring	System::String*
List of double reals	std::vector<double>	System::Collections::Generic::List<double>
List of wide-character strings	std::vector<std::wstring>	System::Collections::Generic::List<System::String*>

type conversions of callback/event arguments may also be required here. Finally, exceptions generated by the Tornado C++ code are converted to .NET exceptions.

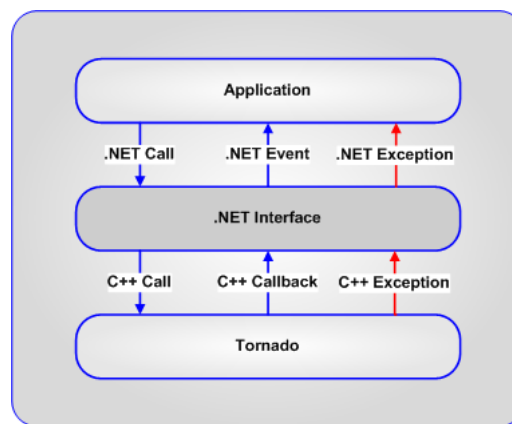


Figure 10.3: Correspondence between Calls, Callbacks (Events) and Exceptions in .NET and C++

In order to use the TornadoNET API from application programs one only needs one assembly file, *i.e.*, *TornadoNET.dll*. Listing 10.4 contains a C# implementation of the same program that was presented in Listing 10.2, using the Tornado .NET API. The program consists of a main routine and two event handlers, respectively for handling *CallbackMessage* and *CallbackStop* events. One of the convenient features of event-oriented mechanisms is that in case no event handler is provided by the programmer, a default dummy implementation will be activated. In the case of the example program, *CallbackTime* events will be handled by a default event handler since no implementation has been provided in the source code.

Listing 10.4: Execution of an Experiment through the Tornado .NET API (C#)

```
using System;
using TornadoNET;

class Example1
{
    static void Main(string[] args)
    {
        try
        {
            // Create Tornado

            TornadoNET.CTornado Tornado = new TornadoNET.CTornado();

            Tornado.EventSetMessage += new TornadoNET.CTornado._Delegate_EventSetMessage(SetMessage);
```

```

Tornado.Initialize("Tornado.Main.xml", true);

// Load experiment

TornadoNET.CExp Exp = Tornado.ExpLoad("PredatorPrey.Simul.Exp.xml");

Exp.EventSetMessage += new TornadoNET.CExp._Delegate_EventSetMessage(SetMessage);
Exp.EventGetStop += new TornadoNET.CExp._Delegate_EventGetStop(GetStop);

// Initialize & run experiment

Exp.Initialize();
Exp.Run();
}
catch (System.Exception Ex)
{
    System.Console.WriteLine(Ex.ToString());
}
}

static private void SetMessage(int Type,
                                string Message)
{
    switch (Type)
    {
        case 0:
            System.Console.Write("I...");
            break;

        case 1:
            System.Console.Write("W...");
            break;

        case 2:
            System.Console.Write("E...");
            break;
    }

    System.Console.WriteLine(Message);
}

static private bool GetStop()
{
    return false;
}
}

```

Listing 10.5 contains a C# implementation of the same program that was presented in Listing 10.3. As the program uses the same event handler implementations as in Listing 10.4, they have been omitted. The listing clearly shows that the event mechanism is the main conceptual and syntactical difference between the C++ and C# versions of the program. The actual processing that takes place (creation of experiments, configuring entities, *etc.*) is remarkably similar. To further illustrate the point that Tornado-based application code has a similar appearance regardless of the programming language, Listing 10.6 shows an implementation in VB.NET (VisualBasic for .NET) of the same program as presented in Listing 10.3 and Listing 10.5. Again, it is apparent that the syntax related to event handling is different, but the Tornado-related processing in terms of experiment manipulation is remarkably similar.

Listing 10.5: Creation and Execution of an ExpOptim Experiment through the Tornado .NET API (C#)

```

using System;
using TornadoNET;

class Example2
{
    static void
    Main(string[] args)
    {
        try

```

```

{
    // Create Tornado

    TornadoNET.CTornado Tornado = new TornadoNET.CTornado();

    Tornado.Initialize("Tornado.Main.xml", true);

    // Create experiment

    TornadoNET.CExpObjEval ExpObjEval = Tornado.ExpCreateObjEval("OUR.Simul.Exp.xml", ".");

    ExpObjEval.EventSetMessage += new TornadoNET.CExp._Delegate_EventSetMessage(SetMessage);

    ExpObjEval.ObjSetProp("EnableStorage", "false");
    ExpObjEval.ObjSetProp("OutputFileName", "OUR.ObjEval.Simul.{ }.out.txt");

    ExpObjEval.ObjQuantityAdd(".OUR_out");
    ExpObjEval.ObjQuantitySetProp(".OUR_out", "EnableMeanDiff", "true");
    ExpObjEval.ObjQuantitySetProp(".OUR_out", "DiffCriterion", "AbsSquared");
    ExpObjEval.ObjQuantitySetProp(".OUR_out", "DiffTimeWeighted", "false");

    ExpObjEval.InputsSetEnabled(true);

    ExpObjEval.InputFileAdd("File");
    ExpObjEval.InputFileSetName("File", "OUR.ObjEval.in.txt");
    ExpObjEval.InputFileSetEnabled("File", true);

    ExpObjEval.ExpSave("OUR.ObjEval.Exp.xml");

    TornadoNET.CExpOptim ExpOptim = Tornado.ExpCreateOptim("OUR.ObjEval.Exp.xml", ".");

    ExpOptim.EventSetMessage += new TornadoNET.CExp._Delegate_EventSetMessage(SetMessage);
    ExpOptim.EventSetRunNo += new TornadoNET.CExp._Delegate_EventSetRunNo(SetRunNo);

    ExpOptim.VarAdd(".Ks");
    ExpOptim.VarSetProp(".Ks", "InitialValue", "2");
    ExpOptim.VarSetProp(".Ks", "LowerBound", "0");
    ExpOptim.VarSetProp(".Ks", "Scaling", "2");
    ExpOptim.VarSetProp(".Ks", "StepSize", "1");
    ExpOptim.VarSetProp(".Ks", "UpperBound", "10");

    ExpOptim.VarAdd(".mumax");
    ExpOptim.VarSetProp(".mumax", "InitialValue", "0.00085");
    ExpOptim.VarSetProp(".mumax", "LowerBound", "0");
    ExpOptim.VarSetProp(".mumax", "Scaling", "0.00085");
    ExpOptim.VarSetProp(".mumax", "StepSize", "0.1");
    ExpOptim.VarSetProp(".mumax", "UpperBound", "10");

    ExpOptim.VarAdd(".s");
    ExpOptim.VarSetProp(".s", "InitialValue", "35");
    ExpOptim.VarSetProp(".s", "LowerBound", "0");
    ExpOptim.VarSetProp(".s", "Scaling", "35");
    ExpOptim.VarSetProp(".s", "StepSize", "2.5");
    ExpOptim.VarSetProp(".s", "UpperBound", "100");

    ExpOptim.VarAdd(".tau");
    ExpOptim.VarSetProp(".tau", "InitialValue", "2.5");
    ExpOptim.VarSetProp(".tau", "LowerBound", "0");
    ExpOptim.VarSetProp(".tau", "Scaling", "2.5");
    ExpOptim.VarSetProp(".tau", "StepSize", "1");
    ExpOptim.VarSetProp(".tau", "UpperBound", "10");

    ExpOptim.SolveOptimSetMethod("Praxis");
    ExpOptim.SolveOptimSetProp("Accuracy", "1e-005");
    ExpOptim.SolveOptimSetProp("Constrained", "false");
    ExpOptim.SolveOptimSetProp("Covariance", "true");
    ExpOptim.SolveOptimSetProp("IllConditioned", "false");
    ExpOptim.SolveOptimSetProp("MaxNoEvals", "1000");
    ExpOptim.SolveOptimSetProp("MaxNoStops", "2");
    ExpOptim.SolveOptimSetProp("MaxStepSize", "10");
    ExpOptim.SolveOptimSetProp("PrintLevel", "3");

```

```

    ExpOptim.SolveOptimSetProp("ScaleBound", "1");

    // Initialize & run experiment

    ExpOptim.Initialize();
    ExpOptim.Run();

    // Save experiment

    ExpOptim.ExpSave("OUR.Optim.Exp.xml");
}
catch (System.Exception Ex)
{
    System.Console.WriteLine(Ex.ToString());
}
}
}

```

Listing 10.6: Creation and Execution of an ExpOptim Experiment through the Tornado .NET API (VB.NET)

Module Example2

```

Sub HandlerEventSetMessage(ByVal Type As Integer, ByVal Message As String)
    If Type = 0 Then
        System.Console.WriteLine("I_" + Message)
    Else
        If Type = 1 Then
            System.Console.WriteLine("W_" + Message)
        Else
            If Type = 2 Then
                System.Console.WriteLine("E_" + Message)
            Else
                System.Console.WriteLine("?_" + Message)
            End If
        End If
    End If
End Sub

Sub HandlerEventSetRunNo(ByVal RunNo As String)
    System.Console.WriteLine("RunNo_" + RunNo)
End Sub

Sub Main()

    Try
        ' Create Tornado

        Dim Tornado As [TornadoNET].CTornado
        Dim ExpObjEval As TornadoNET.CExpObjEval
        Dim ExpOptim As [TornadoNET].CExpOptim

        Tornado = New [TornadoNET].CTornado

        AddHandler Tornado.EventSetMessage, AddressOf HandlerEventSetMessage

        Tornado.Initialize("Tornado.Main.xml", True)

        ' Create experiment

        ExpObjEval = Tornado.ExpCreateObjEval("OUR.Simul.Exp.xml", ".")

        AddHandler ExpObjEval.EventSetMessage, AddressOf HandlerEventSetMessage

        ExpObjEval.ObjSetProp("EnableStorage", "false")
        ExpObjEval.ObjSetProp("OutputFileName", "OUR.ObjEval.Simul.{ }.out.txt")

        ExpObjEval.ObjQuantityAdd(".OUR.out")
        ExpObjEval.ObjQuantitySetProp(".OUR.out", "EnableMeanDiff", "true")
        ExpObjEval.ObjQuantitySetProp(".OUR.out", "DiffCriterion", "AbsSquared")
        ExpObjEval.ObjQuantitySetProp(".OUR.out", "DiffTimeWeighted", "false")
    
```

```
ExpObjEval.InputsSetEnabled(True)

ExpObjEval.InputFileAdd("File")
ExpObjEval.InputFileSetName("File", "OUR.ObjEval.in.txt")
ExpObjEval.InputFileSetEnabled("File", True)

ExpObjEval.ExpSave("OUR.ObjEval.Exp.xml")

ExpOptim = Tornado.ExpCreateOptim("OUR.ObjEval.Exp.xml", ".")

AddHandler ExpOptim.EventSetMessage, AddressOf HandlerEventSetMessage
AddHandler ExpOptim.EventSetRunNo, AddressOf HandlerEventSetRunNo

ExpOptim.VarAdd(".Ks")
ExpOptim.VarSetProp(".Ks", "InitialValue", "2")
ExpOptim.VarSetProp(".Ks", "LowerBound", "0")
ExpOptim.VarSetProp(".Ks", "Scaling", "2")
ExpOptim.VarSetProp(".Ks", "StepSize", "1")
ExpOptim.VarSetProp(".Ks", "UpperBound", "10")

ExpOptim.VarAdd(".mumax")
ExpOptim.VarSetProp(".mumax", "InitialValue", "0.00085")
ExpOptim.VarSetProp(".mumax", "LowerBound", "0")
ExpOptim.VarSetProp(".mumax", "Scaling", "0.00085")
ExpOptim.VarSetProp(".mumax", "StepSize", "0.1")
ExpOptim.VarSetProp(".mumax", "UpperBound", "10")

ExpOptim.VarAdd(".s")
ExpOptim.VarSetProp(".s", "InitialValue", "35")
ExpOptim.VarSetProp(".s", "LowerBound", "0")
ExpOptim.VarSetProp(".s", "Scaling", "35")
ExpOptim.VarSetProp(".s", "StepSize", "2.5")
ExpOptim.VarSetProp(".s", "UpperBound", "100")

ExpOptim.VarAdd(".tau")
ExpOptim.VarSetProp(".tau", "InitialValue", "2.5")
ExpOptim.VarSetProp(".tau", "LowerBound", "0")
ExpOptim.VarSetProp(".tau", "Scaling", "2.5")
ExpOptim.VarSetProp(".tau", "StepSize", "1")
ExpOptim.VarSetProp(".tau", "UpperBound", "10")

ExpOptim.SolveOptimSetMethod("Praxis")
ExpOptim.SolveOptimSetProp("Accuracy", "1e-005")
ExpOptim.SolveOptimSetProp("Constrained", "false")
ExpOptim.SolveOptimSetProp("Covariance", "true")
ExpOptim.SolveOptimSetProp("IllConditioned", "false")
ExpOptim.SolveOptimSetProp("MaxNoEvals", "1000")
ExpOptim.SolveOptimSetProp("MaxNoStops", "2")
ExpOptim.SolveOptimSetProp("MaxStepSize", "10")
ExpOptim.SolveOptimSetProp("PrintLevel", "3")
ExpOptim.SolveOptimSetProp("ScaleBound", "1")

' Initialize & run experiment

ExpOptim.Initialize()
ExpOptim.Run()

' Save experiment

ExpOptim.ExpSave("OUR.Optim.Exp.xml")

Catch Ex As System.Exception
    System.Console.WriteLine(Ex.ToString())
End Try

End Sub

End Module
```

One last example that is presented in the scope of the .NET API illustrates how an experiment can be run in a separate thread. In Listing 10.5, after a simulation experiment is loaded, a thread instance is created and initialized with a reference to a procedure (*ThreadProc*) that is to be run when the thread is started. In this procedure, the simulation experiment is initialized and run. After the new thread is started, the original thread is available for other work. In this case, the value of a certain parameter is modified in the running experiment after 100 milliseconds of execution time. This example by itself may not be very useful, however it does illustrate how interactive applications could be built where users are able to set (and evidently also get) quantity values as they please during the course of a simulation experiment.

Listing 10.7: Execution of an ExpSimul Experiment in a Separate Thread through the Tornado .NET API

```
using System;
using System.Threading;
using TornadoNET;

class Example3
{
    static private TornadoNET.CExpSimul m_ExpSimul;

    static void
    Main( string [] args )
    {
        try
        {
            // Create Tornado

            TornadoNET.CTornado Tornado = new TornadoNET.CTornado();

            Tornado.EventSetMessage += new TornadoNET.CTornado._Delegate_EventSetMessage( SetMessage );

            Tornado.Initialize( "Tornado.Main.xml", true );

            // Load experiment

            m_ExpSimul = (CExpSimul)Tornado.ExpLoad( "PredatorPrey.Simul.Exp.xml" );

            m_ExpSimul.EventSetMessage += new TornadoNET.CExp._Delegate_EventSetMessage( SetMessage );
            m_ExpSimul.EventSetTime += new TornadoNET.CExp._Delegate_EventSetTime( SetTime );

            // Create and start thread

            Thread MyThread = new Thread( new ThreadStart( ThreadProc ) );
            System.Console.WriteLine( "Starting thread ..." );
            MyThread.Start();

            // Change value of parameter .c1 after 100ms

            Thread.Sleep( 100 );
            m_ExpSimul.ModelSetInitialValue( ".c1", 0.002 );
        }
        catch ( System.Exception Ex )
        {
            System.Console.WriteLine( Ex.ToString() );
        }
    }

    static private void
    ThreadProc()
    {
        m_ExpSimul.Initialize();
        m_ExpSimul.Run();

        System.Console.WriteLine( "Thread ended" );
    }
}
```


10.3 Restricted API's

Next to full-fledged, comprehensive API's for C++ and .NET, the Tornado framework also contains a number of restricted API's. The latter do not provide access to the entire Tornado kernel, but only allow for a limited number of frequently occurring operations. The operations that are supported by the restricted API's are the following:

- Initialization of the Tornado kernel (allocation of memory and potentially specification of callback function implementations)
- Loading of experiments
- Retrieving variable values from experiments
- Setting variable values in experiments
- Running experiments (combines initialization and execution of the experiment)
- Clean-up of the Tornado kernel (de-allocation of memory)

It must be noted that the interpretation of *variable values* in this context is rather broad. In the case of atomic experiment types, *variables* actually refer to quantities. As a result, any parameter, independent variable, input variable, output variable, worker variable and state variable can be set and/or retrieved through restricted API's. In the case of compound experiments, the interpretation of the term *variable* is experiment-dependent (as has been discussed in Chapter 8).

10.3.1 C

The first of several restricted API's for Tornado is the C API. In order to use this API from application programs, the following files are required:

- *TornadoC.lib*: Import library
- *TornadoC.dll* (Windows) or *TornadoC.so* (Linux): Dynamically-loadable library
- *TornadoC.h*: TornadoC header file

Listing 10.8 presents the Tornado C API. Listing 10.9 contains an implementation of the same program as in Listing 10.2, using the C API. Important to note is that callbacks are now implemented as regular C functions, instead of through a class implementing abstract callback interfaces.

Listing 10.8: Tornado C API

```
// Callback function types

typedef void (*TCallbackMessage)(unsigned int Type, wchar_t* Message);
typedef unsigned int (*TCallbackGetStop)(void);
typedef void (*TCallbackSetTime)(double Time);
typedef void (*TCallbackSetItems)(wchar_t* ID, wchar_t* Items);
typedef void (*TCallbackSetNextRun)(wchar_t* ID, wchar_t* Info);
typedef void (*TCallbackSetValues)(wchar_t* ID, double* Values);

// Functions

void TInitialize(TCallbackMessage pCallbackMessage,
                TCallbackGetStop pCallbackGetStop,
                TCallbackSetTime pCallbackSetTime,
                TCallbackSetItems pCallbackSetItems,
```

```

        TCallbackSetNextRun pCallbackSetNextRun ,
        TCallbackSetValues pCallbackSetValues ,
        const wchar_t* MainFileName ,
        char LoadPlugins );

void TFinalize(void);

void TExpLoad(const wchar_t* ExpFileName);

double TExpGetValue(const wchar_t* FullName);

void TExpSetValue(const wchar_t* FullName ,
                  double Value);

void TExpRun(void);

```

Listing 10.9: Execution of an Experiment through the Tornado C API

```

#include "TornadoC/TornadoC.h"

#include <stdio.h>

void CallbackMessage(unsigned int Type,
                    wchar_t* Message)
{
    switch (Type)
    {
        case 0:
            printf("I_ _%S\n", Message);
            break;

        case 1:
            printf("W_ _%S\n", Message);
            break;

        case 2:
            printf("E_ _%S\n", Message);
            break;

        default:
            printf("?_ _%S\n", Message);
    }
}

unsigned int CallbackGetStop(void)
{
    return 0;
}

void CallbackSetTime(double Time)
{
}

void CallbackSetItems(wchar_t* ID,
                     wchar_t* Items)
{
}

void CallbackSetNextRun(wchar_t* ID,
                       wchar_t* Info)
{
}

void CallbackSetValues(wchar_t* ID,
                      double* Values)
{
}

int main(int argc ,
        char** argv)
{

```

```
TInitialize ( CallbackMessage ,
             CallbackGetStop ,
             CallbackSetTime ,
             CallbackSetItems ,
             CallbackSetNextRun ,
             CallbackSetValues ,
             L"Tornado.Main.xml" ,
             TRUE);

TExpLoad(L"PredatorPrey.Simul.Exp.xml");

TExpRun();

TFinalize();

return 0;
}
```

10.3.2 JNI

The Java Native Interface (JNI)² is a programming framework that allows Java code running in the Java Virtual Machine (JVM) to call and be called by native applications (programs specific to a hardware and operating system platform) and libraries written in other languages, such as C, C++ and assembly. Typically, the JNI is used to write native methods to handle situations when an application cannot be written entirely in the Java programming language such as when the standard Java class library does not support the platform-specific features or program library. It is also used to modify an existing application, written in another programming language, to be accessible to Java applications.

The TornadoJNI API allows for limited access to the Tornado framework from Java programs, using JNI technology. Listing 10.10 contains the description of the TornadoJNI API. From a functional point of view it is similar to the TornadoC API, apart from the fact that callbacks cannot be defined. Instead, the implementation of the various callbacks is hard-coded in the C implementation of the TornadoJNI API itself. At a later stage, support for callback implementations at the Java level may also be included. The C implementation of the TornadoJNI API is based on the TornadoC API. Listing 10.11 contains a Java program that is similar to Listing 10.9 and that uses the TornadoJNI API.

Listing 10.10: Tornado JNI API

```
public class TornadoJNI
{
    public native void TInitialize (String MainFileName ,
                                   boolean LoadPlugins);

    public native void TFinalize ();

    public native void TExpLoad (String ExpFileName);

    public native double TExpGetValue (String FullName);

    public native void TExpSetValue (String FullName ,
                                     double Value);

    public native void TExpRun ();

    static
    {
        System.loadLibrary ("TornadoJNI");
    }
}
```

²http://en.wikipedia.org/wiki/Java_Native_Interface

Listing 10.11: Execution of an Experiment through the Tornado JNI API

```

public class Example1
{
    public static void main(String[] args)
    {
        TornadoJNI Handle = new TornadoJNI();

        Handle.TInitialize("Tornado.Main.xml", true);

        Handle.TExpLoad("PredatorPrey.Simul.Exp.xml");

        Handle.TExpRun();

        Handle.TFinalize();
    }
}

```

Figure 10.4 clarifies the steps that were required during the development of the TornadoJNI API. Initially, the API definition was coded (cf. Listing 10.10) and saved in a file named *TornadoJNI.java*. Next a C header file (*TornadoJNI.h*) was generated automatically with the Java C Header and Stub Generator (*javah*). The functions declared in this header file were implemented in *TornadoJNI.c* using the TornadoC API. The resulting code was compiled with a C compiler, linked with the TornadoC static library (*TornadoC.lib*) and packaged as a DLL (*TornadoJNI.dll*).

Figure 10.4 also clarifies how Java applications that use the TornadoJNI API are to be built and deployed. First, *TornadoJNI.java* and the Java code of the application (*Application.java*) are to be compiled with a Java compiler (*javac*). At run-time, the two resulting Java class files (*TornadoJNI.class* and *Application.class*) are required, as well as the TornadoC DLL (*TornadoC.dll*) and the TornadoJNI DLL (*TornadoJNI.dll*).

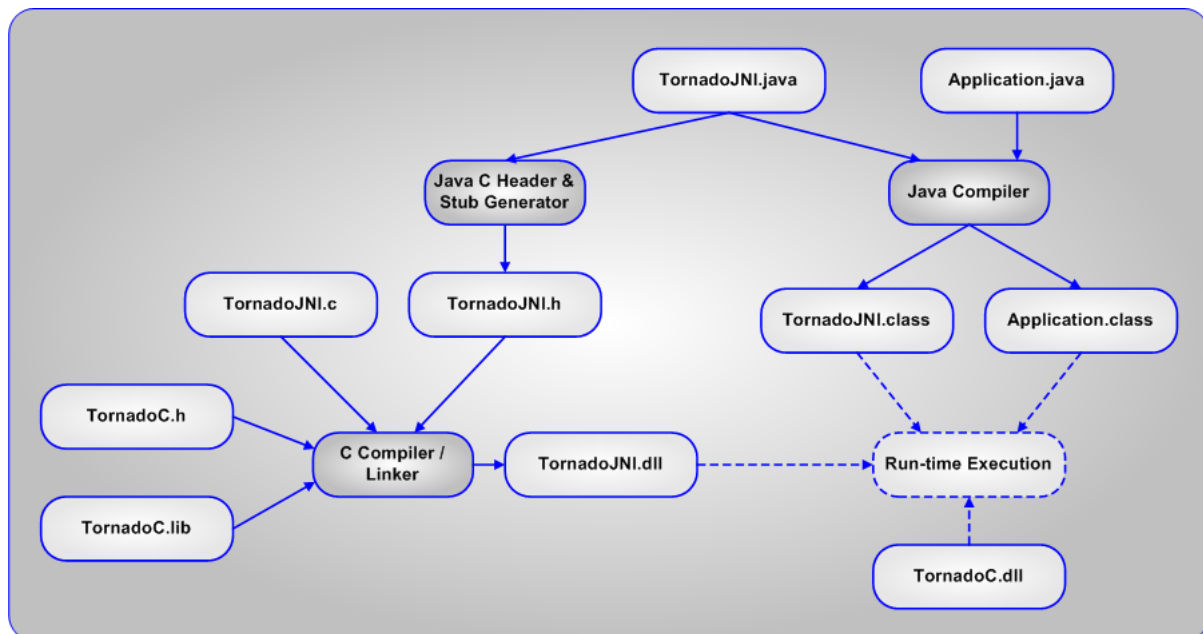


Figure 10.4: Building and Deploying Applications using the Tornado JNI API

10.3.3 MEX

MATLAB allows for external DLL's to be called from its M-file interpreter through a protocol that is referred to as MEX (MATLAB EXternal). In order to be callable from MATLAB, a DLL must reside

in the MATLAB path and must export a function with a particular signature. DLL's are called from the M-file interpreter simply by using the name of the DLL. In addition, a number of parameters may be specified. The signature of the one function that must be exported by a MEX DLL is the following:

```
void mexFunction(int nlhs,
                 mxArray* plhs[],
                 int nrhs,
                 const mxArray* prhs[])
```

MATLAB will use the *nrhs* and *prhs* arguments to supply information about the parameters of the call to the DLL (*i.e.*, Right Hand Side - RHS). Conversely, the *nlhs* and *plhs* arguments should be used by the implementation of *mexFunction* to supply information to MATLAB about the return parameters of the call (*i.e.*, Left Hand Side - LHS). MEX DLL's are stateless. When a call from the M-file interpreter is issued, the DLL is loaded, the exported function is executed, and the DLL is again unloaded.

TornadoMEX is a MEX DLL that was created in order to be able to run Tornado from within MATLAB. Since MEX DLL's are stateless and only export one function, TornadoMEX had to be conceived as an all-in-one operation. More specifically, TornadoMEX implements a function that takes a number of parameters that are interpreted in a number of alternative ways. The general signature of the MATLAB function that is implemented through the TornadoMEX DLL is *TornadoMEX(<Mode>, <Arg1>, <Arg2>, <Arg3>)*. The following situations can occur:

- *<Mode> = 'EndValue'*: The experiment XML file specified through *<Arg1>* is loaded and the initial values set through *<Arg2>* are applied. *<Arg2>* is a string that is structured according to the (*<QuantityName> '=' <Value> ';'**) pattern. *<Arg3>* contains a list of quantities for which the final values are to be returned to MATLAB as a vector. *<Arg3>* is a string that is structured according to the (*<QuantityName> ';'**) pattern.
- *<Mode> = 'Trajectory'*: The experiment XML file specified through *<Arg1>* is loaded and the initial values set through *<Arg2>* are applied. *<Arg2>* is a string that is structured according to the (*<QuantityName> '=' <Value> ';'**) pattern. *<Arg3>* contains a list of quantities for which the trajectories are to be returned to MATLAB as a matrix. *<Arg3>* is a string that is structured according to the (*<QuantityName> ';'**) pattern.
- *<Mode> = 'DataFile'*: The data file specified through *<Arg1>* is loaded and its contents are returned to MATLAB as a matrix. In this case, *<Arg2>* and *<Arg3>* are not used.

Listing 10.12 shows some examples of the use of TornadoMEX from within MATLAB.

Listing 10.12: Examples of the Application of the Tornado MEX API

```
clear
y = TornadoMEX('EndValue', 'PredatorPrey.Simul.Exp.xml', '.cl=0.1;', '.pa.out_1;.ps.out_1;')

clear
y = TornadoMEX('Trajectory', 'PredatorPrey.Simul.Exp.xml', '.cl=0.1;', '.pa.out_1;.ps.out_1;');
plot( y(:,1), y(:,2) )
x1 = min(y(:,2))
x2 = mean(y(:,3))

clear
y = TornadoMEX('DataFile', 'PredatorPrey.Simul.out.txt', '', '');
plot(y(:,1), y(:,2))
```

10.3.4 OpenMI

OpenMI, the Open Modelling Interface³ (Gregersen et al., 2007), has been designed to provide a widely accepted unified method to link model engines in the integrated water catchment management domain. OpenMI was developed during the HarmonIT project, which was a large-scale research project supported by the European Commission under the 5th Framework Programme. Current development and maintainance of OpenMI is handled by the OpenMI Association.

OpenMI provides a number of tools and interface descriptions that allow software vendors and suppliers to link legacy or newly-developed model engines to model engines developed by other parties. OpenMI provides an interface mechanism for Microsoft .NET and for Java. Basically, model engines (*i.e.*, the combination of a model and its solver) are to be wrapped in such a way that they can accept outputs provided by other models as input, and provide outputs to other engines as input. OpenMI is based on chaining of model engines. The top-most model engine starts the simulation process by issuing a request for data (for a certain time point and/or spacial location) to the next model engine, which can again request for data to other engines. The process continues until the initial request can be fully resolved.

An experimental OpenMI wrapper for Tornado was developed, on the basis of the Tornado .NET API. The Tornado OpenMI wrapper was named TornadoOMI and supports the most basic OpenMI model engine linkage functionalities. Upon initialization, TornadoOMI loads a Tornado XML experiment description that is referenced through an OpenMI model engine configuration file. At run-time, it provides data for a number of desired quantities at the time points for which data is requested, by running simulations. Simulation results are stored in an internal buffer. Two situations can occur:

- In the first case, simulation data is already available for the time point for which data is requested. Data can therefore simply be returned by performing interpolation on the model engine's internal data buffer.
- In the second case, no data is yet available for the requested time point. A simulation is therefore to be run from the last time point for which data is available, to the new time point. Since during this simulation, data will be stored in the model engine's internal buffer, future requests for data for earlier time points can be handled as in the first case.

10.3.5 CMSLU

In some cases, a system needs to be modelled that has one or more components of which the behavior is governed by the results of a dynamic simulation. A good example is Model Predictive Control (MPC)⁴. In MPC one or more control actions are determined by the outcome of the simulation of a dynamic model over a certain time horizon. The assumption is that the prediction of future system behavior on the basis of dynamic system simulation may be beneficial to the quality of the control actions.

Simulation of a MPC or similar system is special in the sense that at every simulation time point (or at a subset of time points), another simulation has to be run, making the simulation process recursive. In order to allow for this, an API was created for Tornado that allows for the Tornado kernel to be called from within Tornado models. This API was named TornadoCMSLU as it is implemented on top of the TornadoC API and could be considered an extension of the MSLU library (since it can be called from within Tornado models).

At first sight, one might wonder why it would not be possible to call the Tornado C API from models, since in general, C functions can be called directly from model code. However, there are two issues that

³<http://www.openmi.org>

⁴http://en.wikipedia.org/wiki/Model_predictive_control

prevent TornadoC to be used directly. First of all, models cannot provide for callback implementations, default implementations are therefore required. Secondly, models are composed of equations that are sorted by the model compiler. When mixing procedural code with equations one must therefore ensure that causality is maintained. TornadoCMSLU provides for these two extra features, in addition to the basic functionality that is provided by TornadoC.

Listing 10.13 contains an example of a model that calls Tornado in order to perform a simulation. For the parameter *.pa.p* of the embedded model, the value of the state variable *x* of the encapsulating model is used. Also, for the output variable *y* of the encapsulating model, the final value of the variable *.pa.out_1* of the embedded model is used. The dummy variables are required in order to ensure the correct ordering of statements.

Listing 10.13: Example of the Application of the Tornado C MSLU API

```

fclass DemoMPC

  output Real y(start = 0);

  Real dummy1(start = 0);
  Real dummy2(start = 0);
  Real dummy3(start = 0);
  Real dummy4(start = 0);

  Real w(start = 0);
  Real x(start = 0);

initial equation

  dummy1 = MSLUTInitialize(0, "Tornado.Main.xml", 1);
  dummy2 = MSLUTExpLoad(dummy1, "PredatorPrey.Simul.Exp.xml");

equation

  w = sin(t);
  der(x) = w;

  dummy3 = MSLUTExpSetValue(dummy2, ".pa.p", x);
  dummy4 = MSLUTExpRun(dummy3);
  y = MSLUTExpGetValue(dummy4, ".pa.out_1");

final equation

  MSLUTFinalize(0);

end DemoMPC;

```

10.3.6 CLIPS

CLIPS⁵ is a public-domain software tool for building expert systems. The name is an acronym for “C Language Integrated Production System”. The first versions of CLIPS were developed starting in 1985 at NASA’s Johnson Space Center as an alternative for the existing ART*Inference system. Development at the Johnson Space Center continued until the mid 1990’s, when the development group’s responsibilities ceased to focus on expert system technology.

CLIPS is probably the most widely used expert system tool because it is fast, efficient and free. Although it is now in the public domain, it is still updated and supported by the original author, Gary Riley. CLIPS incorporates a complete object-oriented language, named COOL, for writing expert systems. Although it is written in C, the syntax of COOL more closely resembles that of the programming language LISP. Extensions for CLIPS can be written in C, and CLIPS can be called from C. As other expert system

⁵<http://www.ghg.net/clips/CLIPS.html>

languages, CLIPS deals with rules and facts. Various facts can make a rule applicable. An applicable rule is then asserted. Facts and rules are created by first defining them.

Descendants of the CLIPS language include Jess (rule-based portion of CLIPS rewritten in Java), Haley Eclipse, FuzzyCLIPS (which adds the concept of relevancy into the language), EHSIS (that adds support for GUI, Multimedia, IPC, *etc.* on Windows) and others.

TornadoCLIPS is an executable program that contains the CLIPS COOL interpreter and run-time system, extended with restricted access to the Tornado kernel. TornadoCLIPS extends the COOL language with a number of functions that allow for Tornado to be initialized, for experiments to be loaded and executed, and for quantity values to be set and retrieved. The implementation of the Tornado-related extension of the COOL language was done through the Tornado C API.

Listing 10.14 contains a simple program that illustrates the use of Tornado from within the CLIPS COOL interpreter.

Listing 10.14: Example of the Application of the Tornado CLIPS API

```
(tinitialize "Tornado.Main.xml" 1)
(texpload "PredatorPrey.Simul.Exp.xml")
(texpsetvalue ".ps.p" 20)
(texprun)
(texpgetvalue ".pa.out.1")
(texpgetvalue ".ps.out.1")
(tfinalize)
(exit)
```


11

Command-line User Interface

11.1 Introduction

Of all types of applications that can be built on top of the Tornado kernel, the Tornado Command-line User Interface (CUI) is one of the most basic. It is also the only application that is developed along with the Tornado kernel itself, and is part of the same source code base.

The Tornado CUI is a suite of small text-only non-interactive command-line programs that allow for using and manipulating various aspects of the Tornado kernel. All programs are portable across platforms and were implemented in C++. Implementation was done directly on top of the native C++ interface of the Tornado kernel.

Historically, many operating systems and applications were command-line oriented. For instance, when the UNIX operating system was first introduced in 1969, it came with a set of command-line utilities (some of which are still in use to date). At the end of the 1980's, Graphical User Interfaces (GUI) quickly gained popularity, and CUI tools were banned by many non-expert users of computer systems. Recent years have however seen a slight reverse trend: it is being realized that for some types of work (*e.g.*, batch processing), CUI tools can provide a flexible and efficient solution. These considerations have also contributed to the decision to provide a suite of CUI tools in the scope of Tornado. In fact, the following can be stated as the most important advantages of the availability of a CUI suite for Tornado:

- **Testing of new functionality:** Since the Tornado command-line programs are text-only and directly built on top of the kernel's C++ API, the development effort is limited. As a result, changes to the kernel's C++ API can easily be reflected by the CUI programs. These programs are therefore well-suited as a "first-line" test bed for new or modified kernel features.
- **Automation at the OS level:** Most operating systems allow for command-line programs to be used from a command-line shell, which is a program that allows for typing, editing and executing OS commands. Most shells also provide for constructs similar to programming languages, such as loops and conditional statements. Examples of command-line shells are the Bourne Shell (*sh*), C Shell (*csh*), Bourne Again Shell (*bash*) and Korn Shell (*ksh*) on Unix-like systems, the *command.com* program on older Windows system, and the *cmd.exe* program on more recent Windows systems. Programs written using the features of a command-line shell are called scripts on Unix-like systems, and batch files on Windows systems. The Tornado CUI tools can be called from such scripts or batch files, allowing for automation of Tornado-related tasks at the level of the OS.

- **Efficiency in terms of execution speed and memory consumption:** The level of overhead that is incurred by command-line programs is generally low. Also in the case of the Tornado CUI tools, speed of execution and memory consumption are respectively higher and lower than what can be obtained with other types of (graphical) applications. For those situations where the utmost efficiency is required, the CUI tools are therefore a good option.

11.2 Components of the CUI Suite

The current Tornado CUI suite contains 27 tools. Some of these tools are trivial, others are more convoluted. The following are the categories that these tools can be divided into:

- **Entity Management Utilities:** Allow for managing the persistent representation of entities. Entities can be loaded, queried, modified (in case of read-write entities) and stored in their respective XML representations. Only one query or modification can be done at a time. In between, the XML representation will be saved, and loaded again when the program is restarted.
- **Model Compilers and Builders:** Model compilers allow for converting high-level model descriptions to executable model code. Builders rely on a regular C compiler to compile and link this executable code to a binary plug-in.
- **Conversion Utilities:** Perform conversions of persistent formats.
- **Miscellaneous:** Perform miscellaneous types of operations.

Table 11.1 gives an overview of the current Tornado CUI suite. For each tool, the name of the executable, a description and one of the above-mentioned categories is listed. Below follows more information on the tools mentioned in the table:

- **mof2t:** Has as its main purpose to convert flat Modelica models to executable model code. However, other types of input and output are also supported.
- **t2batch:** Converts an XML representation of a set of jobs (to be discussed in Chapter 14) into a set of batch files on the Windows platform or shell scripts on the Linux platform, for local execution.
- **t2jdl:** Converts an XML representation of a set of jobs into a set of JDL files, for execution through the gLite grid middleware (to be discussed in Chapter 14).
- **t2jobs:** Converts an XML representation of a Tornado virtual experiment to an XML representation of a job for use with the Typhoon distributed execution framework (to be discussed in Chapter 14).
- **t2msl:** Converts an XML representation of a Tornado layout to a coupled model description in MSL.
- **tbuild:** Compiles and links the executable model code that was generated by a model compiler to a binary plug-in for use with Tornado. Internally, *tbuild* relies on a regular C compiler.
- **tcontrols:** Allows for loading, querying, modifying and storing XML representations of graphical input providers.

- **tcreatematlab**: Allows for generating executable model code that acts as a wrapper around executable code that has been generated from MATLAB by the MATLAB compiler. In practice, this tool has proven to be of limited use since the code that is generated by the MATLAB compiler can be quite slow. Actually, the MATLAB compiler is specifically intended for the creation of redistributable run-time programs (for which the MATLAB environment is not required), and does not focus on performance. In fact, all targets generated by the MATLAB compiler consist of a Java archive that is unpacked at run-time and run with a Java Virtual Machine.
- **tcreatesymb**: Allows for creating an XML symbolic model representation on the basis of an executable model plug-in. Normally, executable model code in C and symbolic model representations in XML are generated side-by-side by a model compiler. In case an executable model plug-in exists for which no symbolic model information is available, the latter can be generated from the executable model plug-in using *tcreatesymb*. For sub-models, parameters and variables, automatically generated default names will be used.
- **tcrypt**: Allows for encrypting or decrypting a file using a 3-DES key that is specific for Tornado. This tool is used in the scope of the Tornado licensing mechanism.
- **texec**: Loads and executes an XML representation of a virtual experiment.
- **texp**: Allows for loading, querying, modifying and storing XML representations of virtual experiments.
- **tgethostname**: Returns the hostname of the machine the program is run on.
- **tgetid**: Returns the identifier of a binary plug-in for Tornado.
- **tgetinfo**: Returns information on the environment in which Tornado was installed.
- **tgetmac**: Returns the MAC address (*i.e.*, unique network card address) of the machine the program is run on.
- **tgetmetric**: Returns a number of metrics related to the complexity of a virtual experiment.
- **ticonlib**: Allows for loading, querying, modifying and storing XML representations of icon libraries.
- **tinitial**: Loads an XML representation of a virtual experiment and performs initialization, but does not run the experiment. This tool is quite helpful to track problems related to erroneous initializations.
- **tlayout**: Allows for loading, querying, modifying and storing XML representations of layouts.
- **tmain**: Allows for loading, querying, modifying and storing XML representations of the main singleton. Also allows for the creation of XML representations of new entities through the factory functionality of the main singleton.
- **tmodel**: Allows for loading and querying XML representations of symbolic models.
- **tmodellib**: Allows for loading and querying XML representations of model libraries.
- **tmsl**: Converts a MSL model description to executable model code.
- **tplot**: Allows for loading, querying, modifying and storing XML representations of graphical output acceptors.

- **tproject**: Allows for loading, querying, modifying and storing XML representations of projects (projects are combinations of layouts and related experiments).
- **tprops2html**: Allows for generating a static HTML representation of the properties that are supported by the entities of the Tornado kernel. The *tprops2html* tool is compiled on the basis of the same property descriptions that are also used to build the Tornado kernel itself. Upon invocation, *tprops2html* exhaustively loops through these property lists and generates readable HTML documentation. In this way, it can be stated that the kernel – to a certain extent – documents itself.
- **ttest**: Allows for recursively applying a number of operations on all virtual experiments contained in a hierarchical project. Successful and unsuccessful operations are logged.
- **wco2t**: Converts an XML representation of a layout in WEST-3 format to a representation in the corresponding Tornado format.
- **wxp2t**: Converts an XML representation of a virtual experiment in WEST-3 format to a representation in the corresponding Tornado format.

Table 11.1: Tornado CUI Suite

Name	Description	Category
mof2t	Flat Modelica to Tornado Convertor	Model Compilers and Builders
t2batch	Typhoon to Batch Convertor	Convertor
t2jdl	Typhoon to JDL Convertor	Convertor
t2jobs	Tornado to Typhoon Convertor	Convertor
t2msl	Tornado Layout to MSL Convertor	Convertor
tbuild	Tornado Model Builder	Model Compiler and Builders
tcontrols	Tornado Controls Spec Utility	Entity Management Utilities
tcreatematlab	Tornado MATLAB Model Wrapper Creator	Miscellaneous
tcreatesymb	Tornado Symbolic Model Spec Creator	Miscellaneous
tcrypt	Tornado File Cryptor	Miscellaneous
texec	Tornado Experiment Executor	Miscellaneous
texp	Tornado Experiment Spec Utility	Entity Management Utilities
tgethostname	Tornado Hostname Retriever	Miscellaneous
tgetid	Tornado Plug-in ID Retriever	Miscellaneous
tgetinfo	Tornado Info Retriever	Miscellaneous
tgetmac	Tornado MAC Address Retriever	Miscellaneous
tgetmetric	Tornado Complexity Metric Retriever	Miscellaneous
ticonlib	Tornado Icon Library Spec Utility	Entity Management Utilities
tinitial	Tornado Experiment Initialization Executor	Miscellaneous
tlayout	Tornado Layout Spec Utility	Entity Management Utilities
tmain	Tornado Main Spec Utility	Entity Management Utilities
tmodel	Tornado Model Spec Utility	Entity Management Utilities
tmodellib	Tornado Model Library Spec Utility	Entity Management Utilities
tmsl	Tornado MSL Model Compiler	Model Compiler and Builders
tplot	Tornado Plot Spec Utility	Entity Management Utilities
tproject	Tornado Project Spec Utility	Entity Management Utilities
tprops2html	Tornado Property Lister	Miscellaneous
ttest	Tornado Test Utility	Miscellaneous
wco2t	WCO to Tornado Convertor	Convertor
wxp2t	WXP to Tornado Convertor	Convertor

11.3 Flow of Data

Figure 11.1 gives an overview of the most relevant flows of data across the most important CUI tools. A complete representation of all possible interactions amongst tools would lead to an overly cluttered figure.

As can be seen from the figure, a layout can either be created using the *tlayout* tool, or converted from an already existing WEST-3 layout. From layouts, MSL code can be generated through the application of *t2msl*. Subsequently, this MSL code can be converted to executable model code (and an XML symbolic model representation) by *tmsl*. The latter can also generate a static XML representation of the model library, which can be queried by *tmodellib*.

An alternative path starts by invoking the OpenModelica Compiler on a full Modelica model. The resulting flat Modelica model is then processed by *mof2t* in order to generate executable model code and an XML symbolic model representation.

An executable model plug-in is created from the C code representing an executable model by *tbuild*. This plug-in can be used by *texec* to execute a virtual experiment. These virtual experiments can either be generated through the application of *texp*, or converted from a WEST-3 experiment representation.

Two other paths that have not yet been mentioned are the generation of S-function C code from *mof2t* (to be further processed by the MATLAB *mex* utility for inclusion in Simulink) and the conversion of experiment representations to job descriptions for Typhoon or gLite (to be discussed in Chapter 14).

11.4 Usage

Programs from the Tornado CUI suite can be used by typing the name of the program, followed by a number of optional and fixed arguments, in an operating system shell. For each option, a shorthand and a longhand are available. For instance, for specifying the location of the Tornado main XML configuration file, either *-c* or *-config* can be used.

One of the most important options that can be applied to each program is *-h* (*-help*), which displays an overview of the fixed and optional arguments that can be used. For instance, for the *mof2t* and *tmain* commands, the output contained in Listing 11.1 and Listing 11.2 will be generated.

Listing 11.1: On-line Help for the *mof2t* Command

```
Flat Modelica to Tornado Convertor (Build: Oct 22 2007, 16:14:29)

Usage:
mof2t [options] <FlatModelicaFile | XMLFile>

Options:
-h,  --help                Show this message.
-c,  --config <XMLFile>    Specify Tornado main spec file name.
-l,  --log <File>          Specify log file name (use - for stdout).
-b,  --check_bounds        Enable bounds checking.
-r,  --remove_unused       Enable removal of unused quantities.
-t,  --instrument          Enable code instrumentation.
-i,  --include <String>    Specify additional includes (separated by ;).
-ne, --no_equiv            Disable equiv handling.
-nc, --no_constant        Disable handling of constant equations.
-ni, --no_initial          Disable handling of initial equations.
-no, --no_output           Disable handling of output equations.
-nr, --no_readable         Disable generation of readable names in executable code.
-it, --input_type <Spec>   Specify input type.
-ot, --output_type <Spec>  Specify output type (separated by ;).
-od, --output_dir <Dir>    Specify output directory.
-of, --output_file <File>  Specify output file.
-q,  --quiet               Quiet mode.

Valid input type specs:
```

MOF / XML

Valid output syntax specs:

MOF / MO / MSL / C / SFunction / XMLSymb / XMLExec / MathML / Complexity / GraphViz

Listing 11.2: On-line Help for the *tmain* Command

Tornado Main Spec Utility (Build: Jan 16 2008, 12:23:30)

Usage:

C:\Data\src\Tornado\bin\win32-msvc8\tmain [options] [<Command>]

Options:

-h, --help	Show this message.
-c, --config <XMLMainFile>	Specify Tornado main spec file name.
-d, --dir <Dir>	Specify output directory for specs created by factory.
-l, --log <File>	Specify log file name (use - for stdout).
-o, --output <XMLMainFile>	Specify Tornado Main spec file name for output.
-q, --quiet	Quiet mode.

Main commands:

MainGetProps
MainSetProp <PropName> <Value>
MainGetProp <PropName>
MainGetPropInfo <PropName>

Plugin commands:

PluginEnumerate
PluginAdd <PluginName>
PluginRemove <PluginName>
PluginGetProps <PluginName>
PluginGetPropInfo <PluginName> <PropName>

Unit commands:

UnitEnumerate
UnitAdd <UnitName>
UnitRemove <UnitName>

Project commands:

ProjectCreate <ProjectName>

Controls commands:

ControlsCreate <ControlsName>

Experiment commands:

ExpCreateCI <XMLExpObjEvalFile> <Path>
ExpCreateEnsemble <BaseName> <Path>
ExpCreateMC <XMLExpObjEvalFile> <Path>
ExpCreateMCOptim <XMLExpOptimFile> <Path>
ExpCreateObjEval <XMLExpSimulFile> <Path>
ExpCreateOptim <XMLExpObjEvalFile> <Path>
ExpCreateScen <XMLExpObjEvalFile> <Path>
ExpCreateScenOptim <XMLExpOptimFile> <Path>
ExpCreateScenSSRoot <XMLExpSSRootFile> <Path>
ExpCreateSens <XMLExpSimulFile> <Path>
ExpCreateSeq <BaseName> <Path>
ExpCreateSimul <ModelName> <Path>
ExpCreateSSOptim <ModelName> <Path>
ExpCreateSSRoot <ModelName> <Path>
ExpCreateStats <BaseName> <Path>

Plot commands:

PlotCreate <PlotName>

Icon library commands:

IconLibCreate <IconLibName>

Layout commands:

LayoutCreate <LayoutName>

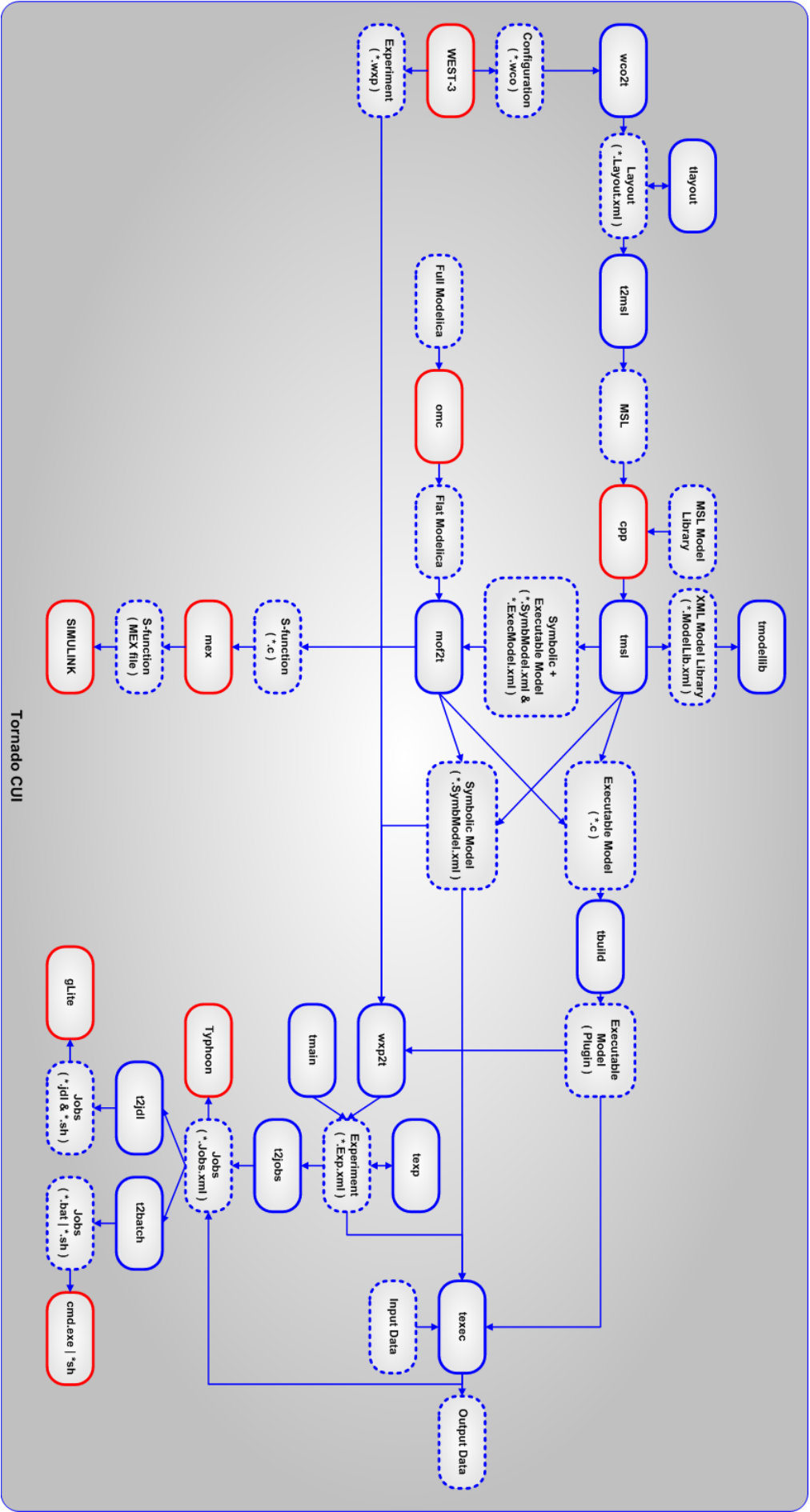


Figure 11.1: Tornado CUI Suite Data Flow

12

Graphical User Interfaces

12.1 Introduction

This chapter gives a concise overview of a number of applications with a Graphical User Interface (GUI) that have been developed on top of the Tornado kernel. First, three applications are discussed that were developed using the first incarnation of the Tornado kernel, *i.e.*, Tornado-I. Finally, two recent applications are discussed that are being developed on top of the second incarnation of the Tornado kernel, *i.e.*, Tornado-II.

12.2 Tornado-I

12.2.1 WEST++

Historically, the first application that was developed using the Tornado-I kernel was a research application named WEST++ (Vangheluwe et al., 1998). WEST++ is a portable, generic modelling and virtual experimentation application that is based on the Tcl language and the Tk graphical widget toolkit¹. The ++ in the name refers to the fact that C++ was used as a programming language, and that the application is more powerful than the WEST-1 program that was commercially available at the time.

Tcl (Tool Command Language) is a scripting language created by John Ousterhout. After its inception in the early 1980's, Tcl quickly gained wide acceptance. It is generally considered to be easy to learn, but nonetheless powerful in competent hands. It is most commonly used for rapid prototyping, scripted applications, GUI's and testing. Tcl is used extensively on embedded system platforms, both in its full form and in several other small-footprinted versions. The combination of Tcl and the Tk GUI toolkit is referred to as Tcl/Tk.

Tcl is an extensible language. New commands can be implemented in traditional programming languages such as C or C++ and then registered with the Tcl interpreter. A set of related commands for extension of the Tcl interpreter is named a package. Tk has a special relationship with respect to Tcl, but from a technical point of view, it is merely a package. Other packages that are frequently used for GUI development with Tcl/Tk are BLT and Tix. Packages implemented in native languages such as C and C++ typically consist of a DLL with a standardized external interface. For use with WEST++,

¹<http://www.tcl.tk>

the Tornado-I kernel was extended with such a standardized Tcl command registration interface and packaged as a DLL.

WEST++ is a distributed application. It consists of several self-contained modules that communicate over the network. In the context of WEST++, these modules are named *macro-modules*. Each macro-module consists of a Tcl interpreter, extended with a number of packages and a socket-based communication layer. The following is an overview of the macro-modules that have been developed for WEST++:

- **IMMC (Inter Macro Module Communication):** Is the heart of the macro-module communication architecture. It can be regarded as a registry with which macro-modules register upon startup. A macro-module can retrieve information about how to contact other macro-modules by issuing a query to the IMMC module.
- **Simul:** Simulation engine that implements dynamic simulation of executable models using a number of hard-coded (*i.e.*, non-pluggable) integration algorithms.
- **Optimization:** Optimization engine that implements parameter estimation through the RMSE (Root Mean Squared Error) approach. A number of hard-coded optimization algorithms are available.
- **SA:** Sensitivity Analysis engine that implements the finite difference approach through forward differences.
- **Exp:** Main module of the Experimentation Environment that coordinates operations among Simul, Optim and SA.
- **HGE (Hierarchical Graph Editor):** Graphical editor that allows for coupled model graphs to be built on the basis of nodes representing sub-models.
- **Mod:** Main module of the Modelling Environment that coordinates operations among multiple HGE instances and allows for the generation of executable models through the application of model compilation and building.
- **MoSS (Model-based Simulation System for Cost Calculation in Wastewater Treatment):** Allows for final values of one experiment to be used as initial values of another, hereby implementing a form of sequential execution of experiments that can be used for cost calculation (Gillot et al., 1999).

Figure 12.1 gives an overview of the relationships between the WEST++ macro-modules. It must be noted that during the development of WEST++, the concept of hierarchical virtual experimentation has first emerged. In WEST++, it has been implemented through re-usable virtual experiment macro-modules that communicate over the network. The rationale for this architecture was that in this way computational load caused by virtual experimentation could be divided over multiple hosts. However, in retrospect, the decision to split the Simul module from the Optim and SA modules was all but ideal. In the scope of Tornado-II, a different approach has therefore been taken.

Figure 12.2 contains a screenshot of the WEST++ GUI in action. The figure shows the layout of the Benchmark Simulation Model No. 1, which is loaded in the HGE module. Also, a dynamic simulation experiment based on this model is loaded in the Exp module and has been run to produce graphical data plots. In the upper left corner is the IMMC module, which contains a list of all active macro-modules.

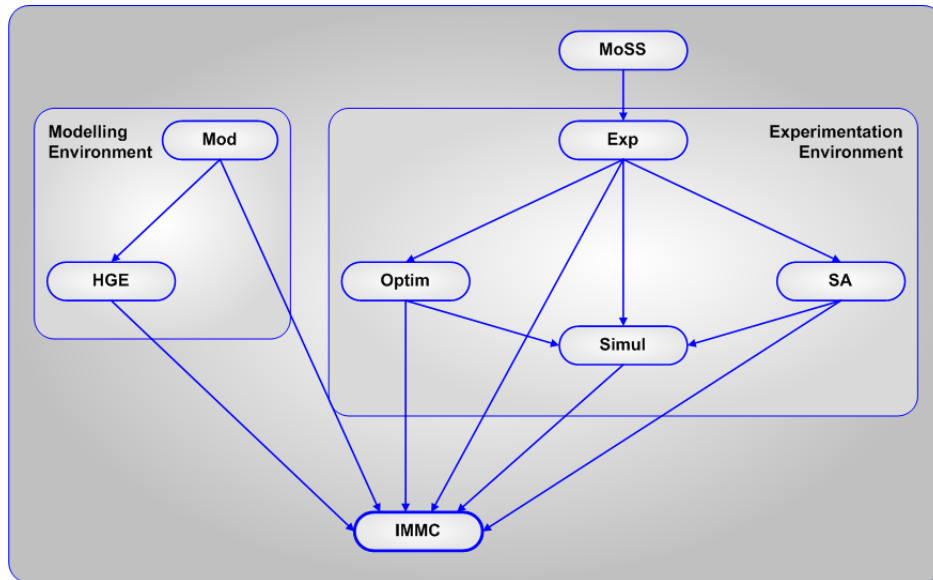


Figure 12.1: WEST++ Macro-Modules

12.2.2 WEST-3

WEST-3 (Vanhooren et al., 2003) is a commercial application² by MOSTforWATER N.V. (formerly HEMMIS N.V.). Its functionality is similar to its research-oriented counterpart, *i.e.*, WEST++. Main differences are the fact that WEST-3 supports an additional experiment type (for performing scenario analysis) and that user interface elements are available for model library browsing, model editing and Petersen matrix editing. On the other hand, WEST-3 is a Windows-only product, whereas WEST++ is portable across platforms. In contrast to WEST++, WEST-3 has a strong focus on water quality management; WEST++ is more generic.

As WEST++, WEST-3 relies on the Tornado-I kernel, although it uses a slightly modified version of it. However, in contrast to WEST++, WEST-3 does not use Tcl/Tk for its GUI, nor does it rely on socket-based communication between macro-modules. Instead, the WEST-3 GUI was developed using Borland Delphi³, the Microsoft COM⁴ middleware and the DevExpress⁵ and TeeChart⁶ widget libraries. In order to be able to use the Tornado-I native C++ API in a Delphi-based context, the Tornado-I kernel was wrapped as a DLL that exports a large number of C functions.

Figure 12.3 shows the WEST-3 Experimentation Environment GUI, again for the BSM1 model. The upper left part is taken by a tree-based model browser and a static graphical representation of the coupled model. Below and to the right are graphical plots of simulated data.

Figure 12.4 shows the WEST-3 model library browser and its associated model editor. The model editor supports syntax highlighting and code completion. Finally, Figure 12.5 shows model editing on the basis of the Petersen matrix representation.

²<http://www.mostforwater.com>

³<http://www.codegear.com/products/delphi/win32>

⁴<http://www.microsoft.com/com>

⁵<http://www.devexpress.com>

⁶<http://www.steema.com/products/teechart>

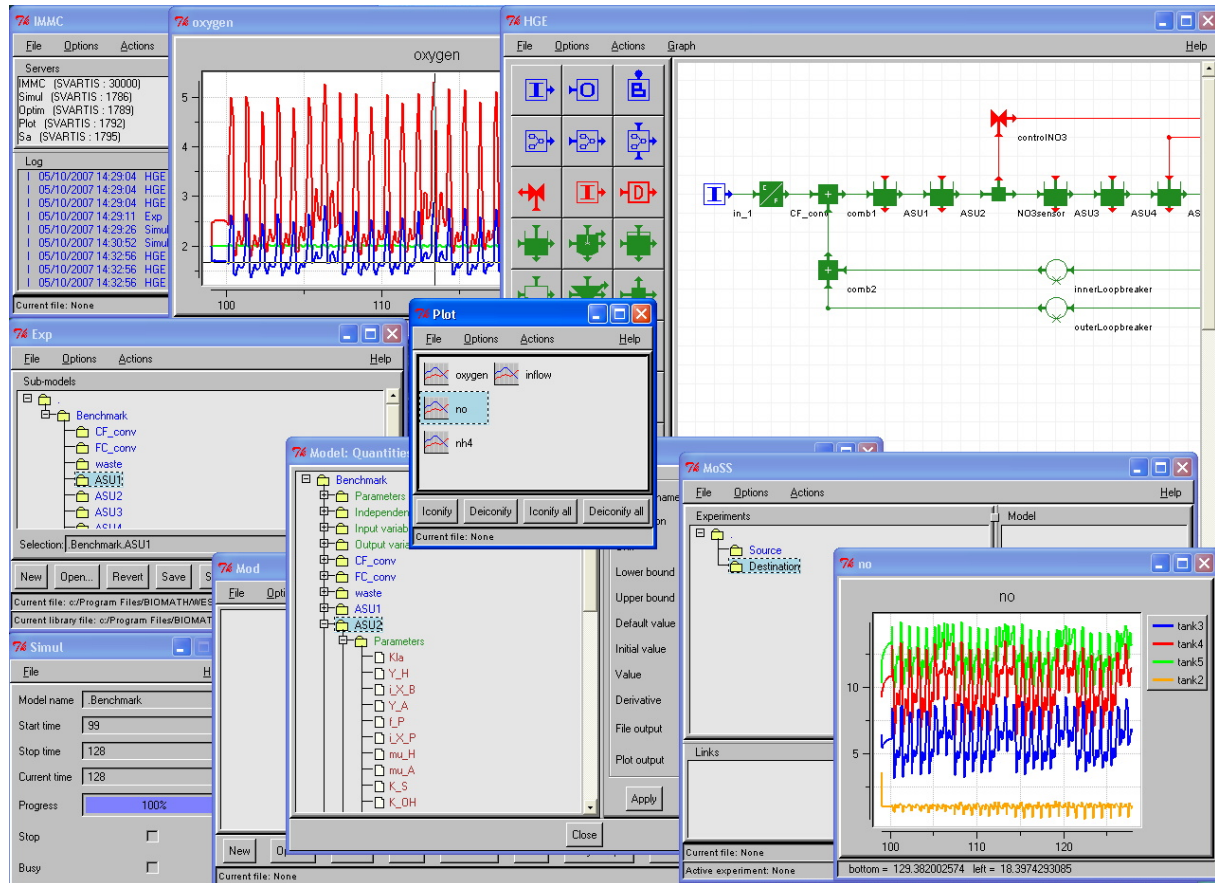


Figure 12.2: WEST++ GUI Applied to the BSM1 Case

12.2.3 EAST

EAST is another portable, research-oriented application that is based on the Tornado-I kernel. EAST is primarily intended to validate a software toolbox for optimal experimental design for the calibration of bioprocess models, as is extensively described in (De Pauw, 2005). It does not contain an environment for modelling, and focuses entirely on virtual experimentation.

In the scope of EAST, the original Tornado-I kernel was modified and integrated with a GUI based on the Qt⁷ toolkit. Qt is a powerful framework for high-performance, cross-platform application development. It includes a C++ class library and tools for cross-platform development and internationalization.

EAST was developed on top of the native C++ API of the Tornado-I kernel. Since Qt was used for GUI development, the static C++ libraries of the Tornado-I kernel could be used directly.

12.3 Tornado-II

12.3.1 MORE

MORE (Sin et al., 2007a,b) is a domain-specific tool that was developed to facilitate and speed up calibration and optimization of WWTP models. It uses a four-step procedure that is mainly based on Monte Carlo simulations:

⁷<http://trolltech.com/products/qt>



Figure 12.3: WEST-3 GUI Applied to the BSM1 Case

- Step 1: Selection of a parameter subset for calibration, based on identifiability analysis, experience supported with sensitivity analysis, or solely based on experience.
- Step 2: Definition of *a priori* distributions on the parameter subset, using experiences available in literature.
- Step 3: Execution of Monte Carlo simulations, including definition of the number of Latin Hypercube samples and the ranking of simulation results according to the weighted Sum of Squared Errors (SSE) criterion, describing the fit of the model to the data.
- Step 4: Evaluation of the Monte Carlo simulations, including the choice of the best model fit (based on the minimum weighted SSE) and an optional local parameter estimation.

MORE is being developed as a GUI on top of the native C++ interface of the Tornado-II kernel. It does not include a modelling environment and focuses entirely on virtual experimentation. For its virtual experimentation, it only uses a limited subset of the functionality offered by the Tornado-II kernel. Also, the MORE GUI is static, in the sense that the dynamic entity property querying mechanism of Tornado is not exploited. Instead, hard-coded dialog boxes are presented to the user to modify entity properties. As a result, the program works in an intuitive manner and follows the user's line of thought, but does not dynamically adapt to changes in the Tornado kernel.

Figure 12.6 shows the MORE GUI applied to a small Oxygen Uptake Rate (OUR) model. On the left, the figure shows data on objective values and simulated variable values. On the right, the simulation trajectories of a number of selected simulation runs are graphically depicted.

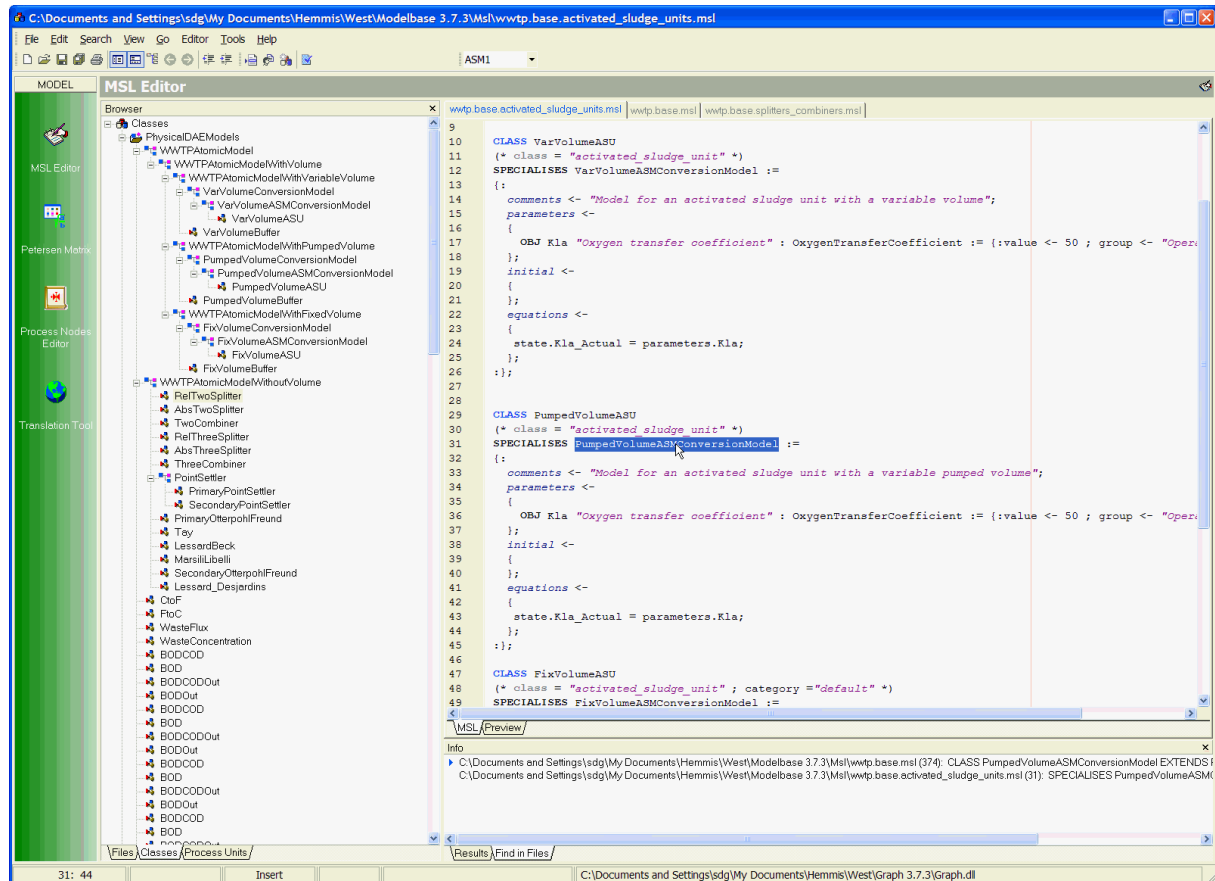


Figure 12.4: WEST-3 Model Library Browser and Model Editor

12.3.2 WEST-4

WEST-4 is the most recent and most innovative Tornado-based GUI development attempted so far. Upon completion, WEST-4 will have a modelling environment containing a model compiler, model builder, graphical layout editor, Petersen matrix editor, model library browser and model editor with syntax highlighting and code completion. The experimentation environment will support a large subset of the virtual experiment types that are implemented by the Tornado-II kernel.

WEST-4 is conceived as an Integrated Development Environment (IDE)⁸ for modelling and virtual experimentation. Its graphical appearance will have similarities with Microsoft's Visual Studio IDE platform. Thanks to extensive docking functionality, WEST-4's many windows can be positioned according to one's preference. Also, all dynamic aspects of Tornado in terms of dynamic entity property querying will be fully exploited by WEST-4. As a result, Tornado-II properties may be added, removed or modified without affecting the WEST-4 source code base.

12.8 shows the WEST-4 GUI applied to the BSM1 case. The view that is shown in this figure consists of the following windows:

- **Project Explorer:** Allows for browsing through the tree-based hierarchical structure of the experiments that are part of the project. In contrast to Tornado, WEST-4 forces every project to have exactly one layout. The number of experiments is unlimited however.

⁸http://en.wikipedia.org/wiki/Integrated_development_environment

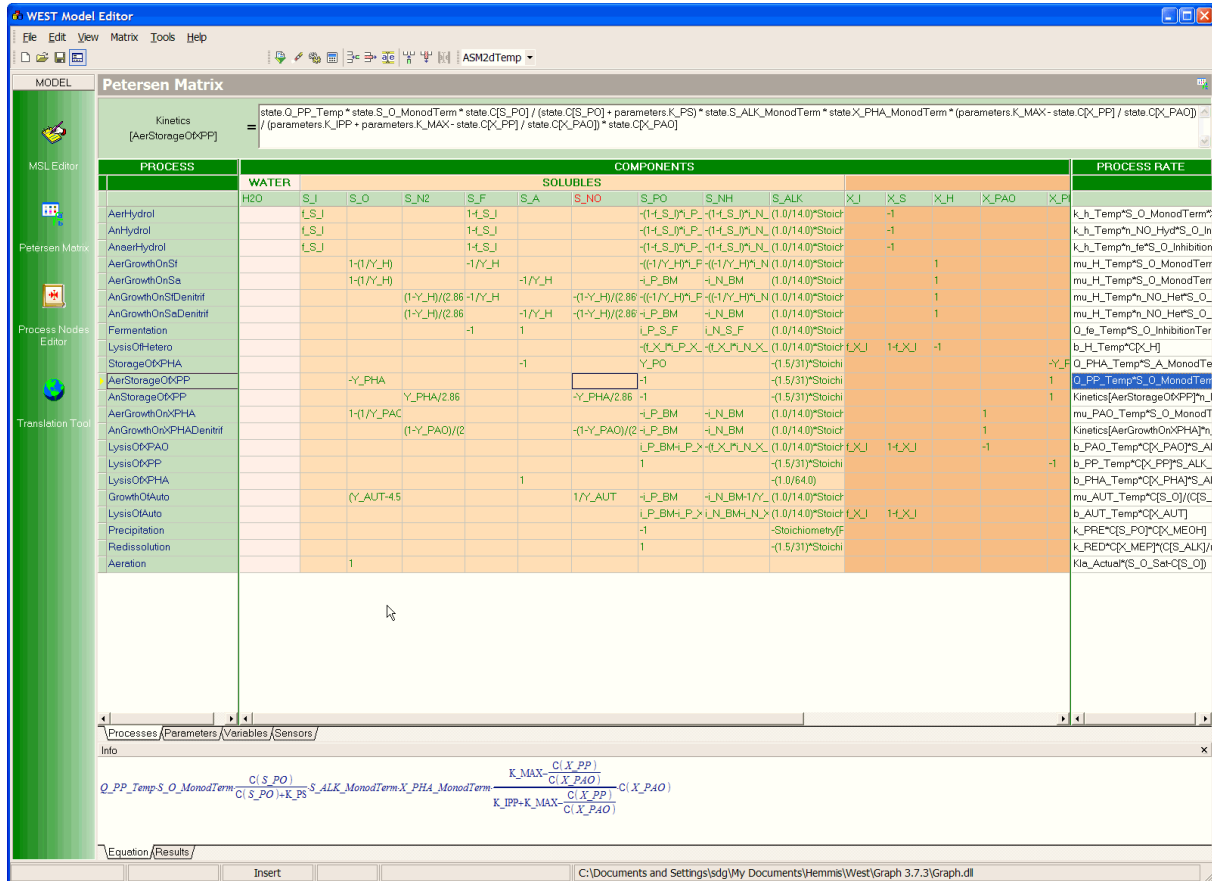


Figure 12.5: WEST-3 Petersen Matrix Editor

- **Property Sheet:** Displays the properties of the entity that is selected in the Project Explorer. Read-only properties can only be viewed, while read-write properties can be edited. The contents of this window is generated dynamically by querying the properties interface of the entity selected in the Project Explorer.
- **Message Log:** Displays error, warning and information messages that have been generated by the Tornado kernel and have been communicated to WEST-4 through the *CallbackMessage* callback interface.
- **Model Explorer:** Allows for browsing through the tree-based hierarchical structure of a layout. This view is an alternative for the graphical Model Layout view, which is not shown in the figure.
- **Model Information:** Displays symbolic information on the sub-model that is selected in the Model Explorer.
- **Experiment Control Center:** Allows for starting, stopping and pausing an experiment and monitoring its progress. Progress information is provided by the Tornado kernel through the *CallbackTime* and *CallbackRunNo* callback interfaces.

Figure 12.7 shows an excerpt of the WEST-4 GUI (Project Explorer and Property Sheet) for a project that contains three simulation experiments. The hierarchical structure of two experiments is collapsed, while the structure of the third is expanded. The Property Sheet shows the properties of the integration

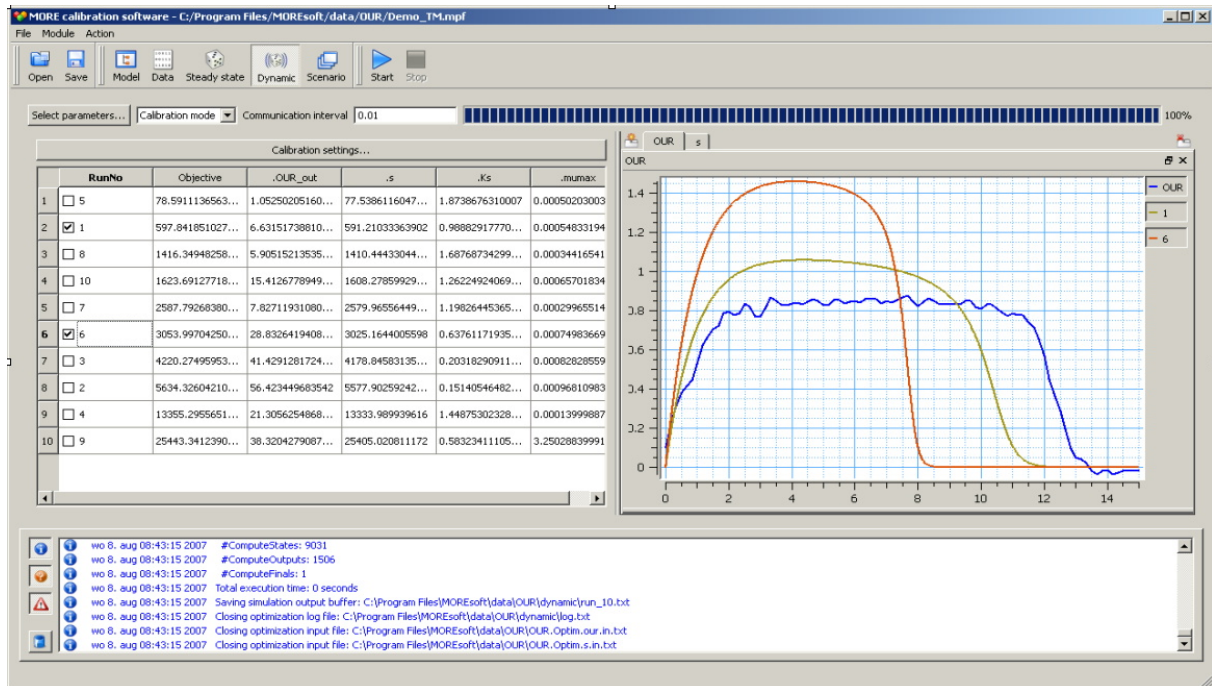


Figure 12.6: MORE GUI Applied to the OUR Case

solver of the latter experiment. This property information was retrieved at run-time from the integration solver plug-in.

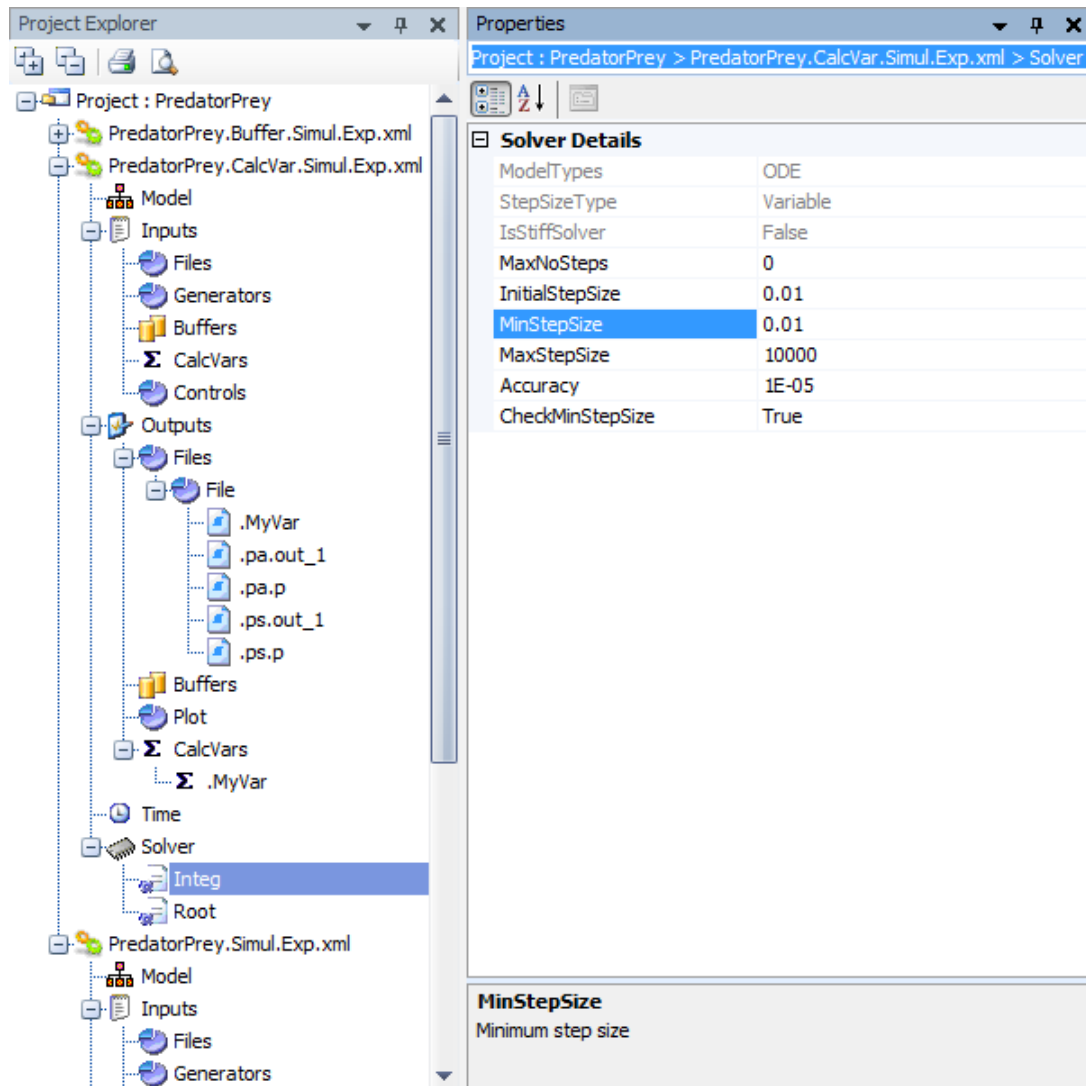


Figure 12.7: WEST-4 Project Explorer and Properties Sheet

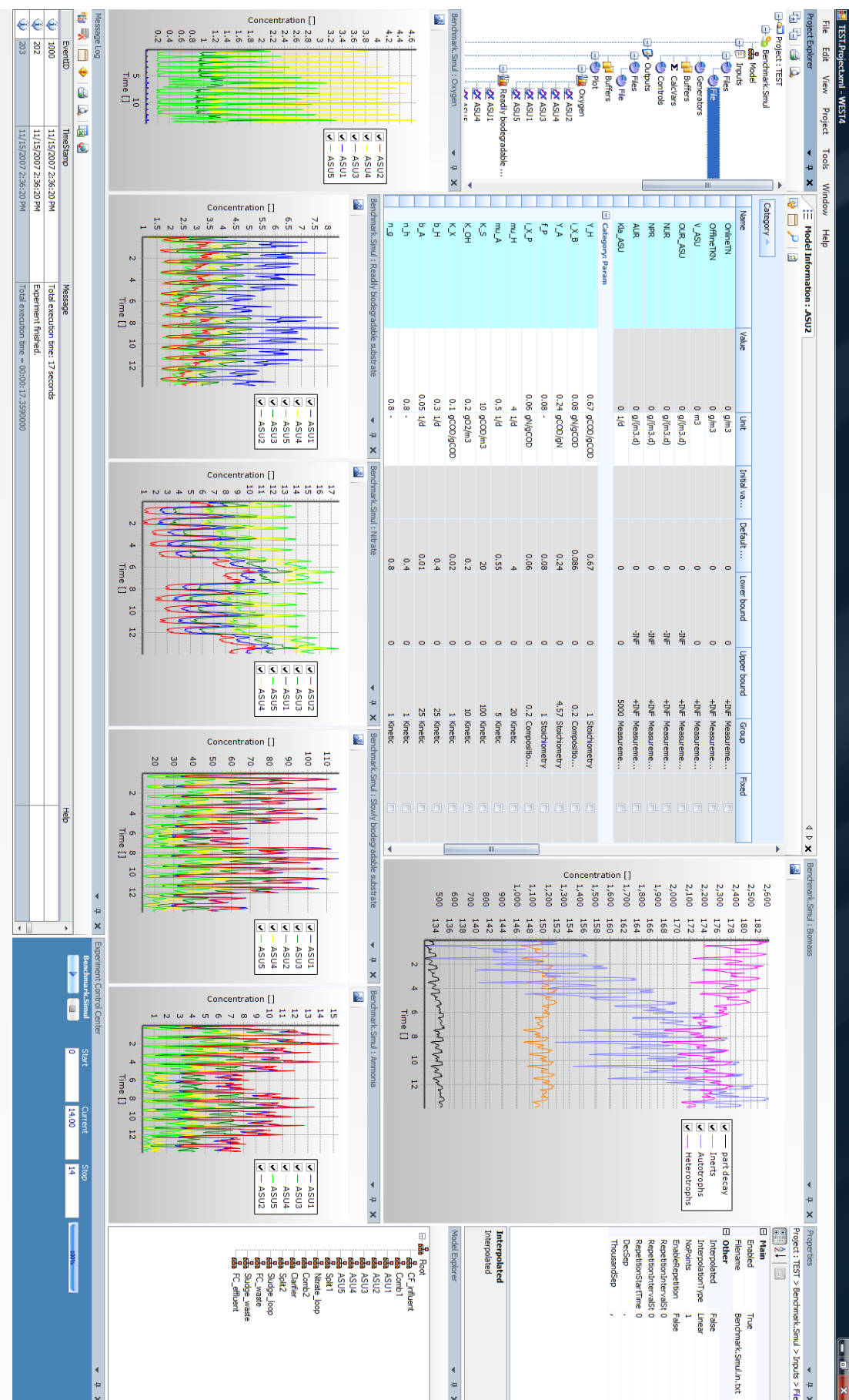


Figure 12.8: WEST-4 Applied to the BSM1 Case

13

Remote Execution

13.1 Introduction

For specific applications, remote execution of a kernel such as Tornado is required. Although *remote execution* and *distributed execution* are terms that are often used interchangeably, there is a clear distinction between both. In the case of remote execution of a task, the task will be executed as a whole in a remote location, typically on another computational node such as a PC, workstation, *etc.* In the case of distribution execution however, the task is first split into a number of parts that are then distributed over a number of computational nodes. For most of the sub-tasks that are created in this way, execution will be remote, in the sense that it will occur on a computational node that differs from the one from which the tasks originate. However, this does not necessarily have to be the case: some sub-tasks may be executed remotely while others are executed on the computational node from which they originate.

There are several scenarios in which remote execution of a kernel such as Tornado is desirable. Below are some of the most relevant cases for which this might be the case:

- **Utilization of remote computational power:** Many of the model compilation and virtual execution processes of Tornado are computationally intensive. One might therefore be interested in executing the Tornado kernel as a whole on a remote node, while keeping the user interface (graphical, command-line or other) of the application on one's own local node. Communication between user interface and remote kernel can be interactive, non-interactive (batch-oriented), or a mixture of both.
- **Automation:** In practice, many water quality systems are regulated by controllers and other devices. In some cases, control actions are decided upon through the automated execution and analysis of simulations and other types of virtual experiments (*cf.* Model Predictive Control (MPC)). One may decide to implement the execution of these virtual experiments on the control devices themselves, in which case the Tornado kernel would become embedded in the controller. However, one may also decide to run the Tornado kernel elsewhere and thus provide for remote execution.
- **Centralization:** In a situation where multiple users may wish to use the Tornado kernel from their applications, it might be decided to centralize the execution of the kernel. In case the kernel is run on a very powerful computational node, this would offer the evident advantage that multiple users can tap into the computational power that is provided by this central node. However, regardless

of the advantage a powerful central computational node may offer, the ability to maintain and administer a centralized Tornado kernel instead of multiple local copies may also prove to be an important advantage.

In computer technology, **middleware**¹ is computer software that connects software components or applications. This software consists of a set of enabling services that allow multiple processes running on one or more machines to interact across a network. Middleware technology evolved to provide for interoperability in support of the move to client/server architectures. It is used most often to support complex, distributed applications. It is adopted for web servers, application servers, content management systems, and similar tools that support application development and delivery. Middleware is especially integral to modern information technology based on XML, SOAP, Web services, and service-oriented architectures.

In the scope of the remote deployment of the Tornado kernel, various middleware technologies can potentially be adopted. The sequel gives an overview of some technologies that could be applied to Tornado, and some technologies that already have been applied to it.

13.2 Sockets

An Internet socket (or commonly, a socket or network socket), is a communication end-point unique to a machine communicating on an IP-based (Internet Protocol) network, such as the Internet. An Internet socket is composed of the following:

- Protocol
- Local IP address
- Local port
- Remote IP address
- Remote port

Operating systems combine sockets with a running process or processes (which use the socket to send and receive data over the network), and a transport protocol with which the process(es) communicate to the remote host. Two widely used Internet socket types are:

- Datagram sockets, which use UDP (User Datagram Protocol)²
- Stream Sockets, which use TCP (Transmission Control Protocol)³

In contrast to the use of TCP connections directly, sockets make a distinction between clients and servers, and are able to implement a queue of clients for a given server socket. Sockets are usually implemented by a library (such as the Berkeley sockets library on Unix or Winsock on Windows).

Socket-based communication has been applied to the Tornado-I kernel in the scope of the WEST++ application. In the case of WEST++, sockets are the support layer for the Inter Macro Module Communication protocol (*cf.* Chapter 12). They are therefore not really used for remote execution, but rather for a limited form of distributed execution.

¹<http://en.wikipedia.org/wiki/Middleware>

²http://en.wikipedia.org/wiki/User_Datagram_Protocol

³http://en.wikipedia.org/wiki/Transmission_Control_Protocol

In EAST, socket communication is used for the interaction between the EAST kernel and a LabVIEW setup, in the scope of an automated optimal experiment design mechanism (see also (De Pauw, 2005), Section 7.3).

The main advantages of the use of sockets are speed of communication and platform-independence. However, disadvantages are many, several of which are related to the fact that socket communication is a very low-level mechanism. It basically transfers byte sequences that have to be explicitly composed and analyzed by the application. Also in the case of Tornado-I, the low-level nature of socket communication has proven to be inconvenient as the complexity of the communication between modules increased.

13.3 CORBA

CORBA⁴ is a mechanism for normalizing the method-call semantics between application objects that reside either in the same address space (application) or in remote address spaces (either on the same host, or on a remote host on a network).

CORBA uses an Interface Definition Language (IDL) to specify the interfaces that objects will present to the outside world. CORBA then specifies a mapping from IDL to a specific implementation language such as C++ or Java. Standard mappings exist for Ada, C, C++, Lisp, Smalltalk, Java, COBOL, PL/I and Python. There are also non-standard mappings for Perl, Visual Basic, Ruby, Erlang, and Tcl implemented by object request brokers (ORB's) written for those languages.

The CORBA specification dictates that there shall be an ORB through which the application interacts with other objects. In practice, the application simply initializes the ORB, and accesses an internal Object Adapter which maintains such issues as reference counting, object instantiation policies, object lifetime policies, *etc.* The Object Adapter is used to register instances of the generated code classes. Generated code classes are the result of compiling the user IDL code which translates the high-level interface definition into an OS and language-specific class base for use by the user application. This step is necessary in order to enforce the CORBA semantics and provide a clean user process for interfacing with the CORBA infrastructure.

Some IDL language mappings are more hostile than others. For example, due to the very nature of Java, the IDL-Java Mapping is rather trivial and makes usage of CORBA very simple in a Java application. The C++ mapping is not trivial but accounts for all features of CORBA, including exception handling. The C mapping is even more obscure (since C is not an object-oriented language) but it is consistent and it correctly handles the RPC (Remote Procedure Call) semantics.

CORBA has several advantages, most importantly the fact that it is independent from the programming language that is used, as well as independent from the operating system that is adopted. CORBA is supported by a large consortium of companies and institutions and has been applied in the scope of extremely demanding applications. Also, the CORBA framework is very elaborate. It not only consists of a language mapping (IDL) mechanism and a data transport protocol, but also contains definitions of standardized network-based services.

The main advantages of CORBA (*i.e.*, its language and operating system independence and its elaborate framework) are indirectly also responsible for its biggest flaws. In principle, all implementations of the CORBA framework should be able to work together seamlessly through the definition of standardized interfaces. However, in practice almost no CORBA implementation fully complies with the standards defined by the OMG⁵ (Object Management Group). Since every CORBA implementation has its own holes and weaknesses, interoperability is not always guaranteed, especially when complex data types and invocation mechanisms are used.

⁴<http://www.corba.org>

⁵<http://www.omg.org>

In the scope of Tornado, CORBA has not yet been applied. Since in recent years CORBA has lost ground to more light-weight technologies such as SOAP, it also seems unlikely that a CORBA interface for Tornado will ever be developed. However, from a technical point of view such an implementation would certainly be feasible.

13.4 DCOM

The Distributed Component Object Model (DCOM)⁶ is a proprietary technology from Microsoft for software components, distributed across several networked computers, to communicate with each other. DCOM, which was originally named Network OLE, extends Microsoft's basic COM technology, and provides the communication substrate under Microsoft's COM+ application server infrastructure. It has now been deprecated in favor of Microsoft .NET. With respect to COM, DCOM had to solve two problems:

- **Marshalling:** Serializing and deserializing the arguments and return values of method calls “over the wire”.
- **Distributed garbage collection:** Ensuring that references held by clients of interfaces are released when, for example, the client process crashes, or the network connection is lost.

One of the key factors in solving these problems was the use of DCE/RPC as the underlying RPC mechanism behind DCOM. DCE/RPC has strictly defined rules regarding marshalling and the owner of the responsibility for freeing memory.

DCOM was originally a major competitor to CORBA. Proponents of both of these technologies saw them as one day becoming the model for code and service re-use over the Internet. However, the difficulties involved in getting either of these technologies to work over Internet firewalls, and on unknown and insecure machines, meant that normal HTTP⁷ (Hyper Text Transfer Protocol) requests in combination with web browsers won out over both of them.

In the scope of Tornado-I, DCOM technology is used to make the WEST-3 back-end available to custom applications through a remote API.

13.5 OPC

OLE⁸ for Process Control (OPC)⁹ is the original name for an open standard specification developed in 1996 by an automation industry task force. The standard specifies the communication of real-time plant data between control devices from different manufacturers.

After the initial release, the OPC Foundation was created to maintain the standard. Since then, standards have been added and names have been changed. While OPC originally stood for “OLE for Process Control”, the official stance of the OPC Foundation is that OPC is no longer an acronym and the technology is simply known as “OPC”.

The OPC Specification was based on the OLE, COM, and DCOM technologies developed by Microsoft for the Microsoft Windows operating system family. The specification defined a standard set of objects, interfaces and methods for use in process control and manufacturing automation applications to facilitate interoperability.

⁶<http://www.microsoft.com/com>

⁷<http://en.wikipedia.org/wiki/HTTP>

⁸OLE refers to Microsoft's Object-Linking and Embedding technology

⁹<http://www.opcfoundation.org>

OPC was designed to bridge Windows-based applications and process control hardware and software applications. It is an open standard that permits a consistent method of accessing field data from plant floor devices. This method remains the same regardless of the type and source of data.

OPC servers provide a method for many different software packages to access data from a process control device, such as a Programmable Logic Controller (PLC) or Distributed Control System (DCS). Traditionally, any time a package needed access to data from a device, a custom interface, or driver, had to be written. The purpose of OPC is to define a common interface that is written once and then re-used by any Supervisory Control and Data Acquisition (SCADA), Human-Machine Interface (HMI), or custom software package. Once an OPC server is written for a particular device, it can be re-used by any application that is able to act as an OPC client.

Although OPC is an open standard, Software Development Kits (SDK's) for this protocol are not freely available. This is the main reason why to date no OPC interface for Tornado exists, in spite of the very important benefit of being able to couple Tornado to a wide range of OPC-compliant devices. As for the case of CORBA, no technical difficulties are expected in case the development of a Tornado OPC interface would be initiated.

13.6 Java RMI

Java Remote Method Invocation, or Java RMI¹⁰, is a Java application programming interface for performing the object equivalent of remote procedure calls.

There are two common implementations of the API. The original implementation depends on Java Virtual Machine (JVM) class representation mechanisms and thus only supports calls from one JVM to another. The protocol underlying this Java-only implementation is known as the Java Remote Method Protocol (JRMP). In order to support code running in a non-JVM context, a CORBA version was later developed. Usage of the term RMI may denote solely the programming interface or may signify both the API and JRMP, whereas the term RMI-IIOP denotes the RMI interface delegating most of the functionality to the supporting CORBA implementation.

The original RMI API was generalized somewhat to support different implementations, such as HTTP transport. Additionally, work was done to CORBA, adding a pass-by-value capability, to support the RMI interface. Still, the RMI-IIOP and JRMP implementations are not fully identical in their interfaces.

Because of its strong relationships to Java and CORBA, technologies that are normally not used in the scope of Tornado, no attempt has thus far been made to apply Java RMI to the Tornado kernel. However, since Tornado is already available from Java through TornadoJNI, it is assumed that it could easily be called through Java RMI.

13.7 SOAP

SOAP¹¹ originally stood for Simple Object Access Protocol, and lately also Service Oriented Architecture Protocol. The original acronym was dropped with version 1.2 of the standard, which became a World Wide Web Consortium (W3C) Recommendation on June 24, 2003, as it was considered to be misleading.

SOAP was originally designed by Dave Winer, Don Box, Bob Atkinson, and Mohsen Al-Ghosein in 1998, with support from Microsoft, as an object-access protocol. The SOAP specification is currently maintained by the XML Protocol Working Group of the W3C.

¹⁰<http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp>

¹¹<http://www.w3.org/TR/soap>

SOAP is a protocol for exchanging XML-based messages over computer networks, normally using HTTP / HTTPS ((Secure) Hyper Text Transfer Protocol). SOAP forms the foundation layer of the Web Services stack, providing a basic messaging framework that more abstract layers can build on.

There are several different types of messaging patterns in SOAP, but by far the most common is the Remote Procedure Call (RPC) pattern, in which one network node (the client) sends a request message to another node (the server) and the server immediately sends a response message to the client.

In recent years, SOAP has substantially gained popularity because of its relative simplicity. In contrast to CORBA and DCOM, SOAP does not rely on binary data representations but rather on text-oriented XML representations transferred through HTTP(S). Consequently, it usually works well in combination with firewalls.

As will be discussed in Chapter 14, SOAP has been used in conjunction with Tornado-II in the scope of distributed execution. In fact, a light-weight kernel for distributed execution, named Typhoon, was built for use with Tornado, using the same design principles as Tornado and using SOAP as a middleware.

13.8 .NET Remoting

.NET Remoting is a Microsoft application programming interface for inter-process communication released in 2002 with the first full version of the .NET framework. It is one in a series of Microsoft technologies that began in 1990 with the first version of Object Linking and Embedding (OLE) for 16-bit Windows. Intermediate steps in the development of these technologies were the Component Object Model (COM), released in 1993, and the Distributed Component Object Model (DCOM), released in 1997. .NET Remoting is restricted only to .NET client applications, so it is not platform-independent.

Analogous to similar technologies such as CORBA and Java RMI, .NET Remoting is complex, yet its essence is straightforward. With the assistance of operating system and network agents, a client process sends a message to a server process and receives a reply. A major advantage of .NET Remoting is that only few source code modifications are needed for existing .NET classes to become remotely callable.

After having provided Tornado with a .NET API as discussed in Chapter 10, only a minor effort was required to make this API remotely accessible through .NET remoting. As mentioned before, the main elements of the .NET Tornado interface are calls, callbacks (events) and exceptions. In order to be able to make remote calls to the API, all API classes had to be derived from the *System::MarshalByRefObject* system base class. Also event and event handler classes had to be derived from this base class. Most time had to be spent on exception classes. In order for these to be usable in a .NET Remoting context, the *[System::Serializable]* attribute had to be added to their declaration. Also, a deserialization constructor and implementation of the *GetObjectData* method needed to be implemented. Lastly, filtering for custom exceptions had to be switched off in the .NET application configuration files.

Listing 13.1 shows the source code of an application that can execute, either locally or remotely (through .NET Remoting), a virtual experiment passed as a command-line argument. The difference between both modes of execution in terms of source code is negligible: in case of remote execution, a .NET XML application configuration file (describing the Tornado server to connect to) is to be loaded; in case of local execution, no such file is required. An example of a client application configuration file is shown in Listing 13.2.

Listing 13.3 shows the source code of a server application that exposes the Tornado kernel as an object to the outside world (through .NET Remoting). In fact, the application by itself is generic since the object that is to be exposed is described by the .NET application configuration file that goes with this application. The application code itself does not contain any reference to the exposed object. An example of a server application configuration file for Tornado is shown in Listing 13.4.

Listing 13.1: .NET Remoting Tornado Client

```

using System;

namespace TornadoNET
{
    class CHandler : System.MarshalByRefObject
    {
        public void
        SetMessage(int Type,
                   string Message)
        {
            switch (Type)
            {
                case 0:
                    System.Console.WriteLine(Message);
                    break;

                case 1:
                    System.Console.WriteLine(Message);
                    break;

                case 2:
                    System.Console.WriteLine(Message);
                    break;
            }

            System.Console.WriteLine(Message);
        }

        public void
        SetTime(double Time)
        {
            System.Console.WriteLine("Time=" + Time);
        }
    }

    class CClient
    {
        public void
        Run(string MainFileName,
            string ExpFileName)
        {
            try
            {
                CHandler MyHandler = new CHandler();

                TornadoNET.CTornado Tornado = new TornadoNET.CTornado();

                Tornado.EventSetMessage +=
                    new TornadoNET.CTornado._Delegate_EventSetMessage(MyHandler.SetMessage);

                Tornado.Initialize(MainFileName, true);

                TornadoNET.CExpSimul ExpSimul = (CExpSimul)Tornado.ExpLoad(ExpFileName);

                ExpSimul.EventSetMessage +=
                    new TornadoNET.CExp._Delegate_EventSetMessage(MyHandler.SetMessage);
                ExpSimul.EventSetTime +=
                    new TornadoNET.CExp._Delegate_EventSetTime(MyHandler.SetTime);

                ExpSimul.Initialize();
                ExpSimul.Run();
            }
            catch (TornadoNET.Util.CEx Ex)
            {
                System.Console.WriteLine(Ex.ToString());
            }
        }
    }

    class texecNET
    {

```



```

static public void
Main( string [] args )
{
    try
    {
        string ConfigFileName = "texecNET.Config.xml";
        string MainFileName = "Tornado.Main.xml";
        string ExpFileName = args [1];
        string Mode = args [2];

        // If needed, load application configuration

        if (Mode == "Remote")
            System.Runtime.Remoting.RemotingConfiguration.Configure( ConfigFileName , false );

        // Instantiate and run client

        CClient Client = new CClient ();
        Client.Run(MainFileName , ExpFileName );
    }
    catch (System.Exception Ex)
    {
        System.Console.WriteLine(Ex);
    }
}
}

```

Listing 13.2: .NET Remoting Tornado Client Configuration File

```

<configuration>
  <system.runtime.remoting>
    <customErrors mode="off" />
    <application>
      <client>
        <wellknown type="TornadoNET.CTornado, ~TornadoNET"
          url="http://localhost:10000/Tornado.rem" />
      </client>
      <channels>
        <channel ref="http" port="0">
          <clientProviders>
            <formatter ref="binary" />
          </clientProviders>
          <serverProviders>
            <formatter ref="binary" typeFilterLevel="Full" />
          </serverProviders>
        </channel>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>

```

Listing 13.3: .NET Remoting Tornado Server

```

using System;

namespace TornadoNET
{
    public class tservNET
    {
        public static void
        Main( string [] args )
        {
            try
            {
                string ConfigFileName = "tservNET.Config.xml";

                // Load application configuration

                System.Runtime.Remoting.RemotingConfiguration.Configure( ConfigFileName , false );
            }
        }
    }
}

```

```

        Console.WriteLine ("Listening_for_requests ...");

        while (true)
            Console.ReadLine();
    }
    catch (System.Exception Ex)
    {
        System.Console.WriteLine(Ex);
    }
}
}
}

```

Listing 13.4: .NET Remoting Tornado Server Configuration File

```

<configuration>
  <system.runtime.remoting>
    <customErrors mode="off" />
    <application>
      <service>
        <wellknown mode="Singleton" type="TornadoNET.CTornado, _TornadoNET"
          objectUri="Tornado.rem" />
      </service>
      <channels>
        <channel ref="http" port="10000">
          <serverProviders>
            <provider ref="wsdl" />
            <formatter ref="soap" typeFilterLevel="Full" />
            <formatter ref="binary" typeFilterLevel="Full" />
          </serverProviders>
          <clientProviders>
            <formatter ref="binary" />
          </clientProviders>
        </channel>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>

```

13.9 ASP.NET

ASP.NET¹² is a web application framework by Microsoft that can be used by developers to build dynamic web sites, web applications and XML web services. It is part of Microsoft's .NET platform and is the successor to the Active Server Pages (ASP) technology. ASP.NET is built on the Common Language Runtime (CLR), allowing programmers to write ASP.NET code using any Microsoft .NET language.

ASP.NET technology has been tested on Tornado through the development of a small Tornado-based web service and a number of client programs. Both service and clients were programmed in C#. The service allows for initializing the Tornado kernel, loading an experiment, getting and setting variables and executing the experiment. All required input resources (input data files, experiment descriptions, executable model, *etc.*) can be transferred from client to server through Base64¹³-based serialization. In the same way, output resources (output data files, *etc.*) can be transferred from service to client.

An ASP.NET web service in C# can be built by simply prefixing methods that should be remotely accessibly by the *[WebMethod]* attribute. Further, so-called **session** and **application** storage is to be used to store global data. Application storage remains valid during the entire lifetime of the service, while session storage is only valid during a particular session.

¹²<http://msdn.microsoft.com/asp.net>

¹³<http://en.wikipedia.org/wiki/Base64>

13.10 Conclusion

Above, several technologies have been discussed that can potentially be applied to provide for remote accessibility of the Tornado kernel. All of these technologies have their own advantages and disadvantages in terms of efficiency, platform-independence and easy of use, and none of the technologies can be considered to be generally widespread. It is therefore important for Tornado to be able to “adapt” to the specific remote execution technology that is required for a certain application and/or a certain context. As has been discussed, several technologies have been applied to Tornado in its two incarnations. Other technologies have not yet been applied, but in case this would be required, an integration of Tornado with these technologies is expected to be realizable without major difficulties.

Table 13.1 gives an overview of the technologies discussed. For each of the technologies that have already been applied to Tornado, it is mentioned whether the technology was integrated with Tornado-I or Tornado-II. Also, it is mentioned in the scope of which application this was done, and to which extent. In the case of a comprehensive integration, the complete Tornado functionality is remotely available. In the case of a restricted integration, only a restricted subset of functionalities are available (more specifically: execution of virtual experiments).

Table 13.1: Relevant Remote Execution Technologies and their Application to Tornado

Technology	Tornado Kernel Incarnation	Tornado-based Application	Extent of Application
Sockets	Tornado-I	WEST++, EAST	Restricted
CORBA	-	-	-
DCOM	Tornado-I	WEST-3	Comprehensive
OPC	-	-	-
Java RMI	-	-	-
SOAP	Tornado-II	Typhoon	Restricted
.NET Remoting	Tornado-II	-	-
ASP.NET	Tornado-II	-	-

14

Distributed Execution

14.1 Introduction

Virtual experimentation with water quality models is a complex and computationally intensive process, which frequently requires 100's or 1000's of simulation runs. A continuous need for software tools that offer extended functionality and improved performance therefore exists. The Tornado-II kernel manages to substantially improve performance with respect to its predecessors, and offers a broad range of virtual experiment types to solve various frequently occurring problems. However, notwithstanding the inherent efficiency of Tornado, situations still abound where complex problems cannot be solved on one single workstation in a reasonable amount of time. In order to alleviate the computational burden, distributed execution is a requirement. Fortunately, many complex problems in Tornado are composed of loosely coupled sub-problems. Coarse-grained gridification (*i.e.*, splitting a compound problem into a number of high-level constituents, to be executed on different computational nodes), is therefore relatively easy to accomplish.

A large variety of distributed execution environments is available nowadays, ranging from elaborate inter-organizational grid infrastructures (*e.g.*, gLite, Condor-G, UNICORE, ...¹) to more localized cluster solutions (*e.g.*, Torque, (Open)PBS, ...²). Solutions based on the migration of processes at the operating system level (*e.g.*, (Open)MOSIX³) and solutions that are specific for a particular application (*e.g.*, the MATLAB distributed execution toolbox) also exist. Although the number of available solutions is large, only a few are truly mature, and transparency and ease of use are often limited. In fact, the world of distributed execution is still highly dynamic and all solutions have their own particular pros and cons. For instance, MOSIX offers a high degree of transparency, but is restricted to one particular platform (*e.g.*, Linux) and is not well-suited for the distribution of data-intensive tasks. On the other hand, the disadvantage of many grid and cluster solutions is that they are often heavily command-line oriented (and hence unsuitable for many types of users), highly convoluted, and again not portable across platforms.

The Tornado user community is very diverse. It consists of a small group of experts (typically found in academia) who have the knowledge to construct atomic models of unit processes, coupled models of plant layouts, and virtual experiments. A second, larger group is mainly found among engineering and consultancy companies. Members of this group will also create coupled models and set up virtual exper-

¹<http://gridcafe.web.cern.ch/gridcafe>

²<http://www.clusterresources.com>

³<http://www.mosix.org>

iments, however they will not construct atomic models and will merely re-use those that are provided by members of the first group. The potentially largest user group is composed of plant operators, who will mainly only run ready-made virtual experiments through simplified user interfaces.

Since, on the one hand, all distributed execution environments have their own pros and cons, and on the other hand, it is very difficult at this stage to predict which solutions will prevail in the end, it would not be wise to graft distributed execution in Tornado to one particular distributed execution environment. A more advisable approach is to introduce a layer of abstraction that hides the details of the actual environment that is being used. Moreover, given the heterogeneity of the Tornado user community, no assumptions can be made about the level of expertise of its users. For instance, one cannot expect users to manually write shell scripts and embed them in job descriptions that are subsequently submitted using a grid infrastructure's command-line tools. The abstraction layer should therefore allow for seamless integration of the distributed execution infrastructure into the application. Ultimately, users should not see the difference between jobs executed locally and remotely.

At this moment, Tornado has an abstraction layer and a generic job description format that allows for two distributed execution environments to be utilized: gLite⁴ and Typhoon (formerly known as WDVE) (Chепен, 2004). Unfortunately, only semi-automatic processing is possible at this point, but this type of processing does constitute an important first step towards transparent distributed execution.

Parts of this chapter have previously been published in (Claeys et al., 2006a) and (Claeys et al., 2006c).

14.2 Gridification of Virtual Experiments

When faced with the computational complexity of modelling and virtual experimentation kernels such as Tornado, there are in general two approaches that can be followed with regard to gridification: either fine-grained gridification at the level of models (distributed simulation (Fujimoto, 2000), *i.e.*, splitting a model in different parts that are all executed on a different computational node), or coarse-grained gridification at the level of virtual experiments (*i.e.*, splitting an experiment into sub-experiments that are executed on different computational nodes). The application of the first approach in the scope of Modelica is discussed in (Aronsson and Fritzson, 2002) and (Aronsson, 2006) on the basis of a tool that adopts a number of scheduling and clustering techniques to partition Dymola-generated simulation code onto several processors. It can intuitively be understood that fine-grained gridification is a fairly complex and convoluted approach that will only yield good results in case of large models in which loosely coupled components can be identified. Since Tornado models usually do not contain large, loosely coupled components, fine-grained gridification has not been withheld as an option. On the other hand, given the inherent hierarchical nature of compound virtual experiments, coarse-grained gridification is relatively easily applicable.

When investigating the dependency relationships between the components of compound virtual experiments in Tornado, two classes can be identified. The first class of experiments (ExpScen, ExpMC, ExpSens, ExpEnsemble, ExpScenSSRoot) is based on the execution of a number of simulations that are basically independent. In the other class of experiments (ExpOptim, ExpCI) a simulation can only be run after a number of other simulations have been executed, hence leading to a sub-experiment dependency relationship. The ExpScenOptim and ExpMCOptim experiments are special in the sense that they belong to the first class when optimizations are considered as the smallest undividable unit of work. In case however simulations are considered to be smallest undividable unit, they belong to the second class. As an example, the flow of execution of the ExpOptim and ExpScen experiment types is represented in Figure 14.1. The meaning of the tasks mentioned in this figure is as follows:

⁴<http://glite.web.cern.ch/glite/>

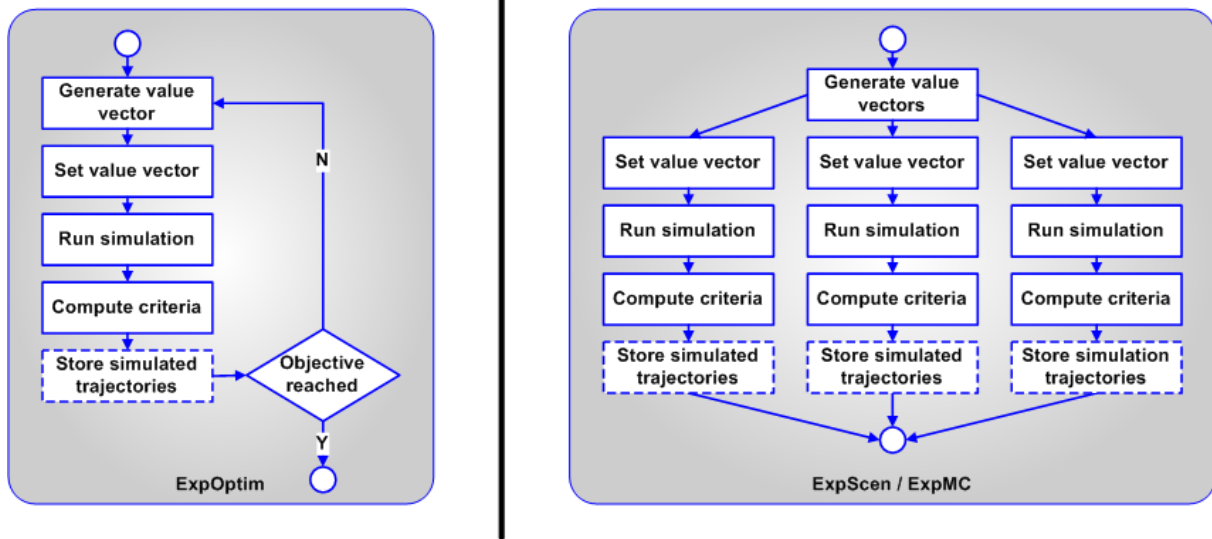


Figure 14.1: Flow of Execution of the ExpOptim and ExpScen Experiment Types

- **Generate value vector:** Create a vector composed of initial values for a number of identified objects (typically model parameters and/or initial conditions)
- **Set value vector:** Apply previously generated initial values to their corresponding objects
- **Run simulation:** Run a simulation using the newly applied initial values
- **Compute criteria:** Compute a number of criteria (objective values) from the simulated trajectories
- **Store simulated trajectories:** Optionally store the simulated trajectories of a number of identified model quantities for future reference

As a first step towards distributed execution in Tornado, a generic job specification format was produced, taking the following considerations into account: simplicity, maintainability, extensibility, application-independence, platform-independence, and support for the notion of sets of related jobs. The format that was produced is XML-based and is known as the Typhoon format. It is loosely based on the Global Grid Forum's JSDL format (Job Submission Description Language⁵). Actually, JSDL was still under development at the time the Typhoon format was defined. Only recently, JSDL has been put forward as a recommended standard.

A graphical representation of the XSD description of the Typhoon job specification is shown in Figure 14.2. From the figure follows that a *Jobs* entity consists of a number of properties and a set of *Job* entities. These *Job* entities are further composed of sets of input and output resources, and a structure that describes the application that the Job is related to. Input resources are items (typically files) that need to be transferred to the remote execution node in order to be able to execute the job. Output resources are items generated during job execution that need to be transferred from the remote execution node back to the user's workstation. The structure of the *App* (Application) entity is application-dependent. In the case of Tornado, it consists of a reference to the input resource that should be considered as the start-up Tornado XML experiment description, and a list of initial values that are to be applied to the experiment loaded (in fact these take precedence over the initial values that are specified in the experiment description itself).

⁵<https://forge.gridforum.org/projects/jsdl-wg>

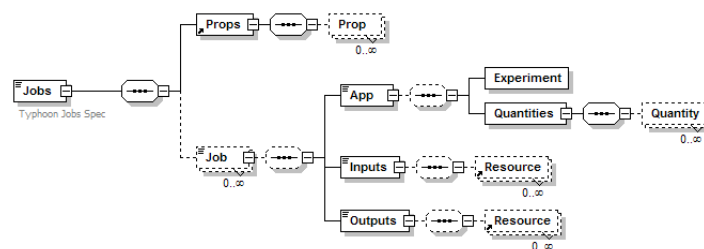


Figure 14.2: Typhoon Job XML Schema Definition

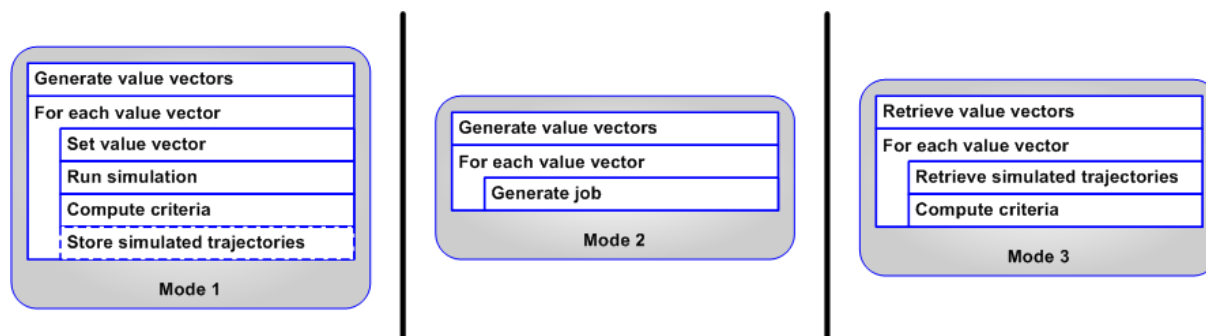


Figure 14.3: Modes of Operation for ExpScen/ExpMC

As a second step towards distributed execution, the implementations of relevant compound experiments (*i.e.*, experiments that rely on independent sub-experiments) were modified in order to support 3 modes of operation:

- **Mode 1** executes the experiment in the classical, non-distributed fashion: first pre-processing is performed, followed by sequential execution of sub-experiments, and finally post-processing.
- **Mode 2** also first performs pre-processing, but then simply generates job descriptions for each sub-experiment, to be executed by a distributed execution environment.
- **Mode 3** only performs post-processing on sub-experiment data that was generated beforehand.

For ExpScen and ExpMC, Figure 14.3 shows the tasks that are performed during the various modes of operation, in the form of Nassi-Schneiderman diagrams⁶. In the case of distributed execution, experiments are first to be run in Mode 2, followed by processing of all generated job descriptions in a distributed execution environment, and completed with a run of the same experiment in Mode 3.

Figure 14.4 represents the types of nodes that are commonly found in grid and cluster systems. Jobs generated on the user's workstation (WS) are transferred to the distributed execution environment's user interface (UI), from where they are submitted to a resource broker (RB). The resource broker subsequently assigns jobs to computational elements (CE), using a particular scheduling policy. Input resources usually travel from WS to UI, from where they are uploaded to a storage element (SE). CE's will retrieve the input resources required for the execution of jobs from the SE. Output resources follow an inverse path: they are uploaded to the SE from CE's and are then transferred to the WS through the UI.

Figure 14.5 represents the relationship between the various modes of operation of an experiment (again taking ExpScen/ExpMC as an example) and these typical grid nodes.

⁶http://en.wikipedia.org/wiki/Nassi-Shneiderman_diagram

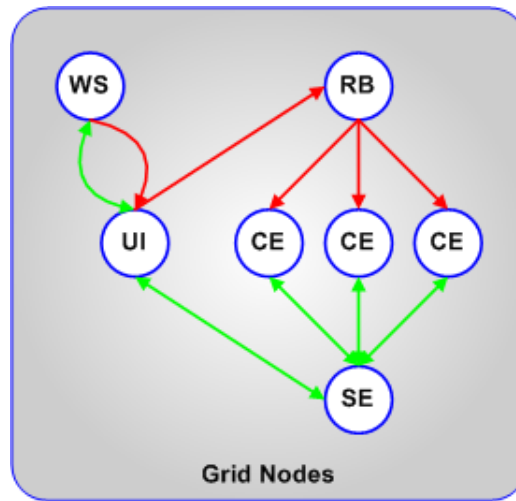


Figure 14.4: Grid Node Types

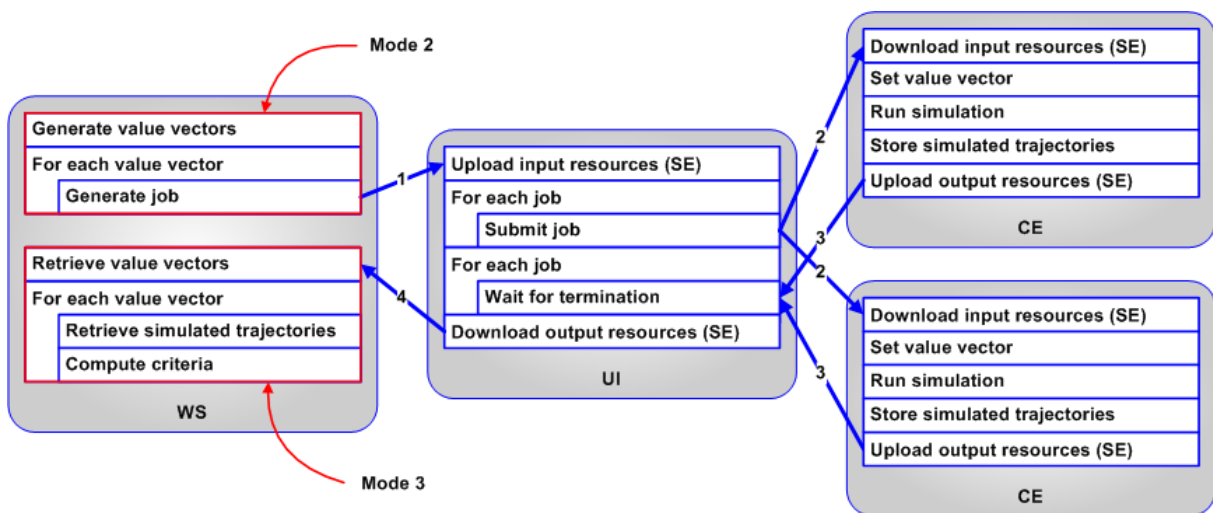


Figure 14.5: Relationship between Distributed Execution and Modes of Operation

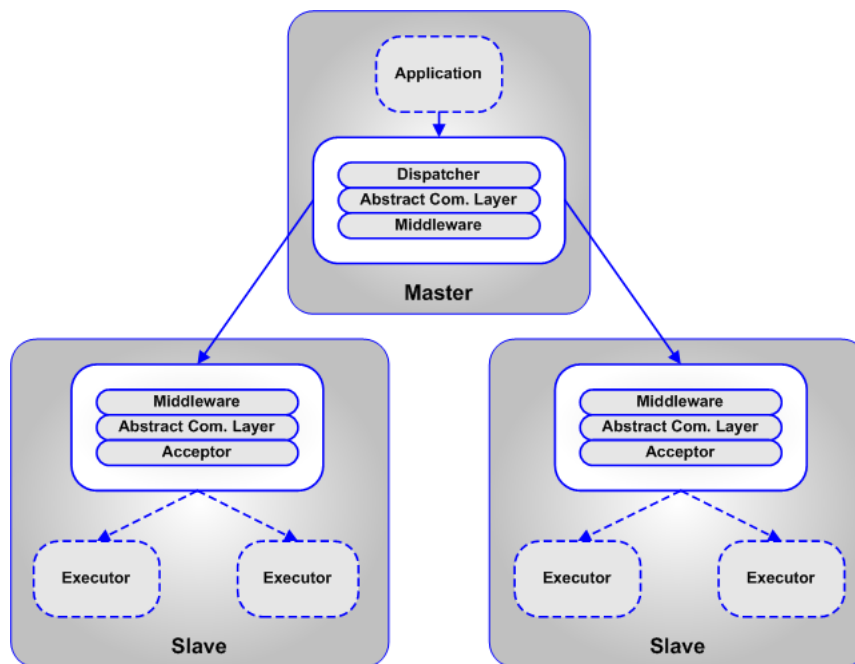


Figure 14.6: High-level Typhoon Architecture

The generation of job descriptions is facilitated by the nature of the generic interface that all experiment types have to comply with. This interface not only allows for initializing and running an experiment, it also provides a way to set and retrieve values of named objects, and methods for retrieving a list of all input resources required and a list of all outputs provided. When calling the two last methods on a compound experiment, the union of all input or output resources related to the experiment itself and each of its sub-experiments is returned.

14.3 Typhoon

Typhoon is a portable, light-weight distributed execution environment for clusters that was developed in the wake of Tornado, using technologies and design principles similar to Tornado itself. It consists of two types of components: a Master that directly interprets Typhoon XML job descriptions and provides a dynamic scheduler, and Slaves that execute jobs handed over by the Master. Next to generic (*i.e.*, application-independent) information on input and output resources, Slaves also receive the application-specific section of job descriptions (*i.e.*, the XML content below the *App* tag in Figure 14.2). In order to parse and interpret this application-specific information, application-specific executor plug-ins can be loaded into each Slave. The most prominent executor plug-in available for Typhoon allows for the execution of Tornado virtual experiment descriptions.

Figure 14.6 presents a high-level view on the architecture of Typhoon. A Master's Dispatcher sends jobs to one or more Slave Acceptors, located on different computational elements, through communication middleware. Dispatcher and Acceptors do not directly interact with the communication middleware, but rather use an abstraction layer that hides the details of the middleware. In this way, the possibility remains to adopt a different middleware without being forced into a substantial re-implementation of the Typhoon kernel.

Typhoon was developed in C++ using the Tornado design principles, and currently uses the SOAP

protocol for its Master-Slave communication. More specifically, the gSOAP⁷ (van Engelen and Gallivan, 2002) toolkit is adopted. This toolkit offers an XML to C/C++ language binding to ease the development of SOAP/XML Web services. It provides a transparent SOAP API through the use of proven compiler technologies. These technologies leverage strong typing to map XML schemas to C/C++ definitions. Strong typing provides a greater assurance on content validation of both WSDL schemas and SOAP/XML messages. The gSOAP compiler generates efficient XML serializers for native and user-defined C and C++ data types. As a result, SOAP/XML interoperability is achieved with a simple API relieving the user from the burden of WSDL and SOAP details, thus enabling him or her to concentrate on the application-essential logic.

A more detailed view on the Typhoon architecture is given in Figure 14.7. It shows that representations of Jobs provided by an application are wrapped by a JobDispatcher entity and registered with the Dispatcher. The Dispatcher tries to allocate jobs to one of the Slaves that have registered with the Master's Registry. When a Slave is found, the Job is sent to the Slave's Acceptor via a proxy class (refer to Chapter 5 for a short description of the proxy concept). Before a Slave registers a Job proxy in its Acceptor, it first embeds it into a JobAcceptor wrapper, which points to an Executor that is to be used for the actual execution of the job.

The Typhoon Master and Slave are available as command-line programs, respectively named *tmaster* and *tslave*, which can be considered as extensions of the Tornado CUI suite. The way these programs are to be used (*e.g.*, the fact that help is provided by using the *-h* or *-help* option) is entirely similar to other programs that are part of the Tornado CUI suite. The Typhoon source code is portable across platforms, the Typhoon command-line tools are therefore available for Windows as well as Linux. However, for Windows the Typhoon Slave is also available as a Windows Service⁸ (previously known as NT Service and similar in concept to Unix daemons).

Typhoon provides for a number of mechanisms to transfer input and output resources (typically binary or textual data files, program executables, *etc.*) between Master and Slaves:

- **Embedding:** In the case of embedding, the resource contents are converted to a string representation through Base64⁹ encoding and transferred through the Typhoon SOAP interface. This approach is practical for small resources (<1Mb), but quite inefficient for larger resources. Figure 14.8 illustrates the principle of transfer of resources through embedding.
- **File Resource Identifiers:** In this case, resources are not embedded. Only a referral to the resources' location on the Master or Slave's local disk is transmitted. This approach is evidently much more efficient than the use of embedding, but it is not as convenient since an external mechanism must be used to actually transfer the resources to the appropriate location before they are being used. In case the Slaves have access to a shared disk this problem is less important, since the transfer of resources can be grouped into one operation. Figure 14.9 illustrates the use of file resource identifiers for input resources.
- **HTTP Resource Identifiers:** This case is similar to the use of file resource identifiers. However, resource identifiers do not point to a location on a local disk, but to a resource that is served by a resource Web server and that can be downloaded through the HTTP protocol. Figure 14.10 illustrates the use of HTTP resource identifiers for input resources.

It should be noted that Typhoon is flexible with respect to the configuration of mechanisms for the transfer of resources. For each input and output resource it can be individually decided which mechanism to use.

⁷<http://www.cs.fsu.edu/~engelen/soap.html>

⁸http://en.wikipedia.org/wiki/Windows_service

⁹<http://en.wikipedia.org/wiki/Base64>

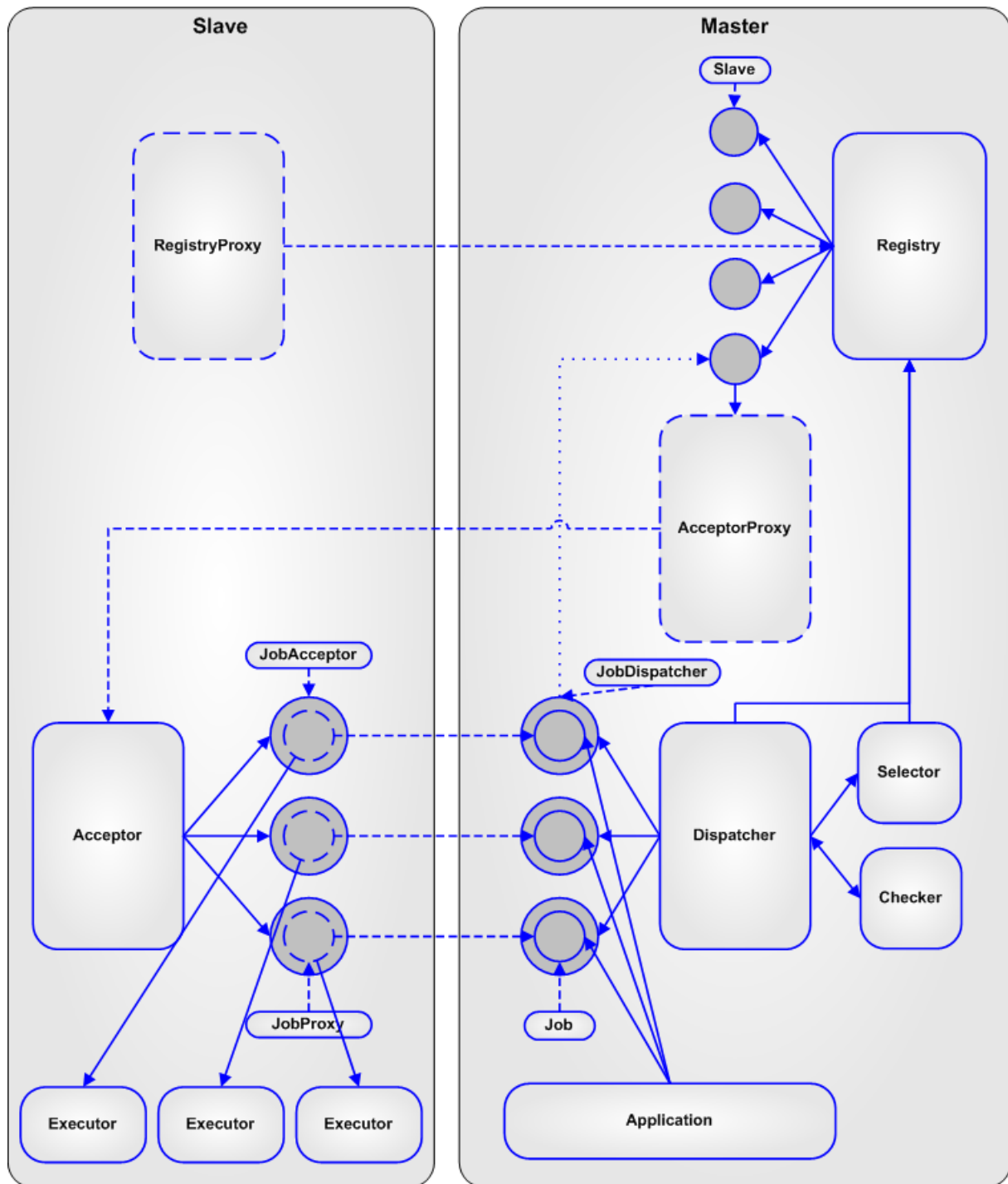


Figure 14.7: Detailed Typhoon Architecture

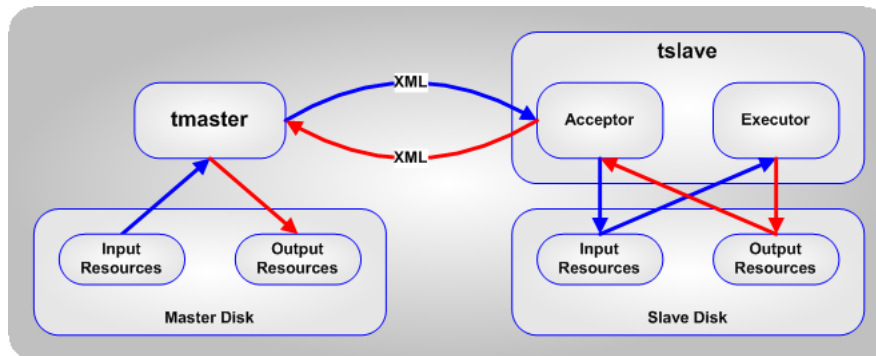


Figure 14.8: Embedding of Resources in Typhoon Job Descriptions

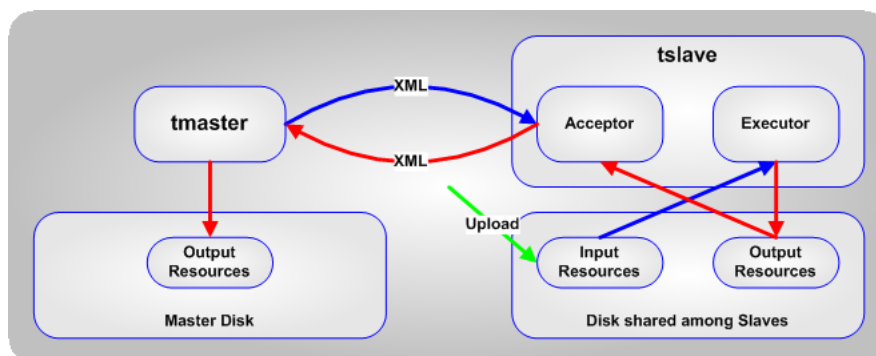


Figure 14.9: File Resource Identifiers in Typhoon Job Descriptions

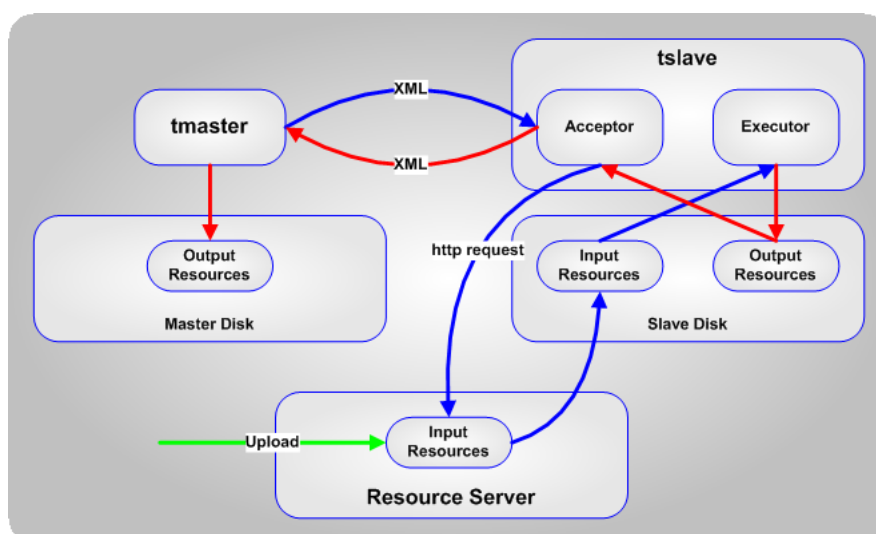


Figure 14.10: HTTP Resource Identifiers in Typhoon Job Descriptions

14.4 gLite

gLite is the middleware that has been developed at CERN¹⁰ (European Organization for Nuclear Research) in the scope of the Large Hadron Collider project¹¹. On the basis of the gLite middleware (previously known as LCG-2), the EGEE¹² grid is operated. EGEE (Enabling Grids for E-science) is the largest grid to date and consists of over 20,000 CPU's in addition to about 5 Petabytes of storage. At Ghent University, a 40-node inter-faculty gLite grid has been set up, which has recently been linked to EGEE and can be used for the execution of Tornado (compound) virtual experiments.

gLite is heavily command-line oriented and does not easily support the notion of sets of jobs. It has its own job description language (JDL - Job Description Language). For the execution of job content, it does not have a plug-in mechanism such as Typhoon, but rather relies on plain operating system commands. As a consequence, a Typhoon job description generated by Tornado that describes a set of n jobs, must be translated to n JDL files and n shell scripts defining the actual task to be executed by the computational elements. The program that performs this conversion was named *t2jdl* and is part of the Tornado CUI suite. In order to facilitate the process of submitting the JDL files generated by *t2jdl*, the latter also generates a number of convenience scripts, allowing for gLite to be used without specific knowledge of its associated command-line suite.

14.5 Adoption of Distributed Execution in the CD4WC Project

CD4WC¹³ stands for “Cost-effective development of urban wastewater systems for Water Framework Directive compliance” and is an EU project that deals with optimizing the efficiency of the urban wastewater system with regard to ecological consequences in rivers on the one hand, and investment and operation costs at the other. The need to solve this problem is a direct consequence of the European Water Framework Directive (WFD) which requests to achieve good quality for ground and surface waters on a river basin scale. With this new water quality based approach, the design of the systems is by far less pre-determined and the options to meet the goals become much more widespread. Criteria to assess the ecological consequences are - apart from the water quality - also secondary resource inputs such as energy, materials and chemicals. Various options and strategies to develop the wastewater system are to be evaluated. Main emphasis is on the dynamic interactions between the sewer, treatment plant and river subsystems as well as on the possibilities of taking measures in the receiving water and at the sources.

The responsibility of Ghent University's Department of Applied Mathematics, Biometrics and Process Control (BIOMATH) in the scope of CD4WC was to evaluate the impact of a number of WWTP plant design and upgrade scenarios (Benedetti et al., 2007; Benedetti, 2006). To this end, 100-shot Monte Carlo simulation was adopted (using LHS - Latin Hypercube Sampling). For design, 10 plant layouts had to be examined for 4 climates and 3 capacities (*cf.* Table 14.1), yielding a total number of $100 \cdot 10 \cdot 4 \cdot 3 = 12,000$ simulations. In addition, 12 upgrade scenarios had to be evaluated, however only for 2 climates and 1 capacity (*cf.* Table 14.2), yielding a total number of $100 \cdot 12 \cdot 2 \cdot 1 = 2,400$ simulations.

Given the fact that one simulation on average lasts for approximately one half hour (on a reference workstation - INTEL x86 3GHz), sequential execution of all cases would require $(12,000 + 2,400) \cdot 0.5h = 7,200h$ or 300 days. Clearly, sequential execution in this case is not a tractable solution.

Since BIOMATH has access to two distributed execution environments (its own 16-node cluster running Typhoon and the 40-node inter-faculty grid running gLite), the computational load generated by the CD4WC project was split over both environments. In a first instance, Tornado Monte Carlo

¹⁰<http://www.cern.ch>

¹¹<http://www.cern.ch/lhc>

¹²<http://www.eu-egee.org>

¹³<http://www.tu-dresden.de/CD4WC>

Table 14.1: Design Cases studied in the CD4WC Project

Layouts	Climates	Capacities
Anaerobic-anoxic-oxic (A2O)	Mediterranean	300k PE
Anaerobic-oxic (AO)	Continental	30k PE
Biodenipho	Alpine	3k PE
Biodenitro	Oceanic	
High loaded AS (HLAS)		
Low loaded AS with chemical P removal (LLAS)		
LLAS with primary settler (LLAS_PS)		
Oxidation ditch with bio-P removal (OD_bioP)		
Oxidation ditch with chemical P removal (OD_simP)		
University of Cape Town process (UCT)		

Table 14.2: Upgrade Cases studied in the CD4WC Project

Layouts	Climates	Capacities
Increase of aerated tank volume by 33% (U1)	Mediterranean	300k PE
U1+increase of final clarifier area by 33%	Continental	
U1+pre-anaerobic tank+C dosage+lower DO setpoint (U3)		
Dosage of external C (U4)		
DO control based on ammonia (U5)		
Internal recycle control based on nitrate (U6)		
U4+U6 (U7)		
Spare sludge storage (U8)		
Sludge wastage control (U9)		
Dynamic step feed (U10)		
Increase in anoxic volume, decrease in aerated volume (U11)		
Buffering ammonia peak loads with the storm tank (U12)		

experiments for each of the $(10 \times 4 \times 3 + 12 \times 2 \times 1) = 144$ parameterized plant layout models were created. All experiments were configured for 100 shots. Afterwards, generic job set descriptions were generated from all experiments by running the experiments in Mode 2. The 24 upgrade-related job set descriptions that were generated were subsequently directly presented to the Typhoon cluster. The remaining 120 design-related job set descriptions were automatically converted to a total of $120 \times 100 = 12,000$ individual gLite jobs through *t2jdl*.

As it is a much simpler system, Typhoon has a higher efficiency than gLite, *i.e.*, the amount of overhead is lower and the number of compute cycles that can be spent on real workload is higher. The efficiency of the current BIOMATH Typhoon cluster is approximately 75% (empirical number), whereas for the inter-faculty gLite grid an efficiency of 60% can be put forward. Taking into account these efficiency values, the total processing time of the Typhoon and gLite workload is as follows:

- Typhoon: $(2,400 \times 0.5h) / (0.75 \times 16) = 100h$ or 4.2 days
- gLite: $(12,000 \times 0.5h) / (0.6 \times 40) = 250h$ or 10.4 days

Instead of the 300 days that would have been required in case of sequential execution, the adoption of Typhoon and gLite only required a total of $4.2 + 10.4 = 14.6$ days of processing (actually, since both environments can be used concurrently, the total time could in principle even be reduced to 10.4 days). This is less than 5% of the total sequential execution time. In addition, thanks to the semi-automated distributed execution facilities of Tornado, the work on the CD4WC project could be performed by a small team of water quality experts with little or no computer science knowledge.

14.6 Adaptive Scheduling

Given a number of computational elements and a number of jobs to be executed, different scheduling algorithms can be devised that perform allocation of jobs to these elements. In case the set of jobs to be executed is known before the scheduling process is activated, and all jobs are handled by this process, the scheduling is called **static**. In case new jobs arrive at regular intervals and need to be scheduled as they arrive, the scheduling process is **dynamic**. Both Typhoon and gLite rely on dynamic scheduling strategies. However, one can also envisage more complex situations in which previously taken scheduling decisions are re-evaluated, causing jobs to be interrupted on one computational element and transferred to another. An approach based on the re-evaluation of scheduling decisions is called **adaptive** (Ch tepen et al., 2007). Clearly, given the dynamic nature of grid architectures, adaptive scheduling is an appealing option. Some of the reasons why adaptive scheduling can be beneficial are the following:

- **Inaccurate job length predictions:** It is the scheduler's responsibility to find a good match between the complexity of a job to be executed and the efficiency of a computational element, with as an overall goal the minimization of the total execution time of the entire set of jobs. In order to be able to perform good match-making, it is important to have accurate data on the expected execution time of a job on a particular computational resource. However, in many cases such data is not available, and rough estimates will have to be used. In case, during the execution of a job on the computational resource that it was assigned to, it appears that the initial estimate does not correspond to reality, it might be beneficial for the scheduler to decide to move the job to another more appropriate resource.
- **Varying resource loads:** As grids are highly dynamic systems, it is difficult to predict the actual load of a computational resource at a certain point in time. Especially in the case of desktop grids (grids composed of desktop machines that may also be used for other tasks but the execution of grid jobs), this problem is pertinent. In case a computational resource does not perform as predicted, it may again be wise for the scheduler to decide to re-allocate jobs.

- **Resource failures:** Even worse than computational elements that do not perform as expected are resources that fail. All jobs that are allocated to a failed resource must evidently be rescheduled.
- **Optimistic scheduling:** When an adaptive scheduling strategy is available, it can be exploited to perform optimistic scheduling, *i.e.*, scheduling of jobs for which it is assumed, although not guaranteed, that the result will be useful. In case the result turns out not to be useful, the job can simply be canceled. Decisions about the usefulness of job results are taken by the application, and hence communicated to the scheduler.
- **Replication:** Replication (*i.e.*, the creation of redundant copies) of a job is a technique that can be adopted in highly unreliable distributed environments. As soon as one copy of a replicated job is correctly executed, the others can be canceled during the next round of the adaptive scheduler.

The implementation of adaptive scheduling strategies is non-trivial. Such strategies are therefore typically not available in mainstream distributed environments. In order to be able to allow for adaptive scheduling a number of mechanisms need to be foreseen that are usually not implemented in mainstream distributed environments, for instance:

- **Progress monitoring:** Monitors the progress of the execution of a job on a computational resource. While re-evaluating previous scheduling decisions, the actual progress of jobs will be compared to the expected (estimated) progress.
- **Failure detection:** Checks computational resources at regular time intervals in order to detect failures.
- **Checkpointing:** Stores the internal state of a job at regular time intervals in order to be able to use this stored state to restart the job on another resource if needed.

Given that in the scope of Tornado a broad variety of job types (with and without inter-dependencies and with execution times ranging from seconds to hours) have to be executed, this framework is well-suited to serve as a use-case for research in the area of adaptive scheduling. Consequently, it is used as such in a joint research project between BIOMATH and Ghent University's Department of Information Technology (INTEC). Research in this project is conducted on the basis of a discrete event simulator, named DSiDE (Dynamic Scheduling in Distributed Environments) that was especially developed for modelling and simulation of aspects of highly dynamic distributed environments.

The DSiDE simulator was implemented in C++ and is XML-centric with respect to its persistent storage formats. It is designed according to the same principles as the Tornado kernel and also uses the Tornado Common Library. The conceptual diagram that is in Figure 14.11 shows that the DSiDE environment consists of two components: *DGen* and *DExec*. The first generates an XML file describing an instance of the scenario to be simulated (arrival of jobs at specific time points, activation and failure of resources, *etc.*) from another XML file that contains a stochastic model of the scenario. The latter runs the simulation on the basis of the scenario description. *DExec* manages a list of events to be handled at the current time point (CEC - Current Event Chain) and a list of future events (FEC - Future Event Chain). It also models typical grid components such as a User Interface (UI), Scheduler, Information Service (IS), Computational Resources (CR) and Storage Resources (SR). A simulation run through DSiDE can have multiple goals, one of the most important being the computation of the total time needed to process all jobs injected in the system.

On the basis of the DSiDE simulator, various scheduling and job management strategies have been investigated through simulation studies:

- Evaluation of replication and rescheduling heuristics for grid systems with varying resource availability (Ch tepen et al., 2006)

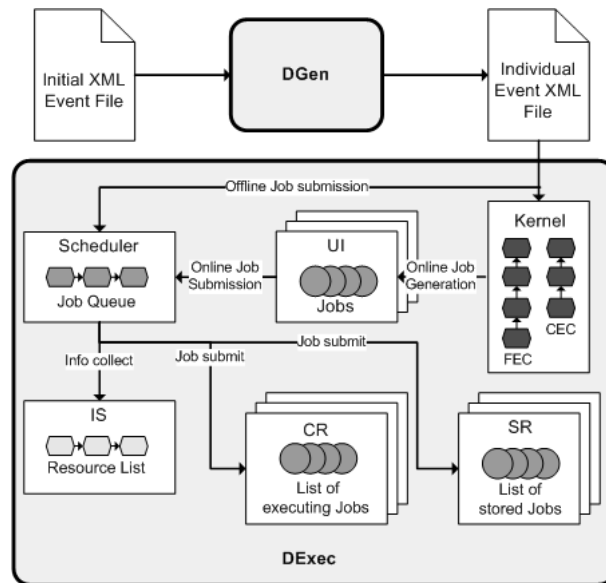


Figure 14.11: Conceptual Diagram of the DSiDE Simulator

- Providing fault-tolerance in unreliable grid systems through adaptive checkpointing and replication (Ch tepen et al., 2007)
- Efficient scheduling of dependent jobs in absence of exact job execution time predictions (Ch tepen et al., 2008)

During a second phase of the joint BIOMATH-INTEC research, the results of these studies will be applied to a real-world distributed execution environment and practical cases, including job ensembles generated by the Tornado kernel.

Part IV

Future Perspectives and Conclusions

15

Future Perspectives

15.1 Introduction

As was mentioned in Chapter 1, Tornado is a generic modelling and virtual experimentation framework for environmental systems, however the priority that was given to research and development tasks was largely dictated by requirements coming from the world of water quality management. As a result, several features that can be considered essential in other domains have not yet been implemented, and *vice versa*. This chapter therefore gives an overview of the features that have so far not been implemented, but are considered important to come to a more complete system. Also, a number of topics are discussed that are not merely potential new features, but entire areas for possible future research. In fact, in some cases this research has already been initiated.

15.2 Modelling Environment

15.2.1 Acausal Systems and Simplification of Equations

The *mof2t* compiler back-end as it is now does not perform causality assignment, *i.e.*, it is not capable of rewriting equations so that each left-hand side consists of one variable. It is able to sort and optimize equations presented to it in explicit form, but cannot yet re-arrange equations in order to bring them to a causal form. The *omc* compiler front-end contains a causality assignment module, but this module is not activated while generating flat Modelica output. When acausal Modelica input is presented to *omc*, it will therefore produce acausal flat Modelica output, which cannot yet be handled by *mof2t*. In order to overcome this problem, two options exist: either *mof2t* can be extended with a causality assignment module, or *omc* can be modified in order to (optionally) activate the causality assignment module when flat Modelica output is generated. At this point, both options are assumed to be feasible. The problem of determining which variable to compute from which equation is well-understood and has seen several solutions, some of which are discussed in (Vangheluwe, 2000). The causality assignment module in *omc* is well-isolated, activation of this code upon generation of flat Modelica can be attempted with or without the help of the developers of *omc*.

When causality assignment is implemented in *mof2t*, it must be realized that next to the problem of determining which variable is to be computed from which equation, there is also the problem of re-

arranging equations in order to bring the appropriate variable to the LHS. Related to this is the need for simplification of the RHS, *i.e.*, the need to bring like terms together so that a more compact and more efficient representation is obtained. The simplification problem does not have a unique solution, several grouping strategies are possible and are determined by the canonical representation format that is decided upon.

As was mentioned in Section 9.4.5, reduction of expressions to a canonical format is of general use in the scope of *mof2t*. The following are some of the reasons why this type of formula rewriting is important:

- Simplification of equations after causality assignment
- Simplification of equations after symbolic differentiation
- Comparison of equations in the scope of complexity evaluations
- Comparison of equations in the scope of error detection

15.2.2 Hybrid Systems

As was mentioned in Chapter 2, water quality systems are mostly continuous, and only exhibit some slight discrete behavior. In most cases, no special time and state event handling mechanisms are required to handle these systems, since discrete events occur with low frequencies which allow for them to be easily “picked up” by integration solvers. However, this is not always the case. So, in order to be able to treat all water quality systems properly (as well as other environmental systems with discrete behavior) proper event handling support is needed.

In order to implement event handling properly, it is important to be able to represent the conditions under which events occur and the actions that need to be carried out in case events occur. It is also important to be able to detect, during the course of a simulation, that an event has occurred and subsequently take the appropriate actions.

MSL is not well-equipped for hybrid system modelling, but the Modelica language has an extensive set of constructs that allow for events to be modelled. However, *mof2t* does not yet support any of these, so substantial work will need to be invested in this respect. At the executable level, the MSL-EXEC format already includes a number of features that allow for event routines to be specified (apart from the standard *ComputeInitial*, *ComputeState*, *ComputeOutput* and *ComputeFinal* routines). However, this system will have to be elaborated upon and integrated with an event detection mechanism that still has to fully developed.

15.2.3 Partial Differential Equations

In water quality modelling, most models depend on one independent variable, *i.e.*, time. Certain unit processes, such as settlers, also consider height as an independent variable. However, instead of modelling these types of unit processes through PDE’s, a manual discretization into a number of layers (typically 10) is often used. Therefore, in order to handle water quality models the way they currently are, no support for the representation and solution of PDE’s is strictly required. However, when looking at other environmental domains, this is clearly not the case. Even for domains that are closely related to water quality management, for instance hydraulics, PDE’s are an absolute requirement. In order to further render the Tornado framework complete, mechanisms for the representation and solution of PDE’s are therefore required.

Modelica nor MSL allow for PDE’s to be represented. In principle, MSL is a tiny bit better off than Modelica, since it enforces time to be explicitly declared as an independent variable. The MSL language

syntax also allows for other independent variables to be declared. However, apart from this there is no real support for PDE's in MSL. In Modelica, time is always implicitly assumed to be the only independent variable, so this fact alone makes the representation of PDE's not possible. However, research has been done with respect to an extension of Modelica that would allow for PDE representations (Saldamli et al., 2005), especially focusing on the description of boundary conditions.

It is very hard if not impossible to find a solver that is able to compute a numerical solution to every type of PDE. Instead, several numerical solvers exist that provide solutions to specific classes of PDE systems. Clearly, in the scope of a software system with the level of genericity of Tornado, such highly problem-specific approaches are not desirable. However, approaches also exist that convert a PDE description into a series of ODE's, which can be solved with regular ODE solvers. In (Le Lann et al., 1998), the following classification of methods for numerically solving PDE's is given:

- Method of lines (MOL)
- Finite difference methods
- Weighted residual methods (*e.g.*, orthogonal collocation over finite elements)
- Finite element methods
- Finite volume methods
- Adaptive grid methods
- Moving grid methods

In future work, these methods could be investigated as to their applicability in the scope of Tornado.

15.3 Experimentation Environment

15.3.1 New Virtual Experiment Types

The Tornado experimentation environment was designed to offer a set of highly-configurable, extensible, and readily available virtual experiment types. It is expected that in the future there will be a desire to extend the already available experiment types, and to add new types.

Extension of existing experiment types can be rather trivial in case merely new options or functionalities are added that do not alter the experiment's object-oriented structure. In case new properties are added, only the property descriptions of the appropriate entities are to be modified, no further changes to interface methods are needed. In case the object-oriented structure changes, new abstract interface classes and implementation classes will have to be developed, changes to the experiment's facade class may be required, and the experiment's XML parser may have to be modified. All of this is quite easy as already available concepts and code fragments simply can be duplicated and modified.

Adding new experiment types is evidently more difficult than extending existing ones. Firstly, it should be noted that, from the onset, Tornado has never been intended as an environment for prototyping new virtual experimentation methodologies. For this, much more suitable, interpretive environments exist, such as MATLAB. Secondly, it should also be noted that next to the implementation of new experiment types in Tornado, there is also always the alternative of solving problems by using already existing experiment types through one of the comprehensive Tornado API's. Given these two remarks, implementation of new experiment types for Tornado should only be considered when the following conditions are met:

- The procedure that has to be implemented as a new virtual experiment type for Tornado is sufficiently stable, *i.e.*, it is clearly known how it is to be structured in terms of entities, and which already existing experiment types should be used as embedded experiments. Clear ideas about the properties that are to be supported by each entity are of minor concern, as these can always easily be added, removed or modified.
- The procedure that has to be implemented is expected to be useful to a substantial number of users and for a substantial number of applications. In case the procedure is highly specialized, both in terms of intended audience as in terms of applicability, it might not be worth the effort to implement it as a virtual experiment type. A solution of the problem by using one or more existing types through one of the Tornado API's, in combination with custom code, might be more advisable.

In case these conditions are met, and it is decided to start the implementation of a new virtual experiment type, the following will have to be implemented:

- Abstract interface classes for the entities that make up the experiment
- Implementation classes for the entities that make up the experiment
- Entity property descriptions
- Facade class that simplifies the use of the experiment and prevents erroneous configuration
- XSD representation of the grammar of persistent representations of the experiment
- XML parser that constructs an internal representation of an experiment instance from a persistent representation
- Serializer that writes out the internal representation of an experiment instance in its persistent format. In fact, the serialization code for experiments in Tornado is not localized in one place, but fragments reside in every entity implementation class and in the facade class. When the serializer is activated from the facade class, all other entity serializers are activated in a recursive manner.
- In case the experiment requires a dynamically loadable numerical solver of a type that is not yet foreseen in the Tornado generalized solver plug-in framework, support for this additional solver type also needs to be added.

Although the number of steps that are required to add a new experiment type to Tornado is substantial, it is believed that - given proper training and a good understanding of the Tornado design and implementation concepts and conventions - this work can be carried out by environmental scientists, who do not necessarily need to be experts in computer science. Admittedly, experience has shown that often it is less easy for environmental scientist to understand the abstract, high-level principles that lie at the basis of Tornado, than to become knowledgeable on more technical, low-level programming constructs.

Viable candidates for addition as new experiment types are the Optimal Experimental Design (OED) procedures that are presented in (De Pauw, 2005). In this case it needs to be investigated whether these OED procedures for parameter estimation and model discrimination should be implemented as alternatives of the same experiment type, or as separate experiment types. Another option might be to implement them as new objective evaluation experiments, which can be used from several compound experiments, in a way similar to the ExpObjEval experiment.

15.3.2 Intelligent Solver Configuration

In Section 8.4, a generalized framework for dynamic loading of numerical solver plug-ins was discussed. This framework allows for a variety of solver implementations to be loaded for tasks such as integration, root-finding, optimization, *etc.* During the discussion of the framework it was noted that the selection of an appropriate solver for a particular task is non-trivial, and that this choice could either be made by the user on the basis of expert knowledge and/or prior experience, or automatically, *i.e.*, through machine-learning techniques based on the analysis of previously collected data.

As most users of software such as Tornado are not specialists on numerical techniques, solver-related choices are typically made on the basis of past experiences. However, when an experienced user is confronted with a case that substantially differs from previous cases, or when a novice is confronted with a complex numerical engine, making an appropriate choice between solvers is usually hard. In case sufficient time and computational resources are available, a brute-force technique such as the solver setting scenario analysis discussed in Section 8.7 could be adopted. In other cases however, an intelligent automated selection procedure would be of considerable help.

Research into intelligent solver configuration in the scope of Tornado has already been initiated. Initial results of this work can be found in (Claeys et al., 2006g) and (Claeys et al., 2007b). In principle, when analyzing a system with the aim of suggesting appropriate solver settings, white and/or black box approaches can be used:

- In the white box approach, the layout of a coupled model and/or the ultimate equation set that makes up the executable model are known. One can therefore attempt to scan these representations for features for which theoretical or empirical data suggest that particular solver settings should be used. For instance, in case an ADM1 model is discovered in a coupled model layout, chances are high that the resulting equation system will be stiff and will require a stiff solver such as CVODE or Rosenbrock.
- In the black box approach, the layout nor the executable equation set of the model under study is available. In these cases, one can study the (timing) results of simulation of the system under a number of solver setting conditions (simulation pre-runs), and try to make suggestions for solver settings on the basis hereof.

Evidently, also gray box approaches can be used where both *a priori* knowledge on the system's features are used, as well as insight gained from simulation pre-runs.

One other aspect that is very important while discussing the intelligent configuration of solvers, is the ultimate purpose of the experiment for which the solver is to be used. For instance, simulations run in the scope of a sensitivity analysis experiment are not well-served by a solver such as CVODE. Although the latter may be fast, it also introduces some jitter on its computed data. When used to compute finite differences, such jittery data can lead to unwanted results.

15.4 Distributed Computing

15.4.1 Job Length Estimation

The availability of accurate estimates on the expected duration of a job is an important asset to any scheduling process that attempts to perform good match-making between jobs to execute, and available resources. As the applications that generate jobs have the best insight into the structure of a job and its potential complexity, it is up to those applications to provide job length estimates to the scheduling process.

In the case of Tornado, the complexity of a job is determined by a large number of factors, some of which are listed below:

- Complexity of the model
 - Total number of equations
 - Number of state variables
 - Stiffness
 - Number of nodes in the AST representation
 - Number of calls of expensive functions (*e.g.*, *pow()*, *sqrt()*, *etc.*)
- Simulation time horizon
- Density of the desired output
- Dynamics of the provided input
- Efficiency of the solver algorithm
- Desired accuracy of the solution

A job length estimation algorithm should take all these factors into account, possibly in addition to historical data on similar jobs and/or simulation pre-runs. Clearly, the intelligent solver configuration problem and the job length estimation problem are related in several ways and should therefore be jointly tackled.

15.4.2 Checkpointing

In order to be able to restart failed jobs, or to move jobs from one computational resource to another in the scope of adaptive scheduling, there is a need for checkpointing (*i.e.*, storage of a job's state at regular intervals). Checkpointing can be implemented in a machine-dependent or an application-dependent manner. A machine-dependent implementation would store an image of the address space of the job. This approach works on any type of job, but is quite expensive as it will store a large amount of redundant information. An application-dependent implementation would store only those variables that are absolutely required to continue the job from its previous state.

In the scope of Tornado simulation experiments, application-dependent checkpointing is rather easy, since by definition model state variables represent the state of a simulation. It therefore suffices to store model state variables at regular intervals. For other types of experiments, finding an appropriate set of variables to store might not be trivial. For instance, it is expected that providing checkpointing to a classical gradient-based or steepest descent optimization algorithm could require a substantial amount of work.

15.5 Virtual Engineering

Problem solving in water quality management and other environmental domains is often an iterative process. Alternative solutions are proposed, presented in a human-friendly manner, validated, and hence either accepted or withdrawn. When support for this process is integrated into software tools, the term “virtual engineering” can be adopted.

Virtual engineering (Bryden and Chess, 2003; Bryden and McCorkle, 2005) is defined as integrating models and related engineering tools for analysis, simulation, optimization, and decision making, within

a computer-generated environment that facilitates multi-disciplinary collaborative product development. Virtual engineering shares many characteristics with software engineering, such as the ability to obtain many different results through different implementations. Although the concept of virtual engineering was born in the domain of mechanics and manufacturing, it can also be applied to other domains.

A virtual engineering environment provides a user-centered, first-person perspective that enables users to interact with an engineered system naturally and provides users with a wide range of accessible tools. This requires an engineering model that includes the biology, physics, and any quantitative or qualitative data from the real system. The user should be able to walk through the system and observe how it works and how it responds to changes in design, operation, or any other engineering modification. Interaction within the virtual environment should provide an easily understood interface, appropriate to the user's technical background and expertise, which enables the user to explore and discover unexpected but critical details about the system's behavior. Similarly, engineering tools and software should fit naturally into the environment and allow the user to maintain her or his focus on the engineering problem at hand. A key aim of virtual engineering is to engage the human capacity for complex evaluation.

For the domain of water quality management, Tornado and its derived applications clearly offer a number of components (representation of models, simulation and other types of virtual experiments, graphical renditions of data) that are important assets in the scope of virtual engineering. However, in order to support the virtual engineering concept further, a number of additional components are required. In the sequel, some of the components that are important in this respect are shortly discussed. All of these could be the focus of future research.

15.5.1 Version management

During any iterative engineering process it is important to be able to keep track of previous versions of an item of information. Version management¹ (also known as version control, revision control, source control or source code management (SCM)) is the process of managing multiple revisions of the same unit of information. It is most commonly used in engineering and software development, *e.g.*, to manage the ongoing development of digital documents such as application source code. Changes to these documents are identified by incrementing an associated number or letter code, termed the "revision number", "revision level", or simply "revision" and associated historically with the person making the change. Stand-alone version control systems (VCS) mostly come from the software engineering industry, but revision control is also embedded in various other types of software such as word processors, spreadsheets and content management systems. Version control systems not only allow for storing several versions of an item of information and to revert to a previous version, but typically also offer branching and merging functionality. Some popular version control systems are CVS² (Concurrent Versions System), Subversion³, Microsoft Visual SourceSafe (VSS), Perforce⁴ and IBM Rational ClearCase⁵, of which especially the first two open source tools are compelling options for use in the scope of Tornado.

Since the Tornado framework deals with a large number of, mostly XML-based, information items (experiments, layouts, high-level models, data files, *etc.*), and these information items undergo several changes during any project, it is important to be able to keep track of the nature, timestamp and author of changes that are made. Luckily, the version management problem in the context of Tornado is not unique. In fact, it is very similar to version management in software engineering and other disciplines. Therefore, it can be assumed that software tools for version management that have proven to successful in these disciplines will also be applicable to Tornado. However, there is one issue that might complicate

¹http://en.wikipedia.org/wiki/Revision_control

²<http://www.nongnu.org/cvs>

³<http://subversion.tigris.org>

⁴<http://www.perforce.com>

⁵<http://www.ibm.com/software/awdtools/clearcase>

matters: Tornado stores most of its information in XML format, whereas many other systems use a mixture of plain text files and binary files. For plain text files, mechanisms are available for discovering differences between files. These mechanisms are adopted while storing new versions of an information item: not the entire contents are stored, but only increments (differences) with respect to the previous version. For binary files, the entire contents are stored for each version. In the case of XML files, simple text-based differences are not useful, since several textual representations exist of the same set of tags representing exactly the same information content. Plain text difference tools can therefore not be used for XML and have to be replaced by XML-aware difference discovery tools. Luckily, several such tools exist (*e.g.*, Altova⁶'s Diffdog).

15.5.2 Collaborative Development

In collaborative development⁷, multiple people, possibly at different locations, collaboratively work on one and the same project. Collaborative development is important in many disciplines, including the water quality domain. To a large extent, software support for collaborative development can be delivered by version control systems, as these can be provided with a remotely accessible front-end. In this way, exchange of data between the project collaborators can be done via a centrally located repository. Synchronization problems resulting from concurrent modifications of the same documents can either be avoided through locking (making a document unavailable to others when it is being modified by a particular person), or merging of concurrent versions.

15.5.3 Workflow management

A workflow⁸ is defined as a reliably repeatable pattern of activity enabled by a systematic organization of resources, defined roles and mass, energy and information flows, into a work process that can be documented and learned. Workflows are always designed to achieve processing intents of some sort, such as physical transformation, service provision, or information processing.

The solutions to several problems in water quality modelling can be regarded as workflows. More specifically, in the scope of Tornado, any process in which consecutively models are built and experiments are executed can be viewed as a workflow. Tornado even has an experiment type that can be used to represent workflows, *i.e.*, its sequential execution experiment (ExpSeq).

As workflows become more complex, appropriate tools are needed to manage them. Version management systems can be of some help, but do not provide a complete solution to the problem as they only focus on multiple versions of one specific item, rather than on the relationship between items. However, specialized workflow management tools exist, such as Microsoft's Windows Workflow Foundation (WF). This technology is part of .NET 3.0, which is available natively on the Windows Vista operating system. Microsoft WF uses a new XML-based language named XAML for declaring the structure of workflows. XAML is used extensively in the .NET Framework 3.0 technologies, particularly in Windows Presentation Foundation (WPF), where it is used as a user interface markup language to define user interface elements, data binding, eventing, and other features, and as mentioned, in Windows Workflow Foundation (WF), in which workflows themselves can be defined using XAML. The serialization format for workflows was previously called XOML, to differentiate it from user interface markup use of XAML, but recently this distinction was abandoned.

⁶<http://www.altova.com>

⁷http://en.wikipedia.org/wiki/Collaborative_Product_Development

⁸<http://en.wikipedia.org/wiki/Workflow>

15.5.4 Conclusion

With respect to Tornado, future work could focus on virtual engineering questions such as the extent to which existing version management and workflow software tools can be adopted to allow for version management, collaborative development and workflow management. It is expected that it will be confirmed that existing generic tools, possibly with customizations, can be used to this purpose. Once these tools are into place, many of the problems that are currently noted with respect to the general management of large volumes of related data items used by multiple persons, will subside.

16

Conclusions

16.1 Overview

In this dissertation, the design and implementation of a framework for modelling and virtual experimentation with complex environmental systems was discussed. As a use-case, the domain of water quality management, which focuses on the biological and/or chemical quality of water in rivers, sewers and wastewater treatment plants, was adopted.

After a general introduction in Chapter 1, an overview of the state-of-the-art with respect to modelling and virtual experimentation with water quality systems was given. Based on an evaluation of the current situation, it was stated in Chapter 3 that - although a large number of generic and domain-specific tools are available that allow for modelling and virtual experimentation in the area of water quality management - firstly, none are fully able to aptly support the most recent research in this area in a timely and convenient fashion, and secondly, future maintainability and re-usability of existing software systems are impeded by a lack of adherence to modern architectural standards and insufficient options for deployment. One of the main complicating factors is that - according to several metrics - the evolution of model complexity follows a trend that is similar to Moore's Law. A number of requirements for software systems that overcome these problems were therefore presented in Chapter 4. These requirements are related to the following:

- Delivery of support for complex modelling
- Delivery of support for complex virtual experimentation
- Flexibility with regard to deployment and integration
- Compliance with modern architectural standards

On the basis of these requirements, a new software framework, named Tornado, was built. It adopts a number of design principles and a variety of powerful software tools and methods that were respectively discussed in Chapter 5 and Chapter 6. In Chapter 7, a common library containing classes for convenience and platform-abstraction, was presented. This library was developed especially for use by the Tornado kernel, but has also been adopted in the scope of a number of other applications that were implemented

in the wake of Tornado (the Typhoon distributed computing environment and the DSIDE discrete event simulator)

Since Tornado is based on the concept of strictly separated Modelling and Experimentation Environments, both environments were individually discussed, starting with the Experimentation Environment in Chapter 8, and followed by the Modelling Environment in Chapter 9.

The issue of deployment and integration of the Tornado framework with other tools was split into five parts, respectively pertaining to application programming interfaces, command-line user interfaces, graphical user interfaces, remote execution and distributed execution. These were discussed in Chapters 10 through 14.

Finally, Chapter 15 both gave an overview of features that are desired but have not yet been implemented into the Tornado framework, and future research directions that are considered relevant to Tornado.

Below, the requirements for software frameworks that aptly fulfill the demands imposed by the most recent water quality research problems are revisited. For each requirement, it is discussed if and how the Tornado framework provides a solution.

16.2 Complex Modelling

With respect to complex modelling, it was stated that a state-of-the-art software tool should support the principle of translation of high-level, readable and maintainable model descriptions to low-level, efficient executable code. If desired, this executable code should also be usable for the construction of self-contained components, which can be used in component-based modelling architectures. High-level model representations should be such that concepts such as equation-based modelling, hierarchical modelling, object-oriented modelling, declarative modelling, acausal modelling and multi-domain modelling are supported. Executable models should be fast to generate, fast to execute and should provide for sufficient stability. Next to generic high-level model representations, domain-specific representations are also required.

In the scope of Tornado, two declarative, hierarchical, object-oriented, equation-based modelling languages play an important role: MSL and Modelica. MSL has been available as of the first incarnation of the Tornado framework (Tornado-I) and has been the cornerstone of model development in the scope of many water quality modelling studies. Since MSL does not support the concepts of acausal and multi-domain modelling, the Modelica language was introduced as of the second incarnation of the Tornado framework (Tornado-II). Albeit very powerful, Modelica is also a complex language. Consequently, a hybrid approach is adopted in which *omc*, the standard OpenModelica model compiler front-end, is used in combination with *mof2t*, a custom model compiler back-end for Tornado. This model compiler back-end allows for the generation of MSL-EXEC executable models that can be used by the Tornado virtual experimentation kernel, as well as for the generation of S-functions that can be adopted in the scope of MATLAB/Simulink.

Since executable models need to be fast to generate, the Tornado executable model format was designed in such a way that meta-information is represented in XML and computational information is described in a compiled general-purpose programming language, *i.e.*, C. In this way, executable models can quickly be produced, using the user's C compiler of choice.

In order to ensure the run-time performance of executable models, three techniques were implemented that reduce the model complexity and hence provide a considerable performance improvement: constant folding, equiv substitution, and lifting of equations. The first computes the results of constant sub-expressions at compile-time, the second removes aliases from a set of equations, and the third allows for detecting initial and output equations and for moving them to appropriate equation sections. The performance that can be gained through the application of these techniques was found to be 10 to 30%.

As a result of the manipulations that are performed on models during the executable code generation process, the output of this process is often not recognizable anymore to a user. Consequently, stability is an important issue: in case a problem does occur, it must be reported in a way that is understandable to a user. Two techniques were implemented that allow for improved stability and error reporting in executable models: code instrumentation and the generation of bounds checking code. The first replaces potentially dangerous constructs (*e.g.*, division-by-zero) by “safe” versions in which arguments are first checked on validity before being applied. The second generates code that performs range checking at run-time (*e.g.*, to detect negative values). By generating this code only for variables that need it, a performance improvement of up to 50% could be realized with respect to kernel-based range checking.

In general, a situation is created in which run-time performance of safe, optimized Tornado models is comparable to the performance of unsafe, non-optimized models. Hence, the time spent on performing safety checks is compensated by the time gained through optimization.

As was discussed in Chapter 15, the Tornado Modelling Environment is not free of *hiata*. Several elements are still missing to make it a complete environment, *e.g.*, support for acausal modelling, canonization of expressions, hybrid systems and PDE's.

16.3 Complex Virtual Experimentation

With respect to complex virtual experimentation, it was stated that an environment is required that provides for an extensible set of readily available and highly configurable virtual experiment types that allow for simulation, steady-state analysis, sensitivity analysis, optimization, uncertainty analysis and optimal experiment design.

In order to realize this goal, a hierarchical virtual experimentation framework was designed and developed in which new virtual experiments can be created by re-using already existing experiments. The composition graph that represents the virtual experiments that have so far been developed for Tornado consists of 15 different types of experiments, which have been named as follows:

- **ExpSimul**: Dynamic Simulation Experiment
- **ExpSSRoot**: Steady-state Analysis Experiment with Root Finder
- **ExpSSOptim**: Steady-state Analysis Experiment with Optimizer
- **ExpObjEval**: Objective Evaluation Experiment
- **ExpScen**: Scenario Analysis Experiment
- **ExpMC**: Monte Carlo Analysis Experiment
- **ExpOptim**: Optimization Experiment
- **ExpCI**: Confidence Information Analysis Experiment
- **ExpSens**: Sensitivity Analysis Experiment
- **ExpStats**: Statistical Analysis Experiment
- **ExpEnsemble**: Ensemble Simulation Experiment
- **ExpSeq**: Sequential Execution Experiment
- **ExpMCOptim**: Optimization Monte Carlo Analysis Experiment

- **ExpScenOptim**: Optimization Scenario Analysis Experiment
- **ExpScenSSRoot**: Steady-state Analysis with Root Finder Scenario Analysis Experiment

Using these types of experiments, a broad range of problems can be solved, ranging from simple simulation studies to large-scale risk analyses, such as those performed in the scope of the EU CD4WC project. All experiment types are highly configurable through an extensive set of properties, which can be dynamically queried and modified. New experiment types can be added by computer scientists or environmental scientists, provided that the design and implementation principles of Tornado are well-understood.

Most virtual experiment types are guided by numerical solver algorithms. For the incorporation of these solvers into Tornado, a generalized framework for abstraction and dynamic loading of solver plug-ins was devised. This framework allows for solvers to be classified according to their purpose (integration, optimization, *etc.*) and for new solvers to be developed on the basis of a library of base classes. At run-time, solvers can be dynamically loaded and removed from the system.

The flexibility and orthogonal design of the Tornado Experimentation Environment is well illustrated by its ability to perform solver setting scenario analysis, which is a method that allows for applying variable sweeping techniques to solver settings (such as integrator tolerances), instead of to model parameters. Using this technique, overall execution times of large-scale compound virtual experiments can be shortened by applying improved solver settings that are discovered through *a priori* exploration of the solver setting space.

16.4 Deployment and Integration Flexibility

Models and virtual experiments that are developed in the scope of Tornado should be usable in a broad range of application types, hence adhering to the Write Once, Run Anywhere principle. This implies that it should be possible to deploy the Tornado kernel in various ways, and to integrate it with other currently existing applications, and potential future applications. More specifically, Tornado should be applicable in stand-alone, remote, web-based, distributed and embedded applications.

In order to allow for the integration of Tornado in other software, several API's were developed. Some of these are comprehensive and allow for any Tornado-related operation to be performed, while others are restricted and only allow for the most frequently occurring operations. Comprehensive API's were developed for C++ and .NET. Restricted API's were developed for C, the Java Native Interface, MATLAB MEX, OpenMI and CLIPS. A special case is the CMSLU interface that allows for the Tornado kernel to be called to run any type of experiment from within a Tornado model.

Several stand-alone applications were developed on the basis of the Tornado kernel in its various incarnations. One of these applications is a suite of command-line programs that is developed along with the Tornado kernel itself and is useful for testing, automation and advanced use. Next to the command-line suite, a number of graphical applications were developed: WEST++, WEST-3 and EAST are based on (modified versions of) the Tornado-I kernel, while MORE and WEST-4 are based on Tornado-II. These graphical applications illustrate that for application development, various technologies can be applied to one and the same Tornado kernel. Some of these applications also support domain-specific model representations, such as the Petersen matrix representation.

With respect to remote and web-based use, a number of technologies were discussed that are potentially applicable to Tornado. Some of these technologies have already been used in the scope of applications (sockets, DCOM and SOAP), while others have merely been adopted in prototype projects (.NET Remoting, ASP.NET). Still some others have not yet been applied, but could in case a need for this would arise (CORBA, OPC, Java RMI).

Finally, coarse-grained gridification was applied to Tornado at the level of sub-experiments (*i.e.*, a set of simulation jobs is carried out concurrently on a pool of computational nodes). As a result, Tornado can be deployed in a distributed fashion on the basis of the Typhoon cluster software (which was developed especially to serve as a light-weight solution for the execution of jobs generated by Tornado), as well as on the basis of the gLite grid middleware. Through the application of these distributed technologies, the extreme computational workload (14,400 simulations each requiring 30' of computation time) that was a result of the CD4WC project's Monte Carlo study could be processed in a timely fashion (approximately 10 days).

16.5 Modern Architectural Standards

One last major requirement for software tools that one wants to remain successfully applicable to complex modelling and virtual experimentation problems in the long run, is adherence to modern architectural concepts and standards. As the application of modern design principles is in some cases irreconcilable with performance requirements, it must be investigated exactly where these performance bottlenecks are located in order to revert to low-level approaches only in case this is absolutely needed.

Tornado was developed from scratch using object-oriented design and implementation principles. C++ was used as a programming language and design patterns such as the singleton, factory, facade and proxy patterns were adopted. Platform-independence of the kernel (but not necessarily its external interfaces) was ensured and thread-safety for high-level entities was provided. For these high-level entities, XML-based persistent formats were devised and described on the basis of XML Schema Definitions. Furthermore, a mechanism that provides for run-time querying of entity properties was provided, which alleviates the need for modification of entity interfaces in case of changes to the set of properties supported by an entity.

16.6 Overall Conclusion

As an overall conclusion, it can be stated that through the design and development of the Tornado framework, the solution of water quality management problems that were hitherto hindered by performance limitations or limitations to the degree to which complexity could be handled, has now become possible. Moreover, thanks to the design of the framework, it is expected that Tornado will be able to adapt to the expected continued increase in complexity for a considerable period of time. In order to render the Tornado framework more complete, a need for future work and the exploration of additional research areas exists. In some cases, this additional research has already been initiated.

Bibliography

- Assyr Abdulle and Alexei A. Medovikov. Second order Chebyshev methods based on orthogonal polynomials. *Numerische Mathematik*, 90(1):1–18, 2001.
- Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles and Tools*. Addison-Wesley, Boston, MA, 1986.
- Y. Amerlinck, S. De Grande, and F. Claeys. WEST - MSL reference guide. Technical report, MOSTfor-WATER N.V., Kortrijk, Belgium, 2007.
- P. Aronsson. *Automatic parallelization of equation-based simulation programs*. PhD thesis, Linköping Studies in Science and Technology, Dissertation No. 1022, Linköping University, Sweden, 2006.
- P. Aronsson and P. Fritzson. Multiprocessor scheduling of simulation code from modelica models. In *Proceedings of the 2nd International Modelica Conference*, Oberpfaffenhofen, Germany, March 18–19 2002.
- E. Ayesa, A. De la Sota, P. Grau, J.M. Sagarna, A. Salterain, and J. Suescun. Supervisory control strategies for the new WWTP of Galindo-Bilbao: The long run from the conceptual design to the full-scale experimental validation. *Water Science and Technology*, 53(4–5):193–201, 2006.
- S.V. Barai and Yoram Reich. Ensemble modelling or selecting the best model: Many could be better than one. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 13(5):377–386, 1999.
- P. Barton and C. Pantelides. Modelling of combined discrete/continuous processes. *AIChE*, 40:966–979, 1994.
- D.J. Batstone, J. Keller, I. Angelidaki, S.V. Kalyuzhnyi, S.G. Pavlostathis, A. Rozzi, W.T.M. Saunders, H. Siegrist, and V.A. Vavilin. *Aerobic Digestion Model No.1 (ADM1)*. Scientific and Technical Report No.13. IWA Publishing, London, UK, 2002.
- Brian Beachkofski and Ramana Grandhi. Improved distributed hypercube sampling. Technical Report 2002-1274, American Institute of Aeronautics and Astronautics, 2002.
- L. Benedetti, D. Bixio, F. Claeys, and P.A. Vanrolleghem. Tools to support a model-based methodology for emission/immission and benefit/cost/risk analysis of wastewater treatment systems. *Environmental Modelling and Software (Accepted)*, 2007.
- Lorenzo Benedetti. *Probabilistic design and upgrade of wastewater treatment plants in the EU Water Framework Directive context*. PhD thesis, Dept. of Applied Mathematics, Biometrics and Process Control (BIOMATH), Ghent University, Belgium, 2006.
- Margaret A. Boden. Failure is not the spur. In G. Selfridge, Oliver, Edwina L. Rissland, and Michael A. Arlib, editors, *Proceedings of the NATO Advanced Research Institute on Adaptive Control of Ill Defined Systems*, pages 143–148, Moretonhampstead, Devon, UK, June 21–26 1984. Plenum Press.
- K.E. Brenan, S.L. Campbell, and L.R. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. Elsevier, New York, NY, 1989.
- R.P. Brent. *Algorithms for Minimization without Derivatives*. Prentice-Hall, Englewood Cliffs, NJ, 1973. Chapter 7.

- A.P. Breunese and J.F. Broenink. Modeling mechatronic systems using the SIDOPS+ language. In *Proceedings of the 1997 International Conference on Bond Graph Modelling and Simulation (ICBGM)*. The Society for Computer Simulation International, 1997.
- P.N. Brown and A.C. Hindmarsh. Reduced storage matrix methods in stiff ODE systems. *Journal of Applied Mathematics and Computation*, 31:40–91, 1989.
- P.N. Brown, A.C. Hindmarsh, and L.R. Petzold. Using krylov methods in the solution of large-scale differential-algebraic systems. *SIAM Journal of Scientific Computing*, 15:1467–1488, 1994.
- K.M. Bryden and K.L. Chess. Virtual engineering offers applications for today, challenges for tomorrow. *Power*, 147(2), 2003.
- K.M. Bryden and D. McCorkle. Virtual engineering. *Mechanical Engineering*, 127(11), 2005.
- M.F. Cardoso, R.L. Salcedo, and S.F. DeAzevedo. The simplex-simulated annealing approach to continuous non-linear optimization. *Computers and Chemical Engineering*, 20(9):1065–1080, 1996.
- Jacob Carstensen, Peter A. Vanrolleghem, Wolfgang Rauch, and Peter Reichert. Terminology and methodology in modelling for water quality management - a discussion starter. *Water Science and Technology*, 36(5):157–168, 1997.
- E. Ward Cheney and David R. Kincaid. *Numerical Mathematics and Computing*. Brooks/Cole, 5th edition, 2003.
- M. Chtepen, F.H.A. Claeys, B. Dhoedt, F. De Turck, P. Demeester, and P.A. Vanrolleghem. Evaluation of replication and rescheduling heuristics for grid systems with varying resource availability. In *Proceedings of the 2006 Parallel and Distributed Computing and Systems Conference (PDCS)*, Dallas, TX, November 13–15 2006.
- M. Chtepen, F.H.A. Claeys, F. De Turck, B. Dhoedt, P.A. Vanrolleghem, and P. Demeester. Providing fault-tolerance in unreliable grid systems through adaptive checkpointing and replication. In *Proceedings of the 2007 International Conference on Computational Science (ICCS)*, Beijing, China, May 27–30 2007.
- M. Chtepen, F.H.A. Claeys, B. Dhoedt, F. De Turck, P. Demeester, and P.A. Vanrolleghem. Efficient scheduling of dependent jobs in absence of exact job execution time prediction. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid) (Submitted)*, Lyon, France, May 19–22 2008.
- Maria Chtepen. Gedistribueerde virtuele experimentering binnen de WEST++ omgeving (in Dutch). Master's thesis, Ghent University, Department of Information Technology, Gaston Crommenlaan 8 bus 201, B-9050, Gent, Belgium, June 2004.
- F. Claeys, M. Chtepen, L. Benedetti, B. Dhoedt, and P.A. Vanrolleghem. Distributed virtual experiments in water quality management. *Water Science and Technology*, 53(1):297–305, 2006a.
- F. Claeys, D. De Pauw, L. Benedetti, I. Nopens, and P.A. Vanrolleghem. Tornado: A versatile efficient modelling & virtual experimentation kernel for water quality systems. In *Proceedings of the iEMSs 2006 Conference*, Burlington, VT, July 9–13 2006b.
- F.H.A. Claeys, M. Chtepen, L. Benedetti, W. De Keyser, P. Fritzson, and P.A. Vanrolleghem. Towards transparent distributed execution in the Tornado framework. In *Proceedings of the 2006 Environmental Application and Distributed Computing Conference (EADC)*, Bratislava, Slovakia, October 16–17 2006c.

- F.H.A. Claeys, P. Fritzson, and P.A. Vanrolleghem. Using Modelica models for complex virtual experimentation with the Tornado kernel. In *Proceedings of the 2006 Modelica Conference*, Vienna, Austria, September 4–5 2006d.
- F.H.A. Claeys, P.A. Vanrolleghem, and P. Fritzson. Boosting the efficiency of compound virtual experiments through a priori exploration of the solver setting space. In *Proceedings of the 2006 European Modeling and Simulation Symposium (EMSS)*, Barcelona, Spain, October 4–6 2006e.
- F.H.A. Claeys, P.A. Vanrolleghem, and P. Fritzson. A generalized framework for abstraction and dynamic loading of numerical solvers. In *Proceedings of the 2006 European Modeling and Simulation Symposium (EMSS)*, Barcelona, Spain, October 4–6 2006f.
- F.H.A. Claeys, P. Fritzson, and Vanrolleghem P.A. Generating efficient executable models for complex virtual experimentation with the Tornado kernel. *Water Science and Technology*, 56(6):65–73, 2007a.
- P. Claeys, F. Claeys, and P.A. Vanrolleghem. Intelligent configuration of numerical solvers of environmental ODE/DAE models using machine-learning techniques. In *Proceedings of the iEMSs 2006 Conference*, Burlington, VT, July 9–13 2006g.
- P. Claeys, B. De Baets, and Vanrolleghem P.A. Towards automatic numerical solver selection using a repository of pre-run simulations. In *Proceedings of the 7th International Symposium on Systems Analysis and Assessment in Water Management (WATERMATEX)*, Washington, D.C., May 7–9 2007b.
- J.C. Cleaveland. Building application generators. *IEEE Software*, 5(4):25–33, 1988.
- S.D. Cohen and A.C. Hindmarsh. CVODE, a stiff/nonstiff ODE solver in C. *Computers in Physics*, 10(2):138–143, 1995.
- J.B. Copp, editor. *The COST simulation benchmark*. European Commission, 2002.
- O. David, S.L. Markstrom, K.W. Rojas, L.R. Ahuja, and I.W. Schneider. *Agricultural System Models in Field Research and Technology Transfer*, chapter The Object Modelling System, pages 317–331. Lewis Publishers, CRC Press LLC, 2002.
- Mónica de Gracia. *Mathematical modelling of waste sludge digestion reactors (in Spanish)*. PhD thesis, TECNUN, University of Navarra, Spain, 2007.
- Frederik De Laender. *Predicting effects of chemicals and freshwater ecosystems: model development, validation and application*. PhD thesis, Dept. of Applied Mathematics, Biometrics and Process Control (BIOMATH), Ghent University, Belgium, 2007.
- Dirk J.W. De Pauw. *Optimal experimental design for calibration of bioprocess models: a validated software toolbox*. PhD thesis, Dept. of Applied Mathematics, Biometrics and Process Control (BIOMATH), Ghent University, Belgium, 2005.
- Dirk J.W. De Pauw and Peter A. Vanrolleghem. Practical aspects of sensitivity function approximation for dynamic models. *Mathematical and Computer Modelling of Dynamical Systems*, 12(5):395–414, 2007.
- V.C.J. De Schepper, L. Benedetti, K.M.A. Holvoet, and P.A. Vanrolleghem. Extension of the River Water Quality Model No.1 with the fate of pesticides. *Journal of Contaminant Hydrology (Submitted)*, 2006.
- Tolessa Chuco Deksisssa. *Dynamic integrated modelling of basic water quality and fate and effect of organic contaminants in rivers*. PhD thesis, Dept. of Applied Mathematics, Biometrics and Process Control (BIOMATH), Ghent University, Belgium, 2004.

- Qiang Du, Vance Faber, and Max Gunzburger. Centroidal voronoi tessellations: Applications and algorithms. *SIAM Review*, 41:637–676, 1999.
- H. Elmqvist, D. Brück, and M. Otter. Dymola - user's manual. Technical report, Dynasim AB, Ideon Research Park, Lund, Sweden, 1996.
- I. Foubert, K. Dewettinck, G. Janssen, and P.A. Vanrolleghem. Modelling two-step isothermal fat crystallization. *Journal of Food Engineering*, 75(4):551–559, 2005.
- P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica*. Wiley-IEEE Press, February 2004. ISBN 0-471-47163-1.
- P. Fritzson, L. Viklund, and D. Fritzson. High-level mathematical modelling and programming. *IEEE Software*, 12(3), 1995.
- Richard M. Fujimoto. *Parallel and Distributed Simulation*. John Wiley & Sons, New York, NY, 2000.
- E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 2003.
- K.R. Gegenfurtner. PRAXIS: Brent's algorithm for function minimization. *Behaviour Research Methods, Instruments & Computers*, 24:560–564, 1992.
- S. Gillot, B. De Clercq, D. Defour, F. Simoens, K. Gernaey, and P.A. Vanrolleghem. Optimisation of wastewater treatment plant design and operation using simulation and cost analysis. In *Proceedings of WEFTEC 1999: 72nd Annual Technical Exhibition & Conference*, New Orleans, LA, October 9–13 1999.
- D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Boston, MA, 1989.
- Malcolm Graham. *Coding error isolation in computational simulation models - With application to wastewater treatment systems*. PhD thesis, the University of Queensland, Brisbane, Australia, 2005.
- P. Grau, M. de Gracia, P.A. Vanrolleghem, and E. Ayesa. A new plant-wide modelling methodology for WWTP's. *Water Research*, 41(19):4357–4372, 2007a.
- P. Grau, P.A. Vanrolleghem, and E. Ayesa. BSM2 Plant-Wide Model construction and comparative analysis with other methodologies for integrated modelling. *Water Science and Technology*, 56(8): 57–65, 2007b.
- Paloma Grau. *A new methodology for the integrated modelling of WWTP's (In Spanish)*. PhD thesis, TECNUN, University of Navarra, Spain, 2007.
- J.B. Gregersen, P.J.A. Gijsbers, and S.J.P. Westen. OpenMI: Open modelling interface. *Journal of Hydroinformatics*, 9(3):175–191, 2007.
- G. Grubel, M. Otter, F. Breiteneker, A. Prinz, G. Schuster, I. Bausch-Gall, and H. Fischer. An ACSL-model translator to the neutral Fortran DSblock-modelformat. In *Proceedings of the 1994 IEEE/IFAC Joint Symposium on Computer-Aided Control System Design*, pages 143–148, March 7–9 1994.
- W. Gujer. Activated sludge modelling: past, present and future. *Water Science and Technology*, 53(3): 111–119, 2006.
- E. Hairer, S.P. Norsett, and G. Wanner. *Differential Equations I. Nonstiff Problems*. Springer Series in Computational Mathematics. Springer-Verlag, 2 edition, 1993.

- A.C. Hearn. REDUCE user's manual. Technical report, The Rand Corporation, Santa Monica, CA, 1987.
- M. Henze, C. Grady, W. Gujer, G. Marais, and T. Matsuo. Activated sludge model no. 1. Technical report, IAWPRC Task Group on Mathematical Modelling for Design and Operation of Biological Wastewater Treatment Processes, IAWPRC, London, 1986.
- M. Henze, W. Gujer, T. Mino, and M. van Loosdrecht. *Activated Sludge Models ASM1, ASM2, ASM2d, and ASM3*. Scientific and Technical Report No.9. IWA Publishing, London, UK, 2000.
- F. Herrera, M. Lozano, and J.L. Derdegay. Tackling real-coded genetic algorithms: operators and tools for behavioural analysis. *Artificial Intelligence Review*, 12(4):265–319, 1998.
- C. Hillyer, J. Bolte, F. van Event, and A. Lamaker. The modcom modular simulation system. *European Journal of Agronomy*, 18:333–343, 2003.
- A.C. Hindmarsh, P.N. Brown, K.E. Grant, S.L. Lee, R. Serban, D.E. Shumaker, and C.S. Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software*, 31(3):363–396, 2005.
- Alan C. Hindmarsh. LSODE and LSODI, two new initial value ordinary differential equation solvers. *ACM-Signum Newsletter*, 15(4):10–11, 1980.
- Katrijn Holvoet. *Monitoring and modelling the dynamic fate and behaviour of pesticides in river systems at catchment scale*. PhD thesis, Dept. of Applied Mathematics, Biometrics and Process Control (BIOMATH), Ghent University, Belgium, 2006.
- J. Hutton. The Maple computer algebra system – a review. *Journal of Applied Econometrics*, 10(3):329–337, 1995.
- IEEE. Standard VHDL analog and mixed-signal extensions. Technical Report 1076.1, IEEE, 1997.
- Güclü Insel, Dave Russell, Bruce Beck, and Peter A. Vanrolleghem. Evaluation of nutrient removal performance for an Orbal plant using the ASM2d model. In *Proceedings of WEFTEC 2003: 76th Annual Technical Exhibition & Conference*, Los Angeles, CA, 2003.
- A. Jeandel, F. Boudaud, P. Ravier, and A. Buhsing. Ulm: Un langage de modélisation, a modelling language. In *Proceedings of the 1996 CESA Conference*, Lille, France, 1996.
- Tao Jiang. *Characterisation and modelling of soluble microbial products in membrane bioreactors*. PhD thesis, Dept. of Applied Mathematics, Biometrics and Process Control (BIOMATH), Ghent University, Belgium, 2007.
- M. Kloas, V. Friesen, and M. Simons. Smile - a simulation environment for energy systems. In Sydow, editor, *Proceedings of the 1995 Symposium on Systems Analysis and Simulation (SAS)*. Gordon and Breach, 1995.
- S. Kops, H. Vangheluwe, F. Claeys, and P.A. Vanrolleghem. The process of model building and simulation of ill-defined systems: Application to wastewater treatment. *Mathematical and Computer Modelling of Dynamical Systems*, 5(4):298–312, 1999.
- S. Kralish and P. Krause. Jams – a framework for natural resource model development and application. In *Proceedings of the iEMSs 2006 Conference*, Burlington, VT, July 2006.
- Jean-Marc Le Lann, Alain Sargousse, and Xavier Joulia. Dynamic simulation of partial differential algebraic systems: Application to some chemical engineering problems. In *Proceedings of the 3rd ICL Joint Conference*, Toulouse, France, July 1998.

- M. Leuthold, Raymond. On the use of Theil's inequality coefficient. *American Journal of Agricultural Economics*, 57(2):344–346, 1975.
- John R. Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly & Associates, Inc., Sebastopol, CA, February 1995.
- Koenraad Mahieu. *Modelling methane oxidation in landfill cover soils using stable isotopes*. PhD thesis, Dept. of Applied Mathematics, Biometrics and Process Control (BIOMATH), Ghent University, Belgium, 2006.
- S.E. Mattsson, M. Andersson, and K.J. Aström. *CAD for Control Systems*, chapter Object-oriented modelling and simulation, pages 31–69. Marcel Dekker Inc, New York, 1993.
- S.E. Mattsson, H. Elmqvist, and M. Otter. Physical system modelling with modelica. *Control Engineering Practice*, 6:501–510, 1998.
- Jurgen Meirlaen. *Immission based real-time control of the integrated urban wastewater system*. PhD thesis, Dept. of Applied Mathematics, Biometrics and Process Control (BIOMATH), Ghent University, Belgium, 2002.
- Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 1997. ISBN 0-13-629155-4.
- E.E.L. Mitchell and J.S. Gauthier. Advanced continuous simulation language (ACSL). *Simulation*, 26(3):72–78, 1976.
- Stefan Näher and Kurt Mehlhorn. LEDA: A library of efficient data types and algorithms. *Lecture Notes in Computer Science*, 443:1–5, 1990.
- J.A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.
- Ingmar Nopens. *Modelling the activated sludge flocculation process: a population balance approach*. PhD thesis, Dept. of Applied Mathematics, Biometrics and Process Control (BIOMATH), Ghent University, Belgium, 2005.
- M. Otter and H. Elmqvist. The DSblock model interface for exchanging model components. In *Proceedings of the 1995 Eurosim Simulation Congress Eurosim*, Vienna, Austria, September 11–15 1994.
- R. Pavelle and P.S. Wang. MACSYMA from F to G. *Journal of Symbolic Computation*, 1:69–100, 1985.
- Britta Petersen. *Calibration, identifiability and optimal experimental design of activated sludge models*. PhD thesis, Dept. of Applied Mathematics, Biometrics and Process Control (BIOMATH), Ghent University, Belgium, 2000.
- E. Petersen. *Chemical Reaction Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1965.
- Linda R. Petzold. *Scientific Computing*, chapter A Description of DASSL: A Differential/Algebraic System Solver, pages 65–68. North-Holland, Amsterdam, Netherlands, 1983a.
- Linda R. Petzold. Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations. *SIAM Journal on Scientific and Statistical Computing*, 4:136–148, 1983b.
- P. Piela, T. Epperly, K. Westerberg, and A. Westerberg. ASCEND: An object-oriented computer environment for modelling and analysis: the modelling language. *Computers and Chemical Engineering*, 15(1):53–72, 1991.

- Adrian Pop and Peter Fritzson. MetaModelica: A unified equation-based semantical and mathematical modeling language. *Lecture Notes in Computer Science*, 4228:211–229, 2006.
- W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C: the art of scientific computing*. Cambridge University Press, New York, NY, 2nd edition, 1992.
- J.M. Rahman, S.P. Seaton, J.-M. Perraud, Hotham H., D.I. Verrelli, and J.R. Coleman. It’s time for a new environmental modelling framework. In D.A. Post, editor, *Proceedings of the 2003 International Congress on Modelling and Simulation*, pages 1727–1732, Townsville, New Zealand, July 14–17 2003. Modelling and Simulation Society of Australia and New Zealand Inc.
- P. Reichert. AQUASIM - a tool for simulation and data-analysis of aquatic systems. *Water Science and Technology*, 30(2):21–30, 1994.
- P. Reichert. A standard interface between simulation programs and systems analysis software. *Water Science and Technology*, 53(1):267–275, 2006.
- P. Reichert, R. von Schulthess, and D. Wild. The use of AQUASIM for estimating parameters of activated sludge models. *Water Science and Technology*, 31(2):135–147, 1995.
- P. Reichert, Borchardt D., Henze M., Rauch W., Shanahan P., Somlyódy L., and P.A. Vanrolleghem. *River Water Quality Model No.1*. Scientific and Technical Report No.12. IWA Publishing, London, UK, 2001.
- Andrea E. Rizzoli, George Leavesley, James C. Ascough, Robert M. Argent, Ioannis N. Athanasiadis, Virginia Brilhante, Filip H.A. Claeys, Olaf David, Marcello Donatelli, Peter Gijssbers, Denis Havlik, Ayalew Kassahun, Peter Krause, Nigel W.T. Quinn, Huub Scholten, Richard S. Sojda, and Fernandino Villa. *State of the Art and Futures in Environmental Modelling and Software*, chapter Integrated Modelling Frameworks for Environmental Assessment and Decision Support (*Accepted for publication*). Developments in Integrated Environmental Assessment (IDEA). Elsevier, 2008.
- Diederik Rousseau. *Performance of constructed treatment wetlands: model-based evaluation and impact of operation and maintenance*. PhD thesis, Dept. of Applied Mathematics, Biometrics and Process Control (BIOMATH), Ghent University, Belgium, 2005.
- P. Sahlin, A. Bring, and E.F. Sowell. The neutral model format for building simulation, version 3.02. Technical report, Department of Building Sciences, The Royal Institute of Science, Stockholm, Sweden, 1996.
- L. Saldamli, B. Bachmann, P. Fritzson, and H. Weismann. A framework for describing and solving PDE models in Modelica. In *Proceedings of the 2005 Modelica Conference*, Hamburg-Harburg, Germany, March 7–8 2005.
- A. Siemers, D. Fritzson, and P. Fritzson. Meta-modeling for multi-physics co-simulations applied for openmodelica. In *Proceedings of the 2006 International Congress on Methodologies for Emerging Technologies in Automation (ANIPLA)*, Rome, Italy, November 13–15 2006.
- Gürkan Sin. *Systematic calibration of activated sludge models*. PhD thesis, Dept. of Applied Mathematics, Biometrics and Process Control (BIOMATH), Ghent University, Belgium, 2004.
- Gürkan. Sin, Dirk J.W. De Pauw, Stefan Weijers, and Peter A. Vanrolleghem. Developing a framework for continuous use of models in daily management and operation of WWTP’s: A life cycle approach. In *Proceedings of the 10th IWA Specialised Conference on Design, Operation and Economics of Large Wastewater Treatment Plants*, Vienna, Austria, September 9–13 2007a.

- Gürkan. Sin, Dirk J.W. De Pauw, Stefan Weijers, and Peter A. Vanrolleghem. An efficient approach to automate the manual trial and error calibration of activated sludge models. *Biotechnology and Bioengineering (Submitted)*, 2007b.
- A.-M. Solvi, L. Benedetti, S. Gillé, P.M. Schosseler, A. Weidenhaupt, and P.A. Vanrolleghem. Integrated urban catchment modelling for a sewer-treatment-river system. In *Proceedings of the 10th International Conference on Urban Drainage*, Copenhagen, Denmark, August 21–26 2005.
- A.-M. Solvi, L. Benedetti, V. Vandenberghe, S. Gillé, P.M. Schosseler, A. Weidenhaupt, and P.A. Vanrolleghem. Implementation of an integrated model for optimised urban wastewater management in view of better river water quality. A case study. In *Proceedings of the 2006 IWA World Water Congress and Exhibition*, Beijing, China, September 11–15 2006.
- Anne-Marie Solvi. *Modelling the sewer-treatment-urban river system in view of the EU Water Framework Directive*. PhD thesis, Dept. of Applied Mathematics, Biometrics and Process Control (BIOMATH), Ghent University, Belgium, 2007.
- W. Spendley, G.R. Hext, and F.R. Himsworth. Sequential application of simplex designs in optimisation and evolutionary operation. *Technometrics*, 4(4):441–461, 1962.
- B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Boston, MA, 3rd edition, 1997.
- John E. Tolsma and Paul I. Barton. On computational differentiation. *Computer and Chemical Engineering*, 22(4/5):475–490, 1998.
- Robert A. van Engelen and Kyle Gallivan. The gSOAP toolkit for web services and peer-to-peer computing networks. In *Proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid2)*, pages 128–135, Berlin, Germany, May 21–24 2002.
- Stijn Van Hulle. *Modelling, simulation and optimization of autotrophic nitrogen removal processes*. PhD thesis, Dept. of Applied Mathematics, Biometrics and Process Control (BIOMATH), Ghent University, Belgium, 2005.
- Veronique Vandenberghe. *Methodologies for reduction of output uncertainty of river water quality models*. PhD thesis, Dept. of Applied Mathematics, Biometrics and Process Control (BIOMATH), Ghent University, Belgium, 2008.
- H. Vangheluwe. *Multi-formalism modelling and simulation*. PhD thesis, Department of Applied Mathematics, Biometrics and Process Control (BIOMATH), Ghent University, Belgium, 2000.
- H. Vangheluwe, E.J.H. Kerckhoffs, and G.C. Vansteenkiste. Simulation for the future: Progress of the ESPRIT basic research working group 8467. In *Proceedings of the 1996 European Simulation Symposium*, Genoa, Italy, October 24–26 1996.
- H. Vangheluwe, F. Claeys, and G.C. Vansteenkiste. The WEST++ wastewater treatment plant modelling and simulation environment. In *Proceedings of the 1998 Simulation in Industry Symposium*, Nottingham, UK, October 26–28 1998.
- H. Vanhooren, J. Meirlaen, Y. Amerlinck, F. Claeys, H. Vangheluwe, and P.A. Vanrolleghem. WEST: modelling biological wastewater treatment. *Journal of Hydroinformatics*, 5(1):27–50, 2003.
- Henk Vanhooren. *Modelling for optimisation of biofilm wastewater treatment processes: a complexity compromise*. PhD thesis, Dept. of Applied Mathematics, Biometrics and Process Control (BIOMATH), Ghent University, Belgium, 2001.

Usama El-Sayed Zaher. *Modelling and monitoring the anaerobic digestion process in view of optimisation and smooth operation of WWTP's*. PhD thesis, Dept. of Applied Mathematics, Biometrics and Process Control (BIOMATH), Ghent University, Belgium, 2005.

B. Zeigler. *Theory of Modelling and Simulation*. John Wiley & Sons, New York, NY, 1976.

Part V

Appendices



Integration Solvers

A.1 Introduction

Below an overview is given of the algorithms that have been made available as integration solver plug-ins for Tornado. Some solvers are based on simple, well-known algorithms for which a full implementation from scratch was made in the scope of Tornado. The solvers that belong to this category are:

- Euler
- Modified Euler
- Midpoint
- Heun
- Ralston
- Runge-Kutta 3
- Runge-Kutta 4
- Adams-Bashforth 2
- Adams-Bashforth 3
- Adams-Bashforth 4
- Milne

For more complex algorithms, solver codes were taken from repositories or literature, converted to C if needed (using *f2c*¹), and subsequently wrapped as a solver plug-in for Tornado. The solvers that belong to this category are:

- CVODE

¹<http://www.netlib.org/f2c>

- DOPRI5
- DOPRI853
- LSODA
- LSODE
- RK4ASC
- ROCK2
- ROCK4
- Rosenbrock
- DASSL
- DASPK
- DASRT

In the sequel, additional information on the Tornado integration solver plug-ins is provided. For solvers of the first category, a mathematical description of the algorithm that was implemented is given. For solvers of the second category, a short textual description and pointers to the source of the solver code that was wrapped are given.

A description of the symbols that are used can be found in Table A.1.

Table A.1: Integrator Solver Symbol Descriptions

Symbol	Description
α	Initial condition
y	Derived variable
f	ODE function
t_i	Time point
k	Temporary computation result
a	Butcher tableau Coefficient
b	Butcher tableau Coefficient
c	Butcher tableau Coefficient

A.2 Simple Methods

A.2.1 AE

Alg The Alg solver is not an integration solver (actually, it is not even a solver) since all it does is compute an algebraic system for a number of time points generated by a time base. In Tornado it is considered as an “integration solver” because this solution was easiest to realize in practice.

A.2.2 ODE: Explicit Single-Step Methods**Euler**

$$\begin{aligned}y_0 &= \alpha \\k_1 &= hf(t_i, y_i) \\y_{i+1} &= y_i + k_1\end{aligned}\quad i \geq 1 \tag{A.1}$$

Modified Euler

$$\begin{aligned}y_0 &= \alpha \\k_1 &= hf(t_i, y_i) \\k_2 &= hf(t_i + h, y_i + k_1) \\y_{i+1} &= y_i + \frac{1}{2}k_1 + \frac{1}{2}k_2\end{aligned}\quad i \geq 1 \tag{A.2}$$

Midpoint

$$\begin{aligned}y_0 &= \alpha \\k_1 &= hf(t_i, y_i) \\k_2 &= hf(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1) \\y_{i+1} &= y_i + k_2\end{aligned}\quad i \geq 1 \tag{A.3}$$

Heun

$$\begin{aligned}y_0 &= \alpha \\k_1 &= hf(t_i, y_i) \\k_2 &= hf(t_i + \frac{2}{3}h, y_i + \frac{2}{3}k_1) \\y_{i+1} &= y_i + \frac{1}{4}k_1 + \frac{3}{4}k_2\end{aligned}\quad i \geq 1 \tag{A.4}$$

Ralston

$$\begin{aligned}y_0 &= \alpha \\k_1 &= hf(t_i, y_i) \\k_2 &= hf(t_i + \frac{3}{4}h, y_i + \frac{3}{4}k_1) \\y_{i+1} &= y_i + \frac{1}{3}k_1 + \frac{2}{3}k_2\end{aligned}\quad i \geq 1 \tag{A.5}$$

Runge-Kutta 3

$$\begin{aligned}y_0 &= \alpha \\k_1 &= hf(t_i, y_i) \\k_2 &= hf(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1) \\k_3 &= hf(t_i + h, y_i - k_1 + 2k_2) \\y_{i+1} &= y_i + \frac{1}{6}k_1 + \frac{4}{6}k_2 + \frac{1}{6}k_3\end{aligned}\quad i \geq 1 \tag{A.6}$$

Runge-Kutta 4

$$\begin{aligned}y_0 &= \alpha \\k_1 &= hf(t_i, y_i) \\k_2 &= hf(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1) \\k_3 &= hf(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2) \\k_4 &= hf(t_i + h, y_i + k_3) \\y_{i+1} &= y_i + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4\end{aligned}\quad i \geq 1 \tag{A.7}$$

A.2.3 ODE: Explicit Multi-Step Methods

Adams-Bashforth 2

$$\begin{aligned} y_0 &= \alpha_0, y_1 = \alpha_1 \\ y_{i+1} &= y_i + h\left(\frac{3}{2}f(t_i, y_i) - \frac{1}{2}f(t_{i-1}, y_{i-1})\right) \quad i \geq 1 \end{aligned} \quad (\text{A.8})$$

Adams-Bashforth 3

$$\begin{aligned} y_0 &= \alpha_0, y_1 = \alpha_1, y_2 = \alpha_2 \\ y_{i+1} &= y_i + h\left(\frac{23}{12}f(t_i, y_i) - \frac{4}{3}f(t_{i-1}, y_{i-1}) + \frac{5}{12}f(t_{i-2}, y_{i-2})\right) \quad i \geq 2 \end{aligned} \quad (\text{A.9})$$

Adams-Bashforth 4

$$\begin{aligned} y_0 &= \alpha_0, y_1 = \alpha_1, y_2 = \alpha_2, y_3 = \alpha_3 \\ y_{i+1} &= y_i + h\left(\frac{55}{24}f(t_i, y_i) - \frac{59}{24}f(t_{i-1}, y_{i-1}) + \frac{37}{24}f(t_{i-2}, y_{i-2}) - \frac{3}{8}f(t_{i-3}, y_{i-3})\right) \quad i \geq 3 \end{aligned} \quad (\text{A.10})$$

A.2.4 ODE: Explicit Predictor-Corrector Methods

Milne

$$\begin{aligned} y_0 &= \alpha_0, y_1 = \alpha_1, y_2 = \alpha_2, y_3 = \alpha_3 \\ y_{i+1}^{(0)} &= y_{i-3} + \frac{4}{3}h(2f(t_{i-2}, y_{i-2}) - f(t_{i-1}, y_{i-1}) + 2f(t_i, y_i)) \quad i \geq 3 \\ y_{i+1}^{(k+1)} &= y_{i-1} + \frac{h}{3}(f(t_{i-1}, y_{i-1}) + 4f(t_i, y_i) + f(t_{i+1}, y_{i+1}^{(k)})) \quad i \geq 3, k \geq 0 \end{aligned} \quad (\text{A.11})$$

A.3 Complex Methods

A.3.1 ODE

CVODE

- **Description:** CVODE (Cohen and Hindmarsh, 1995; Hindmarsh et al., 2005) is a solver for stiff and non-stiff ODE systems given in explicit form. The methods used in CVODE are variable-order, variable-step multi-step methods. For non-stiff problems, CVODE includes the Adams-Moulton formulas, with the order varying between 1 and 12. For stiff problems, CVODE includes the Backward Differentiation Formulas (BDF's) in so-called fixed-leading coefficient form, with order varying between 1 and 5. For either choice of formula, the resulting non-linear system is solved (approximately) at each integration step. For this, CVODE offers the choice of either functional iteration, suitable only for non-stiff systems, and various versions of Newton iteration. In the cases of a direct linear solver (dense or banded), the Newton iteration is a Modified Newton iteration, in that the Jacobian is fixed (and usually out of date). When using SPGMR as the linear solver, the iteration is an inexact Newton iteration, using the current Jacobian (through matrix-free products), in which the linear residual is non-zero but controlled.

- **Source:** The C code for this solver was taken from the SUNDIALS suite.

- **URL:** <http://www.llnl.gov/CASC/sundials>

DOPRI5

- **Description:** DOPRI5 (Hairer et al., 1993) provides the numerical solution of a system of first order ODE's given in explicit form. The method is due to Dormand and Prince and is an explicit Runge-Kutta method of order (4)5, with stepsize control and dense output.
- **Source:** The C code for this solver was taken from the website of Ernst Hairer.
- **URL:** <http://www.unige.ch/~hairer/software.html>

DOPRI853

- **Description:** DOPRI853 (Hairer et al., 1993) provides the numerical solution of a system of first order ODE's given in explicit form. The method is due to Dormand and Prince and is an explicit Runge-Kutta method of order 8(5,3), with stepsize control and dense output.
- **Source:** The C code for this solver was taken from the website of Ernst Hairer.
- **URL:** <http://www.unige.ch/~hairer/software.html>

LSODA

- **Description:** LSODA (Petzold, 1983b) is a solver for ODE's given in explicit form, with automatic method switching for stiff and non-stiff problems. It is a variant of the LSODE package. The fact that it switches automatically between stiff and non-stiff methods implies that the user does not have to determine whether the problem is stiff or not, and the solver will automatically choose the appropriate method. The solver initially assumes that the system is non-stiff and therefore starts with the non-stiff method.
- **Source:** The C code for this solver was taken from the Computational Chemistry List (CCL) website.
- **URL:** <http://www.ccl.net/cca/software/SOURCES/C/kinetics1/lsoda.c.shtml>

LSODE

- **Description:** LSODE (Hindmarsh, 1980) is a solver for stiff and non-stiff ODE's given in explicit form. It is based on the GEAR and GEARB packages.
- **Source:** The FORTRAN code for this solver was taken from the ODEPACK collection of the NETLIB repository.
- **URL:** <http://www.netlib.org/odepack>

RK4ASC

- **Description:** RK4ASC is a variable stepsize Runge-Kutta-Fehlberg solver of order 4(5) that uses Cash-Karp coefficients. This solver is probably the most frequently used Tornado solver plug-in, because of its stability and the fact that the minimum and maximum stepsize can be fixed to a specific value.
- **Source:** The C code was taken from (Press et al., 1992).

ROCK2

- **Description:** ROCK2 (Abdulle and Medovikov, 2001) provides the numerical solution of a (mildly) stiff system of first order explicit ODE's. It is based on a family of second order explicit Runge-Kutta methods with nearly optimal stability domains on the negative real axis. The numerical method is based on a three-term recursion relation. The size (along the negative axis) of the stability domains increases quadratically with the stage number. The solver is intended for problems of large dimensions with eigenvalues of the Jacobian close to the negative real axis. Typically for problems originating from parabolic PDE's.
- **Source:** The C code for this solver was taken from the website of Ernst Hairer.
- **URL:** <http://www.unige.ch/~hairer/software.html>

ROCK4

- **Description:** ROCK4 (Abdulle and Medovikov, 2001) provides the numerical solution of a (mildly) stiff system of first order explicit ODE's. It is based on a family of fourth order explicit Runge-Kutta methods with nearly optimal stability domains on the negative real axis. The numerical method is based on a three-term recursion relation, and a composition of two sub-methods. The size (along the negative axis) of the stability domains increases quadratically with the stage number. The solver is intended for problems of large dimensions with eigenvalues of the Jacobian close to the negative real axis. Typically for problems originating from parabolic PDE's.
- **Source:** The C code for this solver was taken from the website of Ernst Hairer.
- **URL:** <http://www.unige.ch/~hairer/software.html>

Rosenbrock

- **Description:** Rosenbrock is a solver for stiff systems. The Rosenbrock method has the advantage of being rather simple to understand and implement. For moderate accuracies and moderate sized systems, it is competitive with the more complicated algorithms. For more stringent parameters, the Rosenbrock method remains reliable, but merely becomes less efficient than other methods.
- **Source:** The C code was taken from (Press et al., 1992), pp. 738–742.

A.3.2 DAE**DASSL**

- **Description:** DASSL (Petzold, 1983a) uses the backward differentiation formulas of orders one through five to solve a DAE system for Y and Y' . Values for Y and Y' at the initial time must be given as input.
- **Source:** FORTRAN code for this solver was taken from the NETLIB repository.
- **URL:** <http://www.netlib.org/ode/ddassl.f>

DASPK

- **Description:** DASPK (Brenan et al., 1989; Brown and Hindmarsh, 1989; Brown et al., 1994; Petzold, 1983a) is an improved version of DASSL that uses Krylov iteration.
- **Source:** FORTRAN code for this solver was taken from the NETLIB repository.
- **URL:** <http://www.netlib.org/ode/daspk.tgz>

A.3.3 HybridDAE

DASRT

- **Description:** DASRT (Brenan et al., 1989) is a version of DASSL extended with zero-crossing detection.
- **Source:** FORTRAN code for this solver was taken from the NETLIB repository.
- **URL:** <http://www.netlib.org/ode/ddasrt.f>

B

Root-finding Solvers

B.1 Introduction

Below is an overview of the algorithms that have been made available as non-linear equation system (*i.e.*, root-finding) solver plug-ins for Tornado. A short description and pointers to the source of the solver code that was wrapped are given.

B.2 Broyden

- **Description:** Broyden's method was the first to cheaply approximate the Jacobian of a set of equations on the basis of a quasi-Newton method (also known as secant method). Although Broyden's method was the first to introduce this technique to find the zeros of a set of equations, it is still considered to be one of the best.
- **Source:** The C code was taken from (Press et al., 1992), pp. 389–392.

B.3 Hybrid

- **Description:** The purpose of the Hybrid (HYBRD1) algorithm is to find a zero of a system of n non-linear equations in n variables by a modification of the Powell hybrid method. This is done by using the more general non-linear equation solver HYBRD. The user must provide a subroutine which calculates the functions. The Jacobian is then calculated by a forward difference approximation.
- **Source:** The FORTRAN code for this solver was taken from the MINPACK collection of the NETLIB repository.
- **URL:** <http://www.netlib.org/minpack>



Optimization Solvers

C.1 Introduction

The algorithms that have been made available as optimization solver plug-ins for Tornado are reviewed below. The Praxis, Simplex and SA codes that are used in Tornado are modified versions of algorithms that can be found in literature. The GA algorithm was entirely developed from scratch. For each algorithm a description is given below. Mixed with the description, pointers to relevant literature are given.

C.2 Praxis

- **Description:** The Praxis (PRincipal AXIS) algorithm (see also (De Pauw, 2005), Chapter 5) was first described in (Brent, 1973) and is an enhanced version of the direction set method of Powell. It is based on a repeated combination of one-dimensional searches along a set of various directions. Updating the search directions is done by finding those directions that will lead far along narrow valleys or that are non-interfering, which means that they have the special property that a minimization along one direction is not spoiled by subsequent minimizations along another one. Brent also incorporated some random jumps into the procedure to avoid some of the local minima problems that Powell's original algorithm suffered from. R.P. Brent gave an ALGOL-W program. A C-version was written by Karl Gegenfurtner and can be found in (Gegenfurtner, 1992). The version that was implemented in Tornado is a revision of the code by Karl Gegenfurtner.

C.3 Simplex

- **Description:** The Simplex minimization method (see also (De Pauw, 2005), Chapter 5) was first described in (Spendley et al., 1962) and was later significantly improved in (Nelder and Mead, 1965) by allowing irregular simplexes. A simplex is a geometrical figure consisting, in p dimensions, of $p + 1$ points (or vertices) interconnected by line segments forming polygonal faces. For a 2-variable optimization problem, the simplex is a triangle. Each of these points represents a set of optimization variable values and corresponds with one objective function value. Starting from an initial simplex, elementary operations are performed on the vertex with the highest value (for

a minimization problem) resulting in a new vertex in the p -dimensional space. Elementary operations include reflection, expansion and contraction of the vertices and can be used in order to migrate the simplex in the direction of the minimum, by each time replacing the worst performing vertices by new ones. The algorithm is terminated if the relative difference between the objective function values of the vertices and the average value of the whole simplex are below a certain threshold value. The algorithm that is implemented in Tornado is a modified version of the code that can be found in (Press et al., 1992).

C.4 GA

- **Description:** The Genetic Algorithm (GA) code for real-valued variables that is used in Tornado is due to (De Pauw, 2005) and is based on the general principles regarding genetic algorithms that can be found throughout literature. More specifically, BLX-alpha crossover (Herrera et al., 1998) is used in conjunction with linear scaling (Goldberg, 1989) and roulette wheel selection (Goldberg, 1989).

C.5 SA

- **Description:** The Simulated Annealing (SA) code for continuous variables that is used in Tornado is due to Dirk De Pauw (Ghent University, Belgium) and is based on the code that can be found in (Press et al., 1992). Modifications were made related to the cooling schedule (Cardoso et al., 1996), initial temperature determination (Cardoso et al., 1996) and constraint handling (custom code).



Monte Carlo Solvers

D.1 Introduction

This section provides an overview of the algorithms that have been made available as so-called Monte Carlo solver plug-ins for Tornado. These solvers select a predefined number of points from a multi-dimensional parameter space at random. The selection is such that the points are equally spread over the parameter space. For each solver, a short description and pointers to the source of the solver code that was wrapped are given.

D.2 CVT

- **Description:** CVT is a library for creating Centroidal Voronoi Tessellation (CVT) datasets (Du et al., 1999). A centroidal Voronoi tessellation is a Voronoi tessellation of a given set such that the associated generating points are centroids (centers of mass with respect to a given density function) of the corresponding Voronoi regions. The generation of a CVT dataset is of necessity more complicated than the generation of a quasi-random sequence. An iteration is involved, so there must be an initial assignment for the generators, and then a number of iterations. Moreover, in each iteration, estimates must be made of the volume and location of the Voronoi cells. This is typically done by Monte Carlo sampling. The accuracy of the resulting CVT depends in part on the number of sampling points and the number of iterations taken. The library is mostly used to generate a dataset of points uniformly distributed in the unit hypersquare. However, a user may be interested in computations with other geometries or point densities. To do this, the user needs to replace the `USER` routine in the CVT library, and then specify the appropriate values `init=3` and `sample=3`. The `USER` routine returns a set of sample points from the region of interest. The default `USER` routine samples points uniformly from the unit circle. But other geometries are easy to set up. Changing the point density simply requires weighting the sampling in the region.
- **Source:** The C++ code was taken from the website of John Burkardt, the author of the implementation
- **URL:** http://people.scs.fsu.edu/~burkardt/cpp_src/cvt/cvt.html

D.3 IHS

- **Description:** IHS is a library that carries out the Improved distributed Hypercube Sampling (IHS) algorithm (Beachkofski and Grandhi, 2002). N points in an M -dimensional Latin hypercube are to be selected. Each of the M coordinate dimensions is discretized to the values 1 through N . The points are to be chosen in such a way that no two points have any coordinate value in common. This is a standard Latin hypercube requirement, and there are many solutions. This algorithm differs in that it tries to pick a solution which has the property that the points are “spread out” as evenly as possible. It does this by determining an optimal even spacing, and using the duplication factor D to allow it to choose the best of the various options available to it (the more duplication, the better chance of optimization). One drawback to this algorithm is that it requires an internal real distance array of dimension $D \times N \times N$. For a relatively moderate problem with $N = 1000$, this can exceed the easily accessible memory. Moreover, the program is inherently quadratic in N in execution time as well as memory; the computation of the I^{th} point in the set of N requires a consideration of the value of the coordinates used up by the previous points, and the distances from each of those points to the candidates for the next point.
- **Source:** The C++ code was taken from the website of John Burkardt, the author of the implementation
- **URL:** http://people.scs.fsu.edu/~burkardt/cpp_src/ihs/ihs.html

D.4 LHS

- **Description:** The Latin Hypercube Sampling (LHS) method may be considered a particular case of stratified sampling. The purpose of stratified sampling is to achieve a better coverage of the sample space of the input factors with a relatively small number of samples.
- **Source:** SimLab¹ manual
- **URL:** <http://simlab.jrc.cec.eu.int/docs/html/latin.html>

D.5 PR

- **Description:** The Pure Random (PR) method is a very simple algorithm that was not acquired from external sources but entirely implemented in the scope of the Tornado project. It simply picks S points from a N -dimensional space, without imposing any constraints on spread of the points selected.

¹<http://simlab.jrc.cec.eu.int>



Features of Domain-Specific Water Quality Tools

In Table E.1 and Table E.2, an overview is given of the features of a number of domain-specific modelling and virtual experimentation tools for water quality management. The tables were created on the basis of information provided by the vendors of the respective tools:

- BioWin: EnviroSim, Inc.
7 Innovation Drive, Suite 5
Flamborough, Ontario
L9H 7H9 Canada
<http://www.envirosim.com>
- GPS-X: Hydromantis Inc.
1 James Street South, Suite 1601
Hamilton, Ontario
L8P 4R5 Canada
<http://www.hydromantis.com>
- STOAT: WRc plc
Frankland Road
Blagrove, Swindown, Wiltshire
SN5 8YF United Kingdom
<http://www.wrcplc.co.uk>
- WEST: MOSTforWATER N.V.
Koning Leopold III-laan 2
B-8500 Kortrijk
Belgium
<http://www.mostforwater.com>

The questionnaires sent to the vendors of the Aquasim and Simba products were not returned. Hence, no information on these tools is listed in the tables.

Table E.1: Features of Domain-Specific Water Quality Tools (Part 1)

	BioWin	GPS-X	STOAT	WEST-3
Model Representation				
Graphical	Yes	Yes	Yes	Yes
Petersen Matrix	Yes	Yes	No	Yes
Textual Language	No	ACSL	No	MSL
Equation Types				
AE	Yes	Yes	Yes	Yes
ODE	Yes	Yes	Yes	Yes
DDE	Yes	No	No	Yes
DAE	Yes	Yes	No	No
PDE	Yes	Yes	Yes	No
SDE	No	No	No	No
Hybrid	No	Yes	No	No
Virtual Experiment Types				
Steady-state Analysis	Yes	Yes	No	No
Dynamic Simulation	Yes	Yes	Yes	Yes
Sensitivity Analysis	No	Yes	Yes	Yes
Parameter Estimation	No	Yes	Yes	Yes
Scenario Analysis	No	Yes	Yes	Yes
Monte Carlo Analysis	No	Yes	No	Yes
Uncertainty Analysis	No	Yes	No	No
Optimal Experiment Design	No	No	No	No
Numerical Solvers	Proprietary steady-state solver, 7 Integration solvers	Proprietary steady-state solver, Euler, RK2, RK4, Gear, Adams-Moulton, RKF23, RKF45, DASSL	RK*, Gear, Adams	Euler, Heun, Milne, Modified Euler, RK4, RKF45, Adams-Bashforth*, Rosenbrock, VODE
Water Quality Models				
ASM1	Yes	Yes	Yes	Yes
ASM2	No	No	Yes	Yes
ASM2d	Yes	Yes	Yes	Yes
ASM3	Yes	Yes	Yes	Yes
ADM1	Yes	Yes	No	Yes
RWQM1	No	No	No	Yes
Biofilm	Yes	Yes	Yes	Yes
Sensors	No	No	Yes	Yes
Controllers	Yes	Yes	Yes	Yes
Other	Proprietary ASDM	Prefermenter, Proprietary Digester, New General, Mantis, 2-step Mantis, Double Exponential Settling, Reactive Settling, MBR, Sand Filters, PAC, HPO, Oxidation Ditch, SBR, Deep Shaft, DAF, Dewatering	BOD-based ASM, Mosey ADM, TAD	Settling, Reactive Settling, MBR, Sand Filters, SBR, Oxidation Ditch, Dewatering
User Interfaces				
Graphical	Yes	Yes	Yes	Yes
Command-line	No	Yes	No	No
Web-based	No	No	No	No

Table E.2: Features of Domain-Specific Water Quality Tools (Part 2)

	BioWin	GPS-X	STOAT	WEST-3
Input Providers				
Files				
Flat Text	Yes	Yes	Yes	Yes
Excel	Yes	Yes	No	No
Graphical				
Sliders	No	Yes	No	Yes
Switches	No	Yes	No	No
Database	Internal	SQL	MS Access	No
On-line	No	Yes	No	No
Output Acceptors				
Files				
Flat Text	Yes	Yes	Yes	Yes
Excel	Yes	Yes	No	No
Graphical				
Tables	Yes	Yes	Yes	Yes
Line Graphs	Yes	Yes	Yes	Yes
Bar Charts	Yes	Yes	Yes	Yes
Pie Charts	Yes	No	Yes	Yes
3D Graphs	Yes	Yes	Yes	Yes
Database	Internal	SQL	No	No
On-line	No	Yes	No	No
API's				
C/C++	No	No	No	No
.NET	No	No	No	No
Java	No	No	No	No
COM	Yes	No	Yes	Yes
OpenMI	No		Yes	No
Other		FORTRAN, DDE, MATLAB		
Remote Execution				
Sockets	No	No	No	No
CORBA	No	No	No	No
DCOM	No	No	Yes	Yes
OPC	No	Yes	No	No
Java RMI	No	No	No	No
SOAP	No	No	No	No
.NET Remoting	No	No	No	No
Distributed Execution				
Cluster Computing	No	No	No	No
Grid Computing	No			
Platforms				
Windows 2000	Yes	Yes	Yes	Yes
Windows XP	Yes	Yes	Yes	Yes
Windows Vista	Yes	Yes	Yes	Yes
Linux	No	No	No	No
Mac	No	No	No	No
Miscellaneous				
Workflow / Version Management	No	No	Yes	No
Model Encryption	No	Yes	No	Yes
Development Environment	Delphi	FORTRAN, Java	FORTRAN, Visual Basic	C, Delphi
Product Versions	Standard	Entry, Advanced, Professional, Enterprise	Standard	S, C, P, S.A, SCA, API



List of Abbreviations

AE:	Absolute Error
AE:	Algebraic Equation
AOT:	Ahead Of Time compilation
API:	Application Programming Interface
ASDM:	Activated Sludge Digestion Model
ASM:	Activated Sludge Model
AST:	Abstract Syntax Tree
ASU:	Activated Sludge Unit
BAS:	Backward Absolute Sensitivity
BDF:	Backward Differentiation Formula
BPRSQ:	Backward Partial Relative Sensitivity with respect to Quantity
BPRSV:	Backward Partial Relative Sensitivity with respect to Variable
BRS:	Backward Relative Sensitivity
BSM:	Benchmark Simulation Model
CAD:	Computer Aided Design
CAE:	Computer Aided Engineering
CAS:	Central Absolute Sensitivity
CBD:	Causal Block Diagram
CFD:	Computational Fluid Dynamics
CIL:	Common Intermediate Language

CLI:	Common Language Infrastructure
CLR:	Common Language Run-time
CLS:	Common Language Specification
COD:	Chemical Oxygen Demand
CPRSQ:	Central Partial Relative Sensitivity with respect to Quantity
CPRSV:	Central Partial Relative Sensitivity with respect to Variable
CPU:	Central Processing Unit
CRS:	Central Relative Sensitivity
CSO:	Combined Sewer Overflow
CTS:	Common Type System
CUI:	Command-line User Interface
CVT:	Centroidal Voronoi Tessellation
DAE:	Differential-Algebraic Equation
DCS:	Distributed Control System
DDE:	Delay Differential Equation
DSL:	Domain-Specific Language
ExpCI:	Confidence Information Analysis Experiment
ExpEnsemble:	Ensemble Simulation Experiment
ExpMCOptim:	Optimization Monte Carlo Analysis Experiment
ExpMC:	Monte Carlo Analysis Experiment
ExpObjEval:	Objective Evaluation Experiment
ExpOptim:	Optimization Experiment
ExpSSOptim:	Steady-state Analysis Experiment with Optimizer
ExpSSRoot:	Steady-state Analysis Experiment with Root Finder
ExpScenOptim:	Optimization Scenario Analysis Experiment
ExpScenSSRoot:	Steady-state Analysis with Root Finder Scenario Analysis Experiment
ExpScen:	Scenario Analysis Experiment
ExpSens:	Sensitivity Analysis Experiment
ExpSeq:	Sequential Experiment Experiment
ExpSimul:	Dynamic Simulation Experiment
ExpStats:	Statistical Analysis Experiment

FAS: Forward Absolute Sensitivity

FPRSQ: Forward Partial Relative Sensitivity with respect to Quantity

FPRSV: Forward Partial Relative Sensitivity with respect to Variable

FRS: Forward Relative Sensitivity

FSA: Finite State Automata

GA: Genetic Algorithm

GUI: Graphical User Interface

HGE: Hierarchical Graph Editor

HMI: Human-Machine Interface

HTTPS: Secure Hyper Text Transfer Protocol

HTTP: Hyper Text Transfer Protocol

HiL: Hardware in the Loop

IDE: Integrated Development Environment

IDL: Interface Definition Language

IHS: Improved distributed Hypercube Sampling

IJW: It Just Works

IMMC: Inter Macro Module Communication

IUWS: Integrated Urban Wastewater System

JDBC: Java Data Base Connectivity

JDL: Job Description Language

JIT: Just In Time compilation

JNI: Java Native Interface

JRMP: Java Remote Method Protocol

JSDL: Job Submission Description Language

LHS: Latin Hypercube Sampling

LHS: Left Hand Side

MAE: Mean Absolute Error

MDI: Multiple Document Interface

MPC: Model Predictive Control

MRE: Mean Relative Error

NMF: Neutral Model Format

ODE: Ordinary Differential Equation

OS: Operating System

OpenMI: Open Modelling Interface

PDE: Partial Differential Equation

PE: Person Equivalent

PLC: Programmable Logic Controller

PR: Pure Random

PWM: Plant-wide Modelling

RCP: Real-time Control Program

RE: Relative Error

RHS: Right Hand Side

RMI: Remote Method Invocation

RPC: Remote Procedure Call

RWQM: River Water Quality Model

SA: Simulated Annealing

SCADA: Supervisory Control And Data Acquisition

SCM: Source Code Management

SDE: Stochastic Differential Equation

SDK: Software Development Kit

SE: Squared Error

SOAP: Simple Object Access Protocol

STL: Standard Template Library

ST: Symbol Table

TCP: Transfer Control Protocol

TIC: Theil's Inequality Coefficient

TLM: Transmission Line Modelling

UDP: User Datagram Protocol

VCS: Version Control System

VES: Virtual Execution System

WFD: European Water Framework Directive

WSDL: Web Service Description Language

WWTP: Waste Water Treatment Plant



Summary

Computer-based modelling and virtual experimentation (*i.e.*, any procedure in which the evaluation of a model is required, such as simulation and optimization) has proven to be a powerful mechanism for solving problems in many areas, including environmental science. Since the late 1960's, a *plethora* of software systems have been developed that allow for modelling, simulation, and to a lesser extent, also other types of virtual experimentation. However, given the persistent desire to model more complex systems, and the trend towards more complex computational procedures based on model evaluation, it may be required to re-evaluate and improve existing software frameworks, or even to suggest new frameworks. Moreover, recent developments in information technology have caused a growing trend towards flexible deployment and integration of software components, *e.g.*, in a web-based, distributed or embedded context. In order to be able to handle the need for malleable deployment and integration of modelling and virtual experimentation systems with other software components, re-evaluation of existing frameworks and/or development of new frameworks may also be required.

One particular domain of environmental science that has in recent years attracted the interest of researchers in the field of software frameworks, is water quality management (which, amongst other, includes the study of water quality in treatment plants, sewer networks and river systems). This domain can be considered mature since the inception of standard unit process models such as the Activated Sludge Model (ASM) series. Moreover, the complexity of the most advanced integrated models that are currently studied in this domain is such that mainstream software systems simply cannot handle them anymore.

The work that is presented in this dissertation concerns the design and implementation of an advanced framework for modelling and virtual experimentation with complex environmental systems, which attempts to accomplish the following goals: deliver support for complex modelling and complex virtual experimentation, offer a wide variety of deployment and integration options, and ensure compliance with modern architectural standards. As most systems that are studied in environmental science are continuous, focus is on complex continuous system modelling.

The framework that was developed is generic in nature, although the design and implementation process has been strongly guided by demands coming from the field of water quality modelling. Development was done over a longer period of time, *i.e.*, mainly from 1995 until 1998 and from 2003 until 2007. The WEST modelling and virtual experimentation product that is currently commercially available (and is mainly used for modelling and simulation of wastewater treatment plants) is an emanation of the work that was done during the first period. The work done during the second period, has resulted into a

software framework named “Tornado” and should eventually find its way to WEST-4, which is a major rewrite of the former WEST-3 product.

Tornado implements a hierarchical virtual experimentation framework in which new virtual experiments can be created by re-using already existing experiments. The composition graph that represents the virtual experiments that have so far been developed consists of 15 different types of experiments. Using these experiments, a broad range of problems can be solved, ranging from simple simulation studies to large-scale risk analyses using comprehensive Monte Carlo simulation, such as recently performed in the scope of the CD4WC project dealing with support for the EU Water Framework Directive. All experiment types are highly configurable through an extensive set of properties, which can be dynamically queried and modified. New experiment types can be added by computer scientists or environmental scientists, provided that the design and implementation principles of Tornado are well-understood.

Most virtual experiment types in Tornado are guided by numerical solver algorithms. For the incorporation of these solvers into the kernel, a generalized framework for abstraction and dynamic loading of solver plug-ins was devised. This framework allows for solvers to be classified according to their purpose (integration, optimization, *etc.*) and for new solvers to be developed on the basis of a library of base classes. At run-time, solvers can be dynamically loaded or removed from the system for reducing memory consumption.

The flexibility and orthogonal design of Tornado is well illustrated by its ability to perform solver setting scenario analysis, which is a method that allows for applying variable sweeping techniques to solver settings (such as integrator tolerances), instead of to model parameters. Using this technique, overall execution times of large-scale compound virtual experiments can be shortened by applying improved solver settings that are discovered through *a priori* exploration of the solver setting space.

In the scope of Tornado, two declarative, hierarchical, object-oriented, equation-based modelling languages play an important role: MSL and Modelica. MSL has been available as of the first Tornado version and has been the cornerstone of model development in the scope of many water quality modelling studies. Since MSL does not support the concepts of acausal and multi-domain modelling, the Modelica language was introduced in a later version of the framework. Albeit very powerful, Modelica is also a complex language. Consequently, a hybrid approach is adopted in which the standard OpenModelica model compiler front-end is used in combination with a custom model compiler back-end for Tornado. This model compiler back-end allows for the generation of executable models that can be used by the Tornado virtual experimentation kernel, as well as for the generation of S-functions that can be adopted in the scope of MATLAB/Simulink.

Since executable models need to be fast to generate, the Tornado executable model format was designed in such a way that meta-information is represented in XML and computational information is described in a compiled general-purpose programming language, *i.e.*, C. In this way, executable models can quickly be produced, using the user’s C compiler of choice.

In order to ensure the run-time performance of executable models, three techniques were implemented that reduce the model complexity and hence provide a considerable performance improvement of 10 to 30%. These techniques are known as constant folding, equiv substitution, and lifting of equations. The first computes the results of constant sub-expressions at compile-time, the second removes aliases from a set of equations, and the third allows for detecting initial and output equations and for moving them to appropriate equation sections.

As a result of the manipulations that are performed on models during the executable code generation process, the output of this process is often not recognizable anymore to a user. Consequently, stability is an important issue: in case a problem does occur, it must be reported in a way that is understandable to a user. Two techniques were implemented that allow for improved stability and error reporting in executable models: code instrumentation and the generation of bounds checking code. The first replaces potentially dangerous constructs (*e.g.*, division-by-zero) by “safe” versions in which arguments are first

checked on validity before being applied. The second generates code that performs range checking at run-time (*e.g.*, to detect negative values). By generating this code only for variables that need it, a performance improvement of up to 50% could be realized with respect to kernel-based range checking. In general, a situation is created in which run-time performance of safe, optimized Tornado models is comparable to the performance of unsafe, non-optimized models. Hence, the time spent on performing safety checks is compensated by the time gained through optimization.

In order to allow for the integration of Tornado in other software, several application programming interfaces (API's) were developed. Some of these are comprehensive and allow for any Tornado-related operation to be performed, while others are restricted and only allow for the most frequently occurring operations. Comprehensive API's were developed for C++ and .NET. Restricted API's were developed for C, the Java Native Interface, MATLAB MEX, OpenMI and CLIPS. A special case is the CMSLU interface that allows for the Tornado kernel to be called to run any type of experiment from within a Tornado model.

Several stand-alone applications were developed on the basis of the Tornado kernel. One of these applications is a suite of command-line programs that is developed along with the Tornado kernel itself and is useful for testing, automation and advanced use. Next to the command-line suite, a number of graphical applications were developed: WEST++, WEST-3 and EAST are based on the initial version of Tornado, while MORE and WEST-4 are based on the most recent version. These graphical applications illustrate that for application development, various technologies can be applied to one and the same Tornado kernel.

With respect to remote and web-based use, a number of technologies were discussed that are potentially applicable to Tornado. Some of these technologies have already been used in the scope of applications (sockets, DCOM and SOAP), while others have merely been adopted in prototype projects (.NET Remoting, ASP.NET). Still some others have not yet been applied, but could in case a need for this would arise (CORBA, OPC, Java RMI).

Finally, coarse-grained gridification was applied to Tornado at the level of sub-experiments (*i.e.*, a set of simulation jobs is carried out concurrently on a pool of computational nodes). As a result, Tornado can be deployed in a distributed fashion on the basis of the Typhoon cluster software (which was developed especially to serve as a light-weight solution for the execution of jobs generated by Tornado), as well as on the basis of the gLite grid middleware. Through the application of these distributed technologies, the extreme computational workload (14,400 simulations each requiring 30' of computation time) that was a result of the CD4WC project could be processed in a timely fashion (approximately 10 days).

Tornado was developed from scratch using object-oriented design and implementation principles. C++ was used as a programming language and design patterns such as the singleton, factory, facade and proxy patterns were adopted. Platform-independence of the kernel (but not necessarily its external interfaces) was ensured and thread-safety for high-level entities was provided. For these high-level entities, XML-based persistent formats were devised and described on the basis of XML Schema Definitions. Furthermore, a mechanism that provides for run-time querying of entity properties was provided, which alleviates the need for modification of entity interfaces in case of changes to the set of properties supported by an entity.

Overall, it can be stated that through the design and development of the Tornado framework, the solution of water quality management problems that were hitherto hindered by performance limitations or limitations to the degree to which complexity could be handled, has now become possible. Moreover, thanks to the design of the framework, it is expected that Tornado will be able to adapt to the expected continued increase in complexity for a considerable period of time. In order to render the Tornado framework more complete, a need for future work and the exploration of additional research areas exists. In some cases, this additional research has already been initiated.

H

Samenvatting

Computer-gebaseerde modellering en virtuele experimentering (*i.e.*, elke procedure waarin de evaluatie van een model vereist is, zoals bijvoorbeeld simulatie en optimalisatie) heeft zich tot dusver bewezen als een krachtig mechanisme voor het oplossen van problemen in verschillende domeinen, inclusief milieutechnologie. Sinds het einde van de jaren '60 werden een groot aantal software-systemen ontwikkeld die modellering, simulatie en in mindere mate ook andere types van virtuele experimentering toelaten. Echter, gezien de aanhoudende wens tot het modelleren van complexere systemen en de trend naar steeds complexere procedures gebaseerd op modevaluatie, kan het noodzakelijk zijn om bestaande software-systemen te herevalueren of te verbeteren, of zelfs nieuwe raamwerken voor te stellen. Bovendien hebben recente ontwikkelingen in informatietechnologie een groeiende trend veroorzaakt naar flexibele ontplooiing en integratie van software-componenten, bijvoorbeeld in een web-gebaseerde of gedistribueerde context. Om deze nood aan kneedbare ontplooiing en integratie van modelleer- en virtuele experimenteersystemen aan te kunnen, kan herevaluatie van bestaande raamwerken en ontwikkeling van nieuwe raamwerken eveneens vereist zijn.

Eén van de milieutechnologie-domeinen die de laatste jaren de aandacht hebben getrokken van onderzoekers geïnteresseerd in software-raamwerken, is de studie van de waterkwaliteit in riviersystemen, rioolsystemen en zuiveringsinstallaties. Dit domein kan als matuur beschouwd worden sinds het ontstaan van standaard modellen voor basisprocessen zoals de ASM (Activated Sludge Model) serie. Bovendien is de complexiteit van de meest geavanceerde modellen die momenteel in dit domein worden behandeld dusdanig dat klassieke software-systemen ze niet meer kunnen aankunnen.

Het werk dat in dit proefschrift wordt gepresenteerd behelst de vereisten, het ontwerp en de implementatie van een geavanceerd raamwerk voor modelleren en virtueel experimenteren met complexe milieusystemen. Doelstellingen van dit raamwerk zijn: het bieden van ondersteuning voor complex modelleren en complex virtueel experimenteren, het aanbieden van een brede waaier van opties voor ontplooiing en integratie en het verzekeren van overeenstemming met moderne architecturale standaarden. Vermits de meeste systemen die in de milieuwetenschappen bestudeerd worden continu zijn, ligt de focus van het raamwerk op complexe continue systemen.

Het raamwerk dat werd ontwikkeld is generiek van aard, hoewel het ontwerp- en implementatieproces sterk werd gedreven door vereisten uit het waterkwaliteitsdomein. De ontwikkeling vond plaats gedurende een vrij lange tijdspanne, meer bepaald van 1995 tot 1998 en van 2003 tot 2007. De WEST modelleer- en virtuele experimenteersomgeving die momenteel commercieel beschikbaar is (en die voornamelijk gebruikt wordt voor het modelleren en simuleren van waterkwaliteitssystemen) is een voortvloei-

sel van het werk dat werd verricht gedurende de eerste periode. Het werk dat tijdens de tweede periode werd verricht heeft aanleiding gegeven tot een software-raamwerk dat “Tornado” werd gedoopt en zou uiteindelijk zijn weg moeten vinden naar WEST-4, wat een belangrijke herimplementatie is van het originele WEST-3 product.

Tornado implementeert een hiërarchisch virtueel experimenteer-raamwerk waarin nieuwe virtuele experimenten kunnen gecreëerd worden op basis van reeds bestaande experimenten. De compositiegraaf die de tot dusver ontwikkelde virtuele experimenttypes voorstelt omvat 15 componenten. Op basis van deze experimenten kan een breed scala aan problemen opgelost worden, gaande van eenvoudige simulatiestudies tot grootschalige risico-analyses gebaseerd op Monte Carlo simulatie, zoals recent uitgevoerd in het kader van het CD4WC project dat nauw verbonden is met het EU Water Framework Directive. Alle experimenttypes zijn in hoge mate configureerbaar dankzij een uitgebreide verzameling attributen die dynamisch kunnen worden bevraagd en aangepast. Nieuwe experimenttypes kunnen worden toegevoegd door computerwetenschappers of door milieuwetenschappers, op voorwaarde dat deze laatsten de ontwerp- en implementatieprincipes van Tornado goed beheersen.

De meeste virtuele experimenttypes in Tornado worden gestuurd door numerieke algoritmes. Voor het incorporeren van deze algoritmes in de kernel werd een veralgemeend raamwerk voor abstractie en dynamisch opladen van numerieke solver plug-ins ontworpen. Dit raamwerk laat toe om solvers te klassificeren op basis van hun doel (integratie, optimalisatie, enz.) en om nieuwe solvers te ontwikkelen op basis van een bibliotheek met basisklassen. Tijdens de uitvoering kunnen solvers dynamisch worden geladen, of worden verwijderd uit het systeem om het geheugengebruik te beperken.

De flexibiliteit en het orthogonale ontwerp van Tornado worden goed geïllustreerd door de mogelijkheid tot het uitvoeren van zogenaamde solver setting scenario analyse. Deze methode laat toe om klassieke variable sweeping technieken toe te passen op solver settings (zoals integrator tolerances) in plaats van op model parameters. Gebruikmakende van deze techniek kunnen de totale uitvoeringstijden van virtuele experimenten verkort worden door het toepassen van meer optimale solver settings, opgespoord door het uitvoeren van een a priori exploratie van de solver setting ruimte.

In het kader van Tornado spelen twee declaratieve, hiërarchische, object-georiënteerde en op wiskundige vergelijkingen gebaseerde modelleertalen een belangrijke rol: MSL en Modelica. MSL is reeds beschikbaar sinds de eerste versie van Tornado en was tot dusver de hoeksteen van modelontwikkeling in het kader van menige waterkwaliteitsstudie. Vermits MSL de concepten van acausale en multi-domeinmodellering niet ondersteunt, werd de Modelica-taal geïntroduceerd in een latere versie van het raamwerk. Modelica is een zeer krachtige, maar ook zeer complexe taal. Bijgevolg werd een hybride aanpak gehanteerd waarbij de standaard OpenModelica model compiler front-end gebruikt wordt in combinatie met een custom model compiler back-end voor Tornado. Deze model compiler back-end laat enerzijds toe om uitvoerbare modellen te genereren die kunnen gebruikt worden door de Tornado virtuele experimenteerkernel en anderzijds om S-functies aan te maken die kunnen worden toegepast in de context van MATLAB/Simulink.

Vermits uitvoerbare modellen snel moeten kunnen worden gegenereerd, werd het uitvoerbare modelformaat van Tornado dusdanig ontworpen dat meta-informatie in XML wordt voorgesteld en computationele informatie wordt gerepresenteerd in een programmeertaal voor algemene doeleinden, namelijk C. Op deze manier kunnen efficiënte binaire uitvoerbare modellen snel worden geproduceerd, gebruikmakende van de door de gebruiker gekozen C compiler.

Om de run-time performantie van uitvoerbare modellen te garanderen, werden drie technieken geïmplementeerd die de modelcomplexiteit reduceren en een performantievoordeel opleveren van 10 tot 30%. Deze technieken zijn constant folding, equiv substitution en equation lifting. De eerste techniek berekent de resultaten van constante expressies reeds tijdens de compilatie, de tweede verwijdert aliases uit de verzameling vergelijkingen en de derde laat toe om initiële en output vergelijkingen op te sporen en deze te verplaatsen naar de meest geschikte secties.

Als gevolg van de manipulaties die worden doorgevoerd op modellen gedurende de generatie van uitvoerbare code, is het resultaat van dit proces dikwijls niet meer herkenbaar voor een gebruiker. Bijgevolg is stabiliteit een belangrijk punt: in geval een probleem optreedt, moet dit gerapporteerd worden op een manier die begrijpbaar is voor de gebruiker. Twee technieken werden geïmplementeerd die zorgen voor verbeterde stabiliteit en foutrapportering in uitvoerbare modellen: code instrumentatie en de generatie van code voor het testen van grenzen. De eerste techniek vervangt potentieel gevaarlijke constructies (bijvoorbeeld delingen) door veilige versies waarin argumenten eerst getest worden vooraleer te worden toegepast. De tweede techniek genereert code die tijdens de uitvoering test of waarden van variabelen zich tussen de vooropgestelde grenzen bevinden (bijvoorbeeld om negatieve getallen op te sporen). Door deze code enkel te genereren voor variabelen waarvoor de testen effectief zinvol zijn, kan een performantieverbetering tot 50% bereikt worden ten opzichte van een implementatie waarbij de testen onconditioneel door de kernel worden uitgevoerd. In het algemeen werd een situatie gecreëerd waarin de uitvoeringsefficiëntie van veilige, geoptimaliseerde code vergelijkbaar is met de efficiëntie van onveilige, niet-geoptimaliseerde code. Bijgevolg kan gesteld worden dat de tijd die gespendeerd wordt aan het uitvoeren van stabiliteitstesten wordt gecompenseerd door de tijd die gewonnen wordt door optimalisatie.

Met het oog op de integratie van Tornado in andere software werden verschillende Application Programming Interfaces (API's) ontwikkeld. Sommige interfaces zijn uitgebreid en laten toe om alle Tornado-gerelateerde operaties uit te voeren; andere zijn eerder beperkt en laten enkel de meest belangrijke operaties toe. Uitgebreide interfaces werden ontwikkeld voor C++ en .NET, terwijl beperkte interfaces werden uitgewerkt voor C, de Java Native Interface (JNI), MATLAB-MEX, OpenMI en CLIPS. De zogenaamde CMSLU interface is een speciaal geval vermits deze interface toelaat om de Tornado-kernel aan te spreken voor het uitvoeren van virtuele experimenten vanuit een uitvoerbaar model.

Verscheidene op zichzelf staande applicaties werden ontwikkeld op basis van de Tornado-kernel. Eén van deze applicaties is een suite van commandolijnprogramma's die werd uitgewerkt tezamen met de Tornado-kernel en nuttig is voor testen, automatisatie en geavanceerd gebruik. Voorts werden ook een aantal grafische applicaties ontwikkeld: WEST++, WEST-3 en EAST zijn gebaseerd op de initiële versie van Tornado, terwijl MORE en WEST-4 gebaseerd zijn op de meest recente versie. Deze grafische programma's illustreren dat voor het ontwikkelen van applicaties, verschillende technologieën kunnen worden gebruikt in combinatie met eenzelfde Tornado-kernel.

Met betrekking tot remote en web-gebaseerd gebruik werden een aantal technologieën bediscussieerd die potentieel toepasbaar zijn op Tornado. Sommige van deze technologieën werden reeds gebruikt in het kader van applicaties (sockets, DCOM en SOAP), terwijl andere slechts werden toegepast in prototype-projecten (.NET Remoting en ASP.NET). Een aantal technologieën werden nog niet toegepast, maar zouden vanuit technisch oogpunt kunnen worden aangewend indien nodig (CORBA, OPC en Java RMI).

Tenslotte werd gridificatie met een ruwe granulariteit toegepast op Tornado op het niveau van sub-experimenten (een set van simulatie-jobs wordt concurrent uitgevoerd op een poule van computationele eenheden). Bijgevolg kan Tornado worden ontplooid op een gedistribueerde wijze, meer bepaald gebruikmakende van de Typhoon cluster software (dewelke specifiek werd ontwikkeld als lichtgewicht oplossing voor het uitvoeren van Tornado jobs) evenals van de gLite grid middleware. Door de toepassing van deze gedistribueerde technologieën, kon de extreme werklust (14.400 simulaties die elk gemiddeld 30' rekentijd vergen) die werd veroorzaakt door het CD4WC project worden behandeld in ongeveer 10 dagen.

Tornado werd van de grond af ontwikkeld op basis van object-georiënteerde ontwerp- en implementatieprincipes. Als programmeertaal werd C++ gebruikt en ontwerp patronen zoals singleton, factory, facade en proxy werden aangewend. De platform-onafhankelijkheid van de kernel (maar niet noodzakelijk zijn interfaces) werd verzekerd en voor de belangrijkste entiteiten werd thread-safety voorzien. Voor deze entiteiten werden tevens XML-gebaseerde persistente formaten opgesteld en beschreven op basis van XML Schema Definities (XSD). Voorts werd een mechanisme voorzien dat toelaat om eigen-

schappen van entiteiten te bevragen tijdens de uitvoering, met als belangrijk voordeel dat interfaces niet hoeven te worden aangepast wanneer de set van eigenschappen van een entiteit wijzigt.

Als besluit kan worden gesteld dat door de ontwikkeling van het Tornado-raamwerk de mogelijkheid geschapen werd om waterkwaliteitsproblemen op te lossen die tot dusver te complex waren om binnen redelijke tijd te worden doorgerekend. Bovendien wordt verwacht dat dankzij de aard van het ontwerp van Tornado de mogelijkheid zal worden geschapen om gedurende niet nader bepaalde tijd de verwachte toenemende complexiteit te volgen. Om het raamwerk verder te completeren, is er echter nood aan bijkomend werk en de exploratie van bijkomende onderzoeksterreinen. In bepaalde gevallen werd dit bijkomend onderzoek reeds geïnitieerd.