Generating efficient executable models for complex virtual experimentation with the Tornado kernel

Filip H.A. Claeys*, Peter Fritzson and Peter A. Vanrolleghem*,*****

* BIOMATH, Ghent University, Coupure Links 653, B-9000 Gent, Belgium
(E-mail: *Filip.Claeys@biomath.ugent.be*)

** PELAB, Linköping University, SE-581 83 Linköping, Sweden (E-mail: *petfr@ida.liu.se*)

*** modelEAU, Pavillon Pouliot, Université Laval, G1K 7P4 Québec, QC, Canada
(E-mail: *Peter.Vanrolleghem@modelEAU.org*)

Abstract Virtual experimentation is a collective term that includes various model evaluation procedures such as simulation, optimization and scenario analysis. Given the complexity of the models used in these procedures, and the number of evaluations that is required to complete them, highly efficient model implementations are desired. Although water quality management is a domain in which complex virtual experimentation is often adopted, only relatively little attention has thus far been devoted to the automated generation of efficient executable models. This article reports on a number of promising results regarding executable model generation that were obtained in the scope of the Tornado kernel, using techniques such as equiv substitution and equation lifting.

Keywords Code generation; executable models; model compilers; virtual experimentation

Water Science & Technology **Vol 56 No 6 pp 65–73** © IWA Publishing 2007

Introduction

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 the River Water Quality Model No.1 (RWQM1) (Reichert *et al.*, 2001) and the Activated Sludge Model (ASM) series (Henze *et al.*, 2000). Water quality models typically consist of large sets of non-linear Ordinary Differential Equations (ODE). These equations are mostly 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 models such as the well known Benchmark Simulation Model No.1 (BSM1) (Copp *et al.*, 2002) consist of approximately 150 derived variables. Larger systems have up to 1,000 derived variables and over 10,000 (mostly coupled) parameters. On a typical workstation, a simulation run usually lasts minutes to hours.

Virtual experimentation with water quality models is a complex and computationally intensive process, which frequently requires 100s or 1,000s of simulation runs. A continuous need for software tools that offer extended functionality and improved performance therefore exists (Gujer, 2006). An example of a software tool that was recently developed is Tornado (Claeys *et al.*, 2006a). It 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. One factor that strongly influences the execution time of virtual experiments is the efficiency of the executable models that are being used. Clearly, for small problems highly efficient implementations can be hand-crafted. However, manual coding is not a tractable solution for larger problems, and is inappropriate in terms of portability and re-usability. Therefore, approaches in which executable model code is generated automatically on the basis of declarative

doi: 10.2166/wst.2007.581

model representations are favored (Muetzelfeldt 2004). Although such mechanisms have been in use for quite a number of years in water quality research (Vanhooren, 2003), only little attention has been devoted to the quality of the code generated from these declarative descriptions. This article therefore focuses on techniques such as equiv substitution and equation lifting, which have recently been implemented in the scope of the Tornado kernel. These techniques allow for an increase in efficiency and a reduction in size. The effects are illustrated on a number of WWTP plant layouts, including BSM1. Since executable model formats are designed for efficient execution rather than human-readability, care must be taken to detect and report run-time problems in an appropriate way. Two techniques that are helpful in this respect are the automated generation of bounds checking code, and code instrumentation.

The Tornado kernel

Tornado (Claeys *et al.* 2006a, 2006b, 2006c, 2006d) is an advanced kernel for modelling and virtual experimentation that was recently jointly developed by BIOMATH (Ghent University) and MOSTforWATER NV (Kortrijk, Belgium). Tornado introduces a low level of overhead during simulations; performance is therefore comparable to completely hand-crafted solutions. Several APIs are available (native C++, C, .NET, MEX, etc.), which allow for deployment of the kernel in a variety of software systems. The most prominent application that is being built on top of the Tornado kernel is the next generation of the WEST® (Vanhooren, 2003) modelling and simulation tool for WWTPs. The modelling language that has thus far been used in Tornado is MSL (Model Specification Language) (Vanhooren, 2003). However, support for Modelica (Fritzson 2004) has recently been introduced as well. Modelica is similar to MSL in the sense that it is high-level, declarative and object-oriented. As Tornado is merely a kernel (*i.e.* a set of class libraries) it does not come with a full-fledged (graphical of other) user interface. However, in order to facilitate testing, a suite of some 30 command-line tools is included. These tools are surprisingly versatile, and allow for a clear demonstration of the Tornado concepts. The work discussed in this article was realized using the Tornado command-line suite only.

Complex virtual experimentation

Tornado consists of strictly separated modelling and virtual experimentation environments. The experimentation environment allows for running so-called atomic and compound virtual experiments. The latter are hierarchically structured whereas the first cannot be further decomposed. Atomic experiment types that are available in Tornado are dynamic simulation and steady-state analysis. The most straightforward types of compound experiments are optimization, scenario analysis, Monte Carlo analysis (e.g. with Latin Hypercube Sampling) and sensitivity analysis. More convoluted types of compound experiments are also available, such as the computation of aggregated statistical data. Thanks to the object-oriented nature of Tornado, new virtual experiment types can easily be added.

Efficient executable models

One important aspect of Tornado is its ability to generate efficient executable code from models implemented in high-level modelling languages. Through these high-level modelling languages, complex models can be constructed using techniques such as inheritance and composition. Converting high-level models to executable code is called model compilation, and typically consists of two phases. During the first phase (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). Subsequently, the AST is manipulated in order to perform flattening of the model's inheritance and composition hierarchies. The resulting representation will consist of only one model with coalesced declaration and

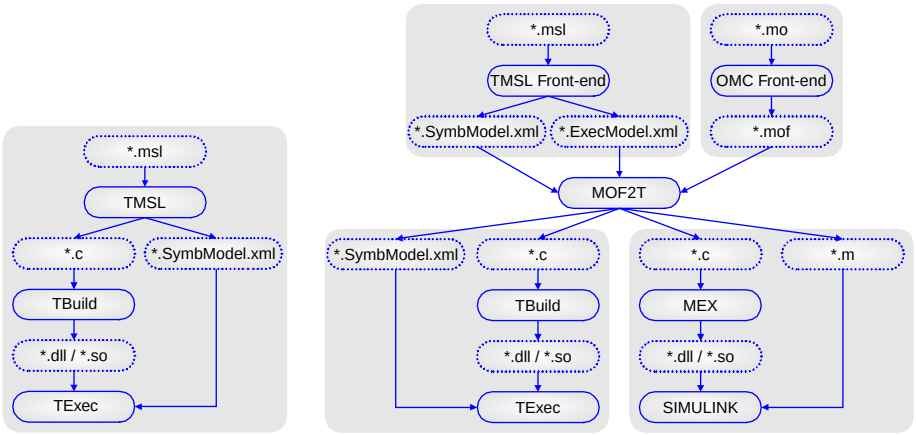

Figure 1 Code generation using the *tmsl* front-end and back-end (left); and using *tmsl*, *omc* and *mof2t* (right)

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. The second phase (back-end) of the model compilation process is aimed at the generation of optimized executable code from the flattened model. In Tornado, C was chosen as a target language, for reasons of efficiency and flexibility (binary C code can be linked to almost any other type of binary code). The generated C code is compiled by a regular C compiler and can be dynamically loaded into the Tornado virtual experiment executor.

Historically, the first high-level modelling language that was supported by Tornado is MSL. Figure 1 (left) shows that MSL input files are processed by the *tmsl* model compiler in order to generate executable C code. Next to C code, the compiler also generates a file containing a representation of the model's non-flattened, non-computational meta-information. Included in this information are human-readable names and attributes of sub-models, parameters and variables. The information is stored in XML format and is referred to as symbolic model info. The C code generated by *tmsl* is processed by a command-line tool named *tbuild*, in order to generate a dynamically loadable library. This library is fed into *texec*, i.e. Tornado's virtual experiment executor, along with a description of the experiment to be executed (in XML format, not shown on the figure).

Modelica is a high-level, declarative and equation-based object-oriented language that has thus far mainly been used for physical system modelling. It is supported by a steadily growing consortium (cf. Modelica Association (http://www.modelica.org)) and has a number of properties that make it equally suitable for biological and ecological system modelling. Modelica is a very complex language from a model compiler point of view. Building a compiler that translates Modelica models into a custom executable model format requires substantial resources. Fortunately, from the two model compiler phases described above, only the second phase depends on the desired executable model format; the first phase is generic. One can therefore re-use an already existing model compiler front-end, if so-desired. The OpenModelica project (http://www.ida.liu.se/ ~ pelab/modelica/OpenModelica.html) offers a Modelica compiler (*omc* - OpenModelica Compiler) that can be used as a front-end to one's own back-end. Since this approach allows for a significant reduction of the model compiler development time, it was adopted in the scope of Tornado.

The Modelica model compiler back-end that was developed for Tornado was named *mof2t* (Claeys *et al.*, 2006b). At this point it only supports a subset of Modelica. The topmost part of Figure 1 (right) shows that Modelica input files are processed by the *omc* front-end in order to generate flattened Modelica output, which is then further

processed by *mof2t*. Since *mof2t* has a number of features that are not present in the original *tmsl* back-end, joint use of the *tmsl* front-end and *mof2t* has been enabled. In this case, a representation of the *tmsl* AST is saved in XML format. This type of information is named executable model info and is passed to *mof2t*, along with the same type of symbolic model info as mentioned before. As can be seen from the lower part of Figure 1, *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 MATLAB SIMULINK S-function. This code 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 of this article will focus on the first type of code generation, although the principles that will be discussed are also applicable to the second type.

One issue that needs to be clarified before some of the special techniques that are implemented in *mof2t* can be discussed, is the nature of the information contained in the executable C code that is generated. Basically, this C code consists of data containers and event routines. Data containers are arrays (with standardized names) in which scalar data items (produced during flattening) are aligned. The following data item types are distinguished: Parameters, Independent Variables, Input Variables, Output Variables, Algebraic Variables and Derived Variables. As the number of data items of complex models can be very large, these arrays can easily hold 10,000 items or more. 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: ComputeInitial (computes parameters and/or initial conditions 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 timepoint), ComputeOutput (computes variables that do not contribute to the state of the system; is executed at each major integration timepoint), and ComputeFinal (computes variables for which only the final value is required; is only executed once at the end of each simulation). 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 2 shows the execution sequence of these routines.

Equiv substitution

As a result of the coupling of sub-models, flattened models will typically contain many equivs, i.e. equations of type $y = x$, where y is an input variable and x is an output variable. Although these equations are trivial, their sheer number can lead to substantial performance degradation, since they are to be computed at each integration timepoint. Through symbolic substitution of y by x in all equations, the equation $y = x$ can be dropped, and superfluous computations can be avoided. The performance gain that can be accomplished by equiv substitution is determined by two factors: the number of equivs versus the total number of equations, and the complexity of the non-equiv equations. In case the number of equivs is high, or the complexity of non-equiv equations is low, equiv

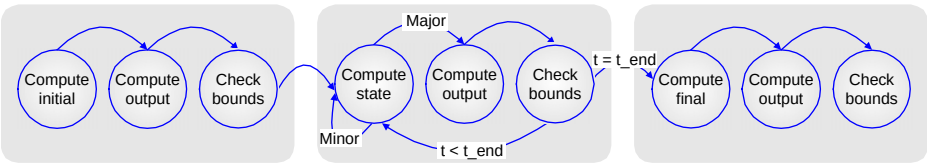

Figure 2 Call graph of executable model event routines

substitution has been shown to lead to a performance increase of 5 to 30%. In case there are only few equiv equations, or the non-equiv equations are of high complexity (e.g. containing power operations), the effect will only be marginal. In fact, experiments have shown that equiv substitution may even lead to slight performance degradation. The reason for this is that it may 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 to the size of the code they are able to compile, equiv substitution may help to resolve certain problems.

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. However, it is not sufficient. First of all, neither MSL nor Modelica allow for equations to be labeled as output equations. Reason for this is that manual discovery of output equations is non-trivial and error-prone. 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. moving equations from one section to another) that can be implemented. By inspecting the right-hand sides of equations in the state section, the equation's variability 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 be 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. This procedure is known as lifting of initial equations. The remaining equations vary during a simulation and are therefore of variability 2. 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 nodes that represent differential equations. 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. This procedure is known as lifting of output equations. In practice, the number of equations that can be additionally identified as initial equations by the model compiler is often limited. Lifting of initial equations will therefore usually not yield significant performance gains. 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 timepoints (instead of at every minor timepoint if they were to remain in the state section) yields significant performance improvements, e.g. 20%.

Bounds checking

As a result of the various manipulations performed by a model compiler (flattening, lifting, etc.), 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 user-friendly warnings or error messages. One way to improve model safety (e.g. in case of divergence due to numerical problems such as inappropriate integrator tolerance set-

tings) is to check each variable against its lowerbound and upperbound during the course of a simulation (both MSL and Modelica allow for 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 specific bounds checking code for the model at hand. This approach allows for several optimizations, since lowerbounds set to $-\infty$ and upperbounds set to ∞ do not need to be checked, hence no code needs to be generated. The event routine that performs bounds checking was named CheckBounds and is called at every major integration timepoint, as can be seen from Figure 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 bounds violation state, in order to avoid the same message to be issued multiple times. 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 simulator.

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 $x\hat{}y$ is not defined when x is negative and y is not an integral value. 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 timepoint. What one would like to see however, is a human-readable representation of the expression that has triggered the run-time error. This can be accomplished by performing code instrumentation, i.e. replacing potentially hazardous operations by macros (preprocessor defines). In these macros, 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 1 gives an overview of a number of operations that have limited domains of validity for their arguments. The table also lists the macros that are generated by the model compiler as a replacement. Finally, the table shows the implementation of the

Table 1 Code Instrumentations

Original	Replacement	Implementation
x/y	_DIV_(x, y, StringID)	y == 0 ? DivisionByZero(StringID): x/y
acos(x)	_ACOS_(x, StringID)	(x < −1) || (x > 1) ? DomainError(StringID): acos(x)
asin(x)	_ASIN_(x, StringID)	(x < −1) || (x > 1) ? DomainError(StringID): asin(x)
log(x)	_LOG_(x, StringID)	x <= 0 ? DomainError(StringID): log(x)
log10(x)	_LOG10_(x, StringID)	x <= 0 ? DomainError(StringID): log10(x)
pow(x)	_POW_(x, y, StringID)	(x < 0) && ((y - (int)y) != 0) ? DomainError(StringID): pow(x, y)
sqrt(x)	_SQRT_(x, StringID)	x < 0 ? DomainError(StringID): sqrt(x)

various macros. 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 (IDs) are passed to the macros (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.

Results

Table 2 contains various results that were obtained by applying the above-mentioned techniques to a number of cases. In the ASU case, a WWTP with alternating aeration is studied. The activated sludge basin is modelled with one completely mixed reactor. BSM1_OL (OL stands for "open loop"; CL stands for "closed loop"), i.e. the Benchmark Simulation Model No.1, is a standardized simulation environment defining a plant layout, a simulation model, influent loads, test procedures and evaluation criteria. In the BSM1_CL 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. The Orbal plant achieves biological nutrient removal and is 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 denitrification in a single reactor, offering reduced costs. The Galindo_OL WWTP is designed for C and N removal and is divided into six identical lines, which have two alternative configurations: Regeneration-Denitrification-Nitrification (RDN) and Denitrification-Regeneration-Denitrification-Nitrification (DRDN). Finally, the Galindo_CL WWTP is modelled with oxygen control by means of a PI controller.

All models were implemented in MSL and subsequently processed by the *tmsl* front-end and *mof2t*. Tests were run on the Windows platform using a reference machine (Intel Pentium M 1,600 MHz; 512 Mb RAM). As a C compiler, Microsoft Visual C++7.1 was used. 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 has an effect on the speed of simulation. Items 8–11 provide insight in the complexity of the model. Items 12–14 and 15–17 give the number of initial, state and output equations, respectively at the beginning and the end of *mof2t*'s processing. In general, only few additional equations can be moved from the state to the initial section. However, the number of output equations is mostly substantial. In each case, the following relations hold: $\#Final_{start} = \#Final_{end}$; $\#Output_{start} = 0$; $\#Initial_{start} + \#State_{start} - \#Equiv = \#Initial_{end} + \#State_{end} + \#Output_{end}$. Items 18–19 provide some information on the size of the executable model code (after C compilation, without bounds checking nor code instrumentation). Items 20–22 respectively give the simulation time of non-optimized code (with equiv substitution and lifting disabled), the simulation time of optimized code (with equiv substitution and lifting enabled), and the speedup of the optimized code versus the non-optimized code. Items 23–25 respectively give the simulation time of optimized code with bounds checking enabled, the simulation time of optimized code with bounds checking *and* code instrumented 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 *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 26–27 provide a comparison with the simulation speed of the same

Table 2 Results of the application of *mof2t* 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-4	1e-6	1e-6	1e-6	1e-5	1e-5
3	Simulation time horizon	7d	28d	28d	25d	400d	400d
4	Output file communication interval	0.01d	15 min	15 min	0.01d	15 min	15 min
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 equivs	172	219	253	623	489	614
11	Number of differential equations	30	178	110	250	270	275
12	Number of initial equations at start	148	170	175	2,433	1,410	1,405
13	Number of state equations at start	722	1,289	1,331	3,551	3,600	3,953
14	Number of output equations at start	0	0	0	0	0	0
15	Number of initial equations at end	149	171	176	2,434	1,553	1,408
16	Number of state equations at end	418	929	943	2,137	2,630	2,998
17	Number of output equations at end	131	140	134	790	338	338
18	Size of executable model DLL (unsafe optimized code)	36 kB	72 kB	92 kB	276 kB	248 kB	252 kB
19	Size of symbolic model XML (unsafe optimized code)	323 kB	551 kB	560 kB	2,604 kB	2,315 kB	2,402 kB
20	Simulation time of non-optimized code	5 s	15 s	48 s	110 s	315 s	465%
21	Simulation time of optimized code	4.5 s	14 s	43 s	86 s	220 s	426%
22	Speedup of unsafe optimized code vs. unsafe non-optimized code	+10%	+7%	+10%	+22%	+30%	+8%
23	Simulation time of optimized code with bounds checking	5 s	14 s	45 s	94 s	235 s	454 s
24	Simulation time of optimized code with bounds checking and instrumentation	5 s	16 s	48 s	106 s	296 s	518 s
25	Speedup of safe optimized code vs. unsafe non-optimized code	−0%	−14%	−7%	−13%	−26%	−14%
26	Simulation time of partially safe non-optimized WEST®-3 code	8.5 s	40 s	112 s	687 s	1,531 s	2,261 s
27	Speedup of safe optimized Tornado code vs. WEST®-3 code	+41%	+60%	+57%	+85%	+81%	+77%

cases implemented in WEST®-3, a simulation tool that is currently commercially available. The WEST®-3 model compiler does not offer 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 non-optimized 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.

Conclusions and future work

As a result of Tornado's low overhead level and the techniques that were recently implemented in the *mof2t* model compiler back-end, complex models can now be simulated several times faster *and* with a higher safety level than through simulation tools commercially available. Moreover, since *mof2t* can be used in unison with *tmsl* and *omc*, the features offered by *mof2t* can be applied to MSL as well as to (a subset of) Modelica models. Future work will focus on other types of symbolic model manipulations, such as symbolic differentiation, reduction of expressions to a canonical form, and static verification of units. As these techniques are intended to be implemented in *mof2t*, they will be applicable to MSL as well as Modelica models.

Acknowledgements

Peter A. Vanrolleghem is Canada Research Chair in Water Quality Modelling.

References

Claeys, F., De Pauw, D., Benedetti, L., Nopens, I. and Vanrolleghem, P. (2006a). Tornado: A versatile efficient modelling and virtual experimentation kernel for water quality systems. *Proc. 2006 iEMSs Conference*. Burlington, VT.

Claeys, F., Fritzson, P. and Vanrolleghem, P.A. (2006b). Using Modelica models for complex virtual experimentation with the Tornado kernel. *Proc. 2006 Modelica Conference*. Vienna, Austria.

Claeys, F., Vanrolleghem, P.A. and Fritzson, P. (2006c). Boosting the efficiency of compound virtual experiments through a priori exploration of the solver setting space. *Proc. 2006 EMSS Conference*. Barcelona, Spain.

Claeys, F., Vanrolleghem, P.A. and Fritzson, P. (2006d). A generalized framework for abstraction and dynamic loading of numerical solvers. *Proc. 2006 EMSS Conference.* Barcelona, Sspain.

Copp, J.B. (ed.) (2002). *The COST simulation benchmark – description and simulator manual* ISBN 92-894-1658-0, Office for Official Publications of the European Communities, Luxembourg.

Fritzson, P. (2004). *Principles of Object-Oriented Modeling and Simulation with Modelica.* Wiley-IEEE Press. *ISBN*, 0–471-47163-1.

Gujer, W. (2006). Activated sludge modelling: past, present and future. *Water, Science and Technology.*, **53**(3), 111–119.

Henze, M., Gujer, W., Mino, T. and van Loosdrecht, M. (2000) *Activated Sludge Models ASM1, ASM2, ASM2D and ASM3, (Scientific and Technical Report No. 9)*, IWA Publishing, London.

Muetzelfeldt, R. (2004). Position paper on declarative modelling in ecological and environmental research. In: *European Commission*, Directorate-General for Research, EUR(20918).

Reichert, P., Borchardt, D., Henze, M., Rarch, W., Shanahar, P., Somlyody, L. and Vanrolleghem, P.A. (2001). *River Water Quality Model No. 1. (Scientific and Technical Report No. 12)*, IWA Publishing, London.

Vanhooren, H., Meirlaen, J., Amerlinck, Y., Claeys, F., Vangheluwe, H. and Vanrolleghem, P.A. (2003). WEST: modelling biological wastewater treatment. *Journal of Hydroinformatics.*, **5**(1), 27–50.