A GENERALIZED FRAMEWORK FOR ABSTRACTION AND DYNAMIC LOADING OF NUMERICAL SOLVERS

Filip H.A. Claeys
Department of Applied Mathematics,
Biometrics and Process Control (BIOMATH)
Ghent University
Coupure Links 653
B-9000 Gent
Belgium
E-mail: fc@biomath.ugent.be

Peter A. Vanrolleghem
model*EAU*
Département de génie civil
Université Laval
Pavillon Pouliot
Québec, G1K 7P4
QC, Canada
E-mail: peter@modelEAU.org

Peter Fritzson
Programming Environments
Laboratory (PELAB)
Linköping University
Campus Valla
SE-581 83 Linköping
Sweden
E-mail: petfr@ida.liu.se

KEYWORDS

Software development, modelling & simulation, numerical solvers, abstraction, dynamic loading

ABSTRACT

Scientific software often relies on numerical solvers for tasks such as integration of differential equations, optimization and finding roots of linear or non-linear systems. Typically, the availability of only one solver for a certain task is not sufficient in generic software systems, since each solver usually only has a limited area of application. However, the inclusion of multiple solvers into complex industrial quality software is often cumbersome. One issue is the fact that solver codes are often implemented in different programming languages, other issues are related to a lack of standardization of the methods provided by the codes, fixed I/O routines and disregard of thread-safety concerns. This article describes how some of these issues were handled in the scope of Tornado, an advanced kernel for modelling and virtual experimentation (*i.e.*, any evaluation of a model). Particularly important is that in view of the maintainability and extensibility requirements that are often imposed by complex scientific software, there is a need for abstraction and dynamic loadability of solvers.

INTRODUCTION

In many areas, including modelling & 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 has 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 (Ordinary Differential Equation), DAE's (Differential-Algebraic Equation) and PDE's (Partial Differential Equation). 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 (*e.g.*, through machine-learning techniques based on the analysis of previously collected data) (Claeys et al., 2006b). The problem of choosing amongst solver alternatives however is not the focus of this article. Instead, this article 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.

Throughout this article, the Tornado kernel for modelling and virtual experimentation is used as a case. Therefore, the next section constitutes an introduction to Tornado. Afterwards, a short section is devoted to the availability of solvers. The two subsequent sections discuss the issues that can be encountered when integrating numerical solvers into quality software, and the solutions that have been provided in the scope of Tornado. In each case some practical examples are given.

THE TORNADO KERNEL

Tornado (Claeys et al., 2006a) is an advanced kernel for modelling and virtual experimentation (*i.e.*, any evaluation of a model such as simulation, optimization, scenario analysis, . . .). Although the kernel is generic in nature, it is mostly adopted in the water quality domain (*i.e.*, the study of biological and/or chemical water-related processes in rivers, sewers and wastewater treatment plants). Water quality models typically consist of large sets of non-linear Ordinary Differential Equations (ODE) and/or Differential-Algebraic Equations (DAE). These equations are mostly well-behaved, although discontinuities occur regularly. The complexity of water quality models is therefore not in the nature of the equations, but in the sheer number.

The Tornado kernel attempts to offer a compromise between the computational efficiency of custom hard-coded (typically FORTRAN or C) model implementations and the flexibility of less computationally efficient generic tools such as MATLAB. In Tornado, hierarchical models are specified in high-level, declarative, object-oriented modelling languages such as MSL (Vanhooren et al., 2003) and Modelica (Fritzson, 2004). From these high-level specifications, efficient executable code is generated by a model compiler. Using the executable models generated by the model compiler, Tornado allows for running a variety of so-called *virtual experiments*. Virtual experiments are the virtual-world counterpart of real-world experiments, similar to the way models relate to real-world systems (Kops et al., 1999).

For each type of virtual experiments that is implemented in Tornado, one or more numerical solver types are needed. Overall, Tornado includes solvers for integration of ODE's and DAE's, optimization of non-linear systems, root finding of linear and non-linear systems, random number generation, sampling from standard statistical distributions, and Latin Hypercube Sampling (LHS).

Tornado is portable across platforms and was designed according to the three-tier principle (Voth et al., 1998). Implementation was done in C++, using advanced language features. Most persistent representations of information types are XML-based. The grammar of these representations is expressed in XSD (XML Schema Definition) format and mimics very closely the internal representation of the respective types of information.

Several applications (graphical and other) can be built on top of Tornado. One example includes the next generation of the WEST® (Vanhooren et al., 2003) modelling and simulation tool for wastewater treatment plants.

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 1 lists some of the most popular on-line resources.

Table 1: Popular numerical solver repositories

Repository	Description	URL
Netlib	Network Library	http://www.netlib.org
GAMS	Guide to Available Math. s/w	http://gams.nist.gov
ACM TOMS	Transactions on Math. s/w	http://www.acm.org/toms
NAG	Numerical Algorithms Group	http://www.nag.co.uk

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. One will therefore 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* 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, recompilation 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, optimiza-

tion, ...) 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 integrators, these settings for instance include stepsizes and tolerances. Solvers that perform similar tasks may not only have a different number of settings, 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 - *stdout*). Sometimes a distinction is made between standard output for informational messages and warnings and standard error (*stderr*) 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 (Graphical User Interface).

- **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 MDI (Multiple Document Interface) GUI's and multi-threaded computation servers.

- **Unappropriate 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 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.

THE TORNADO SOLVER FRAMEWORK

Since Tornado has to deal with various solver types (in order to support the respective types of virtual experiments), and for each type of solver, multiple solver implementations need to be available (in order to cover as many areas of application), 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 application. 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. The remainder of this section gives a more structured description of the implementation of the Tornado solver framework.

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. All classes are based on one or more abstract interfaces (Stroustrup, 1997). In order to distinguish implementation classes from abstract interfaces, the names of implementation classes are prefixed with a capital "C", while abstract interfaces are prefixed with a capital "I". Figure 1 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 except 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.

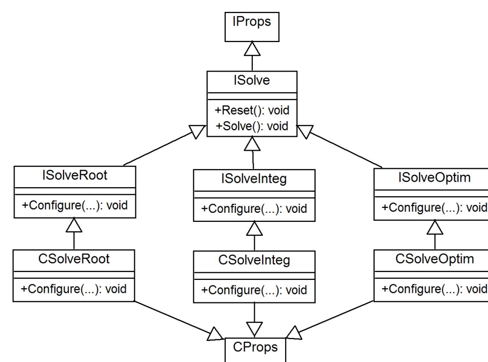

Figure 1: Solver inheritance hierarchy (excerpt)

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 pointers to a num-

ber of entities that are required for the operation of the solver. As an example, Table 2 lists the arguments of this method for the integrator solver base class. The purpose of the callback arguments will be explained further in this section.

Table 2: Arguments of the SolveInteg::Configure() method

Argument	Description
CallbackMessage	Called when messages are generated
CallbackStop	Called to discover if processing is to be stopped
CallbackLicense	Called to discover if 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 1 also shows that abstract interfaces and base classes are derived from a class named *Props*, which stands for *properties*. This class implements a mechanism for specification and run-time querying of solver settings and will also be discussed later in this section.

All solvers 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 apply the appropriate casting.

As mentioned before, Tornado will load solver plug-ins at startup. The list of plug-ins 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 6 solvers are to be loaded (3 integrators, 2 optimizers and 2 root finders). The names mentioned are plug-in files names, which could be absolute or relative path names (possibly containing environment variables). At the moment a total of 36 solver plug-ins are available for Tornado.

```
<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 step-sizes for integrators) are not manipulated through specialized method calls, for 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. Each property is a structure that contains the members listed in Table 3.

Table 3: Property members

Name	Type	Description
Name	wstring	Property name
DataType	TDataType	Boolean, Integer, Real or String
AccessType	TAccessType	Read-only or Read/Write access
Description	wstring	Textual description
Default	CValue	Default value
Range	pair<CValue, CValue>	Lowerbound and Upperbound
Value	CValue	Actual value

In reality, the situation is somewhat more complicated than this, since actual values and meta-information (all other property members except for the actual value) are stored separately in Tornado. Figure 2 contains an informal Entity Relationship (ER) diagram that gives some more detail. It shows that the properties structure (*CProps*) that is associated with a solver consists of two elements: a map of values (*ValueMap*) and a list of meta-information structures (*PropsInfo*). There is a 1-to-1 mapping between values and meta-information structures based on the uniqueness of property names.

Figure 3 contains an informal ER diagram for the *CValue*

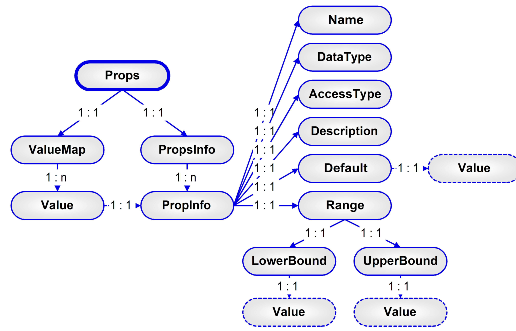

Figure 2: CProps ER diagram

structure. It shows that it contains a specification of the data type of the value, and a union containing an overlay of instances of the various possible data types (Boolean, Integer, Real, String). From this also follows that the inclusion of a data type specification in *CValue* is actually redundant, as the data type is also stored in the meta-information that is associated with a property.

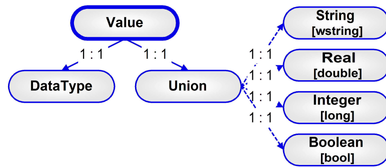

Figure 3: CValue ER diagram

One will notice that the meta-information structure contains a range of validity expressed as a lowerbound and upperbound. Lowerbounds and upperbounds will only be used in their true sense when the data type is either Integer or Real. However, in case the data type is set to String, a special meaning will be attached to the lowerbound. By assigning it a string composed of sub-strings separated by semicolons, it can be enforced that the value of the property can only be set to one of the sub-strings. No other values will hence be accepted.

Table 4 contains an overview of the properties that are supported by the CVODE integrator plug-in for Tornado, which originates from the SUNDIALS suite (http://www.llnl.gov/CASC/sundials). 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.

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. In this case, 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

Table 4: Properties of CVODE integrator

Name	Type	Access	Default	Range
ModelTypes	S	R	ODE	-
StepSizeType	S	R	Variable	-
IsStiffSolver	B	R	true	-
MaxNoSteps	I	R/W	0	0 - ∞
AbsoluteTolerance	R	R/W	1e-7	0 - ∞
RelativeTolerance	R	R/W	1e-7	0 - ∞
LinearMultistepMethod	S	R/W	Adams	Adams;BDF
IterationMethod	S	R/W	Functional	Functional; Newton
LinearSolver	S	R/W	Dense	Dense;Band; Diag;SPGMR
CVBandUpperBandwidth	I	R/W	0	0 - ∞
CVBandLowerBandwidth	I	R/W	0	0 - ∞
CVSPGMRGSType	S	R/W	ModifiedGS	ModifiedGS; ClassicalGS

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 generates 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. Callback interfaces are usually quite simple in Tornado. This is illustrated by the following code fragment that shows the declaration of the message callback interface.

```
class ICallbackMessage
{
  public:

    virtual void
    Info(const std::wstring& Message) = 0;

    virtual void
    Warning(const std::wstring& Message) = 0;

    virtual void
    Error(const std::wstring& Message) = 0;
};
```

As is shown in Table 2, integrator solvers can make use of 4 callbacks in Tornado. Next to the message callback, there is also a callback for testing if 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* tool (http://www.netlib.org/f2c). 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.

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.

CONCLUSION

This article touches on the issues that are associated with the integration of multiple numerical solver codes into quality software. Through a number of examples and implementation details it is shown how these problems have been overcome in the Tornado kernel for modelling and virtual experimentation. The approach followed in Tornado is based on the definition of abstract interfaces and dynamic loading of solver plug-ins, containing meta-information that can be queried at run-time.

REFERENCES

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, 2006a, *Accepted*.

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, 2006b, *Accepted*.

P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, February 2004. ISBN 0-471-47163-1.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 2003.

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.

W.H. Press, S.A. Teukolsky, W.T. Vettering, and B.P. Flannery. *Numerical Recipes in C: the art of scientific computing*. Cambridge University Press, New York, NY, 2nd edition, 1992.

B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

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.

G.R. Voth, C. Kindel, and J. Fujioka. Distributed application development for three-tier architectures: Microsoft on Windows DNA. *IEEE Internet Computing*, 2(2):41–45, 1998.

AUTHOR BIOGRAPHIES

FILIP H.A. CLAEYS was born in Ghent, Belgium. He received a MSc in Computer Science from Ghent University and a Master's in Artificial Intelligence from K.U.Leuven. He currently works as a senior software engineer for HEMMIS N.V. and leads a research group in the field of modelling and simulation software tools at Ghent University.

PETER A. VANROLLEGHEM, bio-engineer, PhD, heads the model*EAU* research team at Université Laval (Québec) and has ample experience with modelling, monitoring and control of wastewater treatment systems. He has over 175 peer-reviewed papers and is very active within the International Water Association.

PETER FRITZSON is head of the Programming Environment Laboratory at Linköping University, Sweden. He holds the positions of research manager at MathCore Engineering AB, chairman of the Scandinavian Simulation Society and vice-chairman of the Modelica Association. He has published 10 books and over 100 scientific papers.