THE OMNET++ DISCRETE EVENT SIMULATION SYSTEM

András Varga
Department of Telecommunications
Budapest University of Technology and Economics
Pázmány Péter sétány 1/d.
1117 Budapest, Hungary
E-mail: andras@whale.hit.bme.hu

KEYWORDS
discrete simulation, performance analysis, computer systems, telecommunications, hierarchical

ABSTRACT

The paper introduces OMNeT++, a C++-based discrete event simulation package primarily targeted at simulating computer networks and other distributed systems. OMNeT++ is fully programmable and modular, and it was designed from the ground up to support modeling very large networks built from reusable model components. Large emphasis was placed also on easy traceability and debuggability of simulation models: one can execute the simulation under a powerful graphical user interface, which makes the internals of a simulation model fully visible to the person running the simulation: it displays the network graphics, animates the message flow and lets the user peek into objects and variables within the model. These features make OMNeT++ a good candidate for both research and educational purposes. The OMNeT++ simulation engine can be easily embedded into larger applications. OMNeT++ is open-source, free for non-profit use, and it has a fairly large user community.

INTRODUCTION

OMNeT++ is a C++-based discrete event simulator for modeling communication networks, multiprocessors and other distributed or parallel systems. OMNeT++ is open-source, and it can be used either under the GNU General Public License or under its own license that also makes the software free for non-profit use. The motivation of developing OMNeT++ was to produce a powerful open-source discrete event simulation tool that can be used by academic, educational or research-oriented commercial institutions for the simulation of computer networks and distributed or parallel systems. OMNeT++ tries to fill the gap between open-source, research-oriented simulation software such as ns (Bajaj et al. 2000) and expensive commercial alternatives like OPNET (OPNET Technologies, Inc.). A later section of this paper presents a comparison with other simulation packages. OMNeT++ is available on Unix systems and on Windows, using the Cygwin or the Microsoft Visual C++ compiler. It should also be possible to port OMNeT++ to other systems with minor effort.

OMNeT++ has been available to the public since September 1997, and by now it has a fairly large user community and a mailing list. Besides anonymous downloads, there have been registered downloads from over 40 universities worldwide, and the indicated application areas ranged from mobile/wireless to ATM and optical networks simulation, and from hardware simulations to queuing systems. There are a number of OMNeT++-related projects, such as the development of a complete TCP/IP model suite at University Karlsruhe (Kaage et al. 2001; Wehrle et al. 2001), the Remote OMNeT++ project (Erdei et al. 2001; Wagner and Erdei 2001) for managing remote simulations on a flock of workstations, and research on parallel execution, using the Statistical Synchronization Method (Lencse 1997; Lencse 1998).

THE DESIGN OF OMNeT++

OMNeT++ was designed from the beginning to support network simulation in the large. This objective lead to the following main design requirements:

- To enable large-scale simulation, simulation models need to be hierarchical, and should be built from reusable components as much as possible.
- Simulation programs are infamous for long debugging periods. Thus, the simulation software should place large emphasis on easy traceability and debuggability of simulation models to reduce debugging time. (The same feature set is also useful for educational use of the software.)
- The simulation software itself should be modular, customizable and should allow embedding simulation models into larger applications such as network planning software. (Embedding brings additional requirements about the memory management, restartability, etc. of the simulation).

- Data interfaces should be open: it should be possible to generate and process input and output files with commonly available software tools.

The following sections go through the most important aspects of OMNeT++, highlighting the design decisions that helped achieve the above main goals.

Model Structure

An OMNeT++ model consists of modules that communicate with message passing. The active modules are termed *simple modules*; they are written in C++, using the simulation class library. Simple modules can be grouped into *compound modules* and so forth; the number of hierarchy levels is not limited. The concept of simple and compound modules is similar to DEVS (Zeigler 1990; Chow and Zeigler 1994) atomic and coupled models. Messages can be sent either via connections that span between modules, or directly to their destination modules.

Both simple and compound modules are instances of *module types*. While describing the model, the user defines module types; instances of these module types serve as components for more complex module types. Finally, the user creates the system module as an instance of a previously defined module type. When a module type is used as a building block, there is no distinction whether it is a simple or a compound module. This allows the user to transparently split a simple module into several simple modules within a compound module, or do the opposite, re-implement the functionality of a compound module in one simple module, without affecting existing users of the module type.

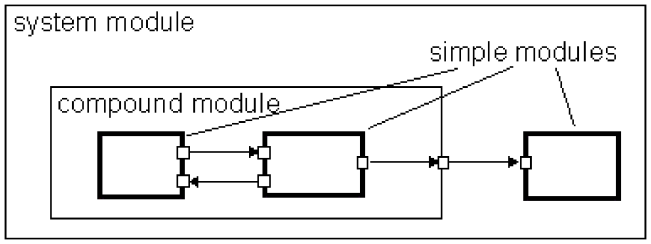

Fig. 1. Model Structure in OMNeT++. Boxes represent simple modules (thick border), and compound modules (thin border). Arrows connecting small boxes represent connections and gates.

Modules communicate with *messages* which – in addition to usual attributes such as timestamp – may contain arbitrary data. Simple modules typically send messages via *gates*, but it is also possible to send them directly to their destination modules. Gates are the input and output interfaces of modules: messages are sent out through output gates and arrive through input gates. An input and an output gate can be linked with a *connection*. Connections are created within a single level of module hierarchy: within a compound module, corresponding gates of two submodules, or a gate of one submodule and a gate of the compound module can be connected. Due to the hierarchical structure of the model, messages typically travel through a chain of connections, to start and arrive in simple modules. Compound modules act as 'cardboard boxes' in the model, transparently relaying messages between their inside and the outside world. Connections can be assigned properties such as propagation delay, data rate and bit error rate. One can also define connection types with specific properties (termed *channels*) and reuse them in several places.

Modules can have *parameters*. Parameters are mainly used to pass configuration data to simple modules, and to help define model topology. Parameters may take string, numeric or pointer values. Because parameters are represented as objects in the program, parameters – in addition to holding constants – may transparently act as sources of random numbers with the actual distributions provided with the model configuration, they may interactively prompt the user for the value, and they might also hold expressions referencing other parameters. Compound modules may pass parameters or expressions of parameters to their submodules. Parameters can be passed by value or by reference. Parameters taken by reference may be used to propagate global model parameter changes during simulation execution – this technique might be very useful in certain simulation scenarios such as parameter optimization.

The Design of the NED Language

The user defines the structure of the model (the modules and their interconnection) in OMNeT++'s topology description language, *NED*. Typical ingredients of a NED description are simple module declarations, compound module definitions and network definitions. Simple module declarations describe the interface of the module: gates and parameters. Compound module definitions consist of the declaration of the module's external interface (gates and parameters), and the definition of submodules and their interconnection. A network definition basically defines a model as an instance of a module type.

NED and the OMNeT++ model structure were designed to allow building models "in the large": they promote reusable model components via module types and unlimited compound module hierarchy levels. NED also supports partitioning large NED files into several smaller ones via file inclusion.

The OMNeT++ package includes a graphical editor which uses NED as its native file format; moreover, the editor can work with arbitrary, even hand-written NED code. The editor is a fully two-way tool, i.e. the user can edit the network topology either graphically or in NED source view, and switch between the two views at any time. This is made possible by design decisions about the NED language itself. First, NED is a declarative language, and as such, it does not use an imperative programming language for defining the internal structure of a compound module. Allowing arbitrary programming constructs would make it practically infeasible to write two-way graphical

editors which could work directly with both generated and hand-made NED files. (Generally, the editor would need AI capability to understand the code.)

Most graphical editors only allow the creation of fixed topologies. However, NED contains declarative constructs (resembling loops and conditionals in imperative languages), which enable parameterizing topologies: it is possible to create common regular topologies such as ring, grid, star, tree, hypercube, or random interconnection whose parameters (size, etc.) are passed in numeric-valued parameters. The potential of parameterized topologies and associated design patterns have been investigated in (Varga and Pongor 1997) and (Varga 1998). With parameterized topologies, NED holds an advantage in many simulation scenarios both over OPNET where only fixed model topologies can be designed, and over ns where building model topology is programmed in Tcl and often intermixed with simulation logic, so it is generally impossible to write graphical editors which could work with existing, hand-written code.

The NED language can be mapped one-to-one to XML; the graphical editor is capable of exporting and importing XML files. As XML is rapidly becoming the preferred way of exchanging structured information between applications, the existence of an XML binding for NED means greater opportunities for interfacing OMNeT++ with other systems. As an example, a network topology stored in an SQL database by a network management program can be imported into OMNeT++ in two steps. First, topology data are extracted from the database in XML format, and second, the resulting XML is transformed into NED XML by an XML style sheet transformation (XSLT). There are commonly available XML tools for both steps.

Model Libraries

While OMNeT++ itself does not contain a standard module library, it was designed with the expectation in mind that libraries of reusable modules will come to existence as soon as the software gets more widely deployed. Such libraries would contain network protocol models, application and traffic source models, etc.

As of January 2001, the following detailed protocol models are available for OMNeT++: TCP (Kaage et al. 2001), IP with QoS services (Wehrle et al. 2001), SCSI (also from Karlsruhe), FDDI (as part of the package). Simplified models for Ethernet, Token Ring, GSM, and file system simulation are also available. Further protocol models such as Hyperlan 2 are under development.

Simple Module Programming Model

Simple modules are the active elements in a model. They are atomic elements in the module hierarchy: they cannot be divided any further. Simple modules are programmed in C++, using the OMNeT++ simulation class library. The simulation kernel does not distinguish between messages and events – events are also represented as messages.

Simple modules are programmed using the process-interaction method. The user implements the functionality of a simple module by subclassing the `cSimpleModule` class. Functionality is added via one of two alternative programming models: *(1) coroutine-based*, and *(2) event-processing function*. When using *coroutine-based programming*, the module code runs in its own (non-preemptively scheduled) thread, which receives control from the simulation kernel each time the module receives an event (=message). The function containing the coroutine code will typically never return: usually it contains an infinite loop with *send* and *receive* calls.

When using *event-processing function*, the simulation kernel simply calls the given function of the module object with the message as argument – the function has to return immediately after processing the message. An important difference between the *coroutine-based* and *event-processing function* programming models is that with the former, every simple module needs an own CPU stack, which means larger memory requirements for the simulation program. This is of interest when the model contains a large number of modules (over a few ten thousands).

It is possible to write code which executes on *module initialization* and *finalization:* the latter takes place on successful simulation termination, and finalization code is mostly used to save scalar results into a file. OMNeT++ also supports *multi-stage initialization:* situations where model initialization needs to be done in several "waves". Multi-stage initialization support is missing from most simulation packages (including OPNET and ns), and it is usually emulated with broadcast events scheduled at zero simulation time, which is a less clean solution.

Message *sending and receiving* are the most frequent tasks in simple modules. Messages can be sent either via output gates, or directly to another module. Modules receive messages either via one of the several variations of the *receive* call (when using coroutine-based programming), or messages are delivered to the module in an invocation from the simulation kernel (when using the event-processing function).

It is possible to *modify the topology* of the network dynamically: one can create and delete modules and rearrange connections while the simulation is executing. Even compound modules with parameterized internal topology can be created on the fly.

Design of the Simulation Library

The OMNeT++ provides a rich object library for simple module implementers. There are several distinguishing factors between this library and other general-purpose or simulation libraries. The OMNeT++ class library provides reflection functionality which makes it possible to implement high-level debugging and

tracing capability, as well as automatic animation on top of it (as exemplified by the *Tkenv* user interface, see later). Memory leaks, pointer aliasing and other memory allocation problems are common in C++ programs not written by specialists; OMNeT++ alleviates this problem by tracking object ownership, doing ownership-based automatic deallocations and detecting bugs caused by aliased pointers and misuse of shared objects. The requirements for ease of use, modularity, open data interfaces and support of embedding also heavily influenced the design of the class library.

The consequential use of object-oriented techniques makes the simulation kernel very compact and slim: it is not bloated with unnecessary functionality and therefore facilitates a comprehensive understanding of its internals. This is an important argument in promoting OMNeT++ especially, but only, for educational use.

Contents of the Simulation Library

This section provides a very brief catalog of the classes in the OMNeT++ simulation class library. The classes were designed to cover most of the common simulation tasks.

OMNeT++ has the ability to generate *random numbers* from several independent streams. The common distributions are supported, and it is possible to add new distributions programmed by the user. It is also possible to load user distributions defined by histograms.

The class library offers *queues* and various other *container classes*. Queues can also operate as priority queues.

Messages are objects which may hold arbitrary data structures and other objects (through aggregation or inheritance), and can also embed other messages.

OMNeT++ supports *routing* traffic in the network. This feature provides the ability to explore actual network topology, extract it into a graph data structure, then navigate the graph or apply algorithms such as Dijkstra to find shortest paths.

There are several *statistical classes*, from simple ones which collect the mean and the standard deviation of the samples to a number of distribution estimation classes. The latter include three highly configurable *histogram* classes and the implementations of the P^2 (Jain and Chlamtac 1985) and the *k-split* (Varga and Fakhamzadeh 1997) algorithms. It is also supported to write *time series* result data into an output file during simulation execution, and there are tools for post-processing the results.

Internal Architecture

OMNeT++ simulation programs possess a modular structure. The logical architecture is shown Fig 2.

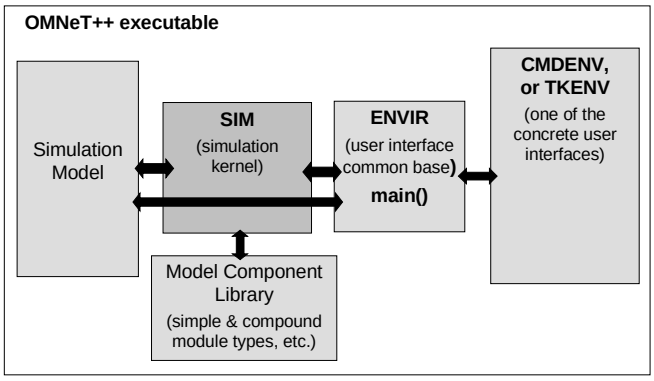

Fig. 2. Logical Architecture of an OMNeT++ Simulation Program

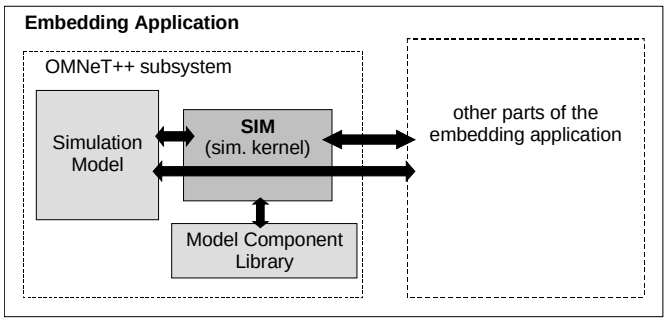

Fig. 3. Embedding OMNeT++

The Model Component Library consists of the code of compiled simple and compound modules. Modules are instantiated and the concrete simulation model is built by the simulation kernel and class library (*Sim*) at the beginning of the simulation execution. The simulation executes in an environment provided by the user interface libraries (*Envir*, *Cmdenv* and *Tkenv*) – this environment defines where input data come from, where simulation results go to, what happens to debugging output arriving from the simulation model, controls the simulation execution, determines how the simulation model is visualized and (possibly) animated, etc.

By replacing the user interface libraries, one can customize the full environment in which the simulation runs, and even embed an OMNeT++ simulation into a larger application (Fig. 3.). This is made possible by the existence of a generic interface between *Sim* and the user interface libraries, as well as the fact that all *Sim*, *Envir*, *Cmdenv* and *Tkenv* are physically separate libraries. It is also possible for the embedding application to assemble models from the available module types on the fly – in such cases, model topology will often come from a database.

Animation and Tracing Facility

An important requirement for OMNeT++ was easy debuggability and traceability of simulation models. Associated features are

implemented in *Tkenv*, the GUI user interface of OMNeT++. *Tkenv* uses three methods: automatic animation, module output windows and object inspectors. *Automatic animation* (i.e. animation without any programming) in OMNeT++ is capable of animating the flow of messages on network charts and reflecting state changes of the nodes in the display. Automatic animation perfectly fits the application area, as network simulation applications rarely need fully customizable, programmable animation capabilities.

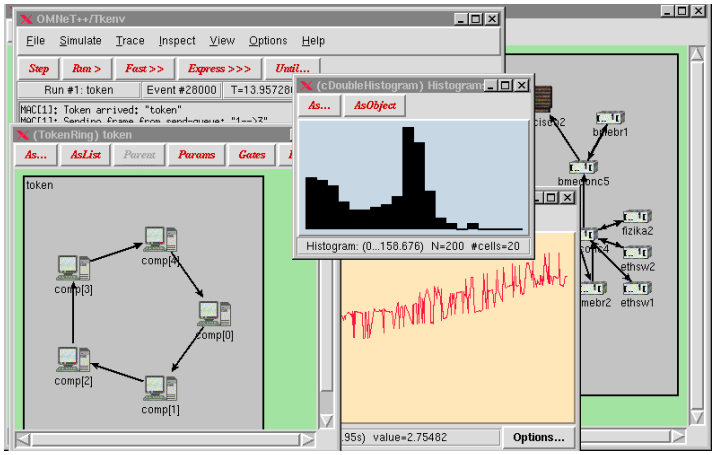

Fig. 4. Screenshot of the *Tkenv* User Interface of OMNeT++

Simple modules may write textual debugging or tracing information to a special output stream. Such debug output appears in *module output windows*. It is possible to open separate windows for the output of individual modules or module groups, so compared to the traditional printf()-style debugging, module output windows make it easier to follow the execution of the simulation program.

Further introspection into the simulation model is provided by *object inspectors.* An *object inspector* is a GUI window associated with a simulation object. Object inspectors can be used to display the state or contents of an object in the most appropriate way (i.e. a histogram object is displayed graphically, with a histogram chart), as well as to manually modify the object. In OMNeT++, it is automatically possible to inspect every simulation object, there is no need to write additional code in the simple modules to make use of inspectors.

It is also possible to turn off the graphical user interface altogether, and run the simulation as a pure command-line program. This feature is useful for batched simulation runs.

COMPARISON WITH OTHER SIMULATION TOOLS

There are numerous network simulation tools on the market today, both commercial and non-commercial ones. This section gives a very brief overview by picking some of the most important or most representative ones in both categories and comparing them to OMNeT++: Parsec (Bagrodia et al. 1998), SMURPH (Gburzynski 1996), ns (Bajaj et al. 2000), Ptolemy (Davis et al. 1999), NetSim++ (Maranda et al. 1996), C++Sim (Little and McCue 1993), CLASS (Marsan et al. 1994) as non-commercial, and OPNET (OPNET Technologies, Inc.), EcoPREDICTOR (formerly COMNET III; Compuware Corp.) as commercial tools. Among them, OPNET and ns deserve the most attention, as they are the most widely accepted and most mature packages, targeted roughly at the same segment of network simulation as OMNeT++ is.

The OMNeT++ home page (Varga 1997) contains a list of Web sites with collections of references to network simulation tools where the reader can get a more complete list. A fairly detailed comparison of OMNeT++ vs OPNET and Parsec can be found in the OMNeT++ User Manual.

Programmability

Does the simulation tool have the necessary power to express details in the model? In other words, can the user implement arbitrary new building blocks like in OMNeT++ or he is confined to the predefined blocks implemented by the supplier? Some tools like EcoPREDICTOR are not programmable by the user to this extent therefore they cannot be compared to OMNeT++. Specialized network simulation tools like CLASS (used specifically for ATM research) also rather fall into this category.

Model libraries and available models

What protocol models are readily available for the simulation tool? OPNET has probably the largest selection of ready-made protocol models (including TCP/IP, ATM, Ethernet, etc.). ns also has a large number of protocol models, mostly centered around TCP/IP. CLASS only supports ATM networks. As it was mentioned earlier, OMNeT++ currently has detailed TCP/IP, SCSI and FDDI models.

Support for structured, reusable simulation models

Does the simulation tool enforce separation of topology and functionality? Does it support reusable model components and hierarchical models? Network simulation tools naturally share the property that a model ("network") consists of "nodes" (blocks, entities, modules, etc.) connected by "links" (channels, connections, etc.). Some simulation tools (Parsec, C++Sim) do not provide explicit support for topology description: in Parsec, one must program a "driver entity" which boots the model by creating the necessary nodes and interconnecting them. This solution does not enforce the separation of defining model structure from defining the functionality, and possibilities for model component reuse are rather poor. Other tools (ns, CLASS)

do not allow hierarchy (nesting) in the network, which allows less flexibility in the design of the model.

ns uses Tcl scripts as the means of defining network topology. This allows significant flexibility in building the topology, but also makes it nearly impossible to create a graphical editor for "ns models" in general.

OPNET allows hierarchical models with arbitrarily deep nesting (like OMNeT++), but with some restrictions (namely, the "node" level cannot be hierarchical). A significant difference from OMNeT++ is that OPNET models are always of fixed topology, while OMNeT++'s NED and its graphical editor allow parameterized topologies. In OPNET, the preferred way of defining network topology is by using the graphical editor. The editor stores models in a proprietary binary file format, which means in practice that OPNET models are usually difficult to generate by program (it requires writing a lengthy C program that uses an OPNET API, while OMNeT++ models are simple text files which can be generated e.g. with perl).

Programming the Components

What is the programming model supported by the simulation environment? What functionality does the simulation library offer? For programming the components, network simulators typically use either a thread/coroutine-based programming model (Parsec, C++Sim), or FSMs built upon a message-receiving function (OPNET, ns, SMURPH and NetSim++). OMNeT++ is the only one among the examined tools which supports both programming models.

The simulation library of Parsec and C++Sim only provide very basic functions (like random number generation). OPNET and OMNeT++ provide rich simulation libraries of roughly comparable functionalities. The OPNET simulation library is based on C, while the one in OMNeT++ is a C++ class library.

Debugging and Tracing Support

What debugging or tracing facilities does the simulation tool offer? Simulation programs are infamous for long debugging periods and it is usually difficult to verify if they operate correctly, so support for efficient debugging and tracing is an important issue. Tracing via log messages (using e.g. printf()) is available to all simulation programs. Another useful tool is OPNET's powerful command-line simulation debugger. From the three tools OMNeT++ offers for debugging purposes (module output windows, inspectors, automatic animation), module output windows and inspectors are missing from all other fully programmable simulation tools listed, partly due to the lack of a graphical runtime environment. Animation, however, is supported by several of them.

All examined simulation environments that support some animation do provide automatic animation. Some of them (e.g. OPNET) also support customizable animation which OMNeT++ does not. Technically, animation comes in three flavors: integrated, client-server (i.e. viewer runs as a separate process), or off-line (i.e. record & playback). For example, OPNET is capable of client-server and off-line animation, NetSim++ and Ptolemy supports client-server animation, while ns uses off-line animation. OMNeT++ provides integrated animation. All three methods have their advantages and disadvantages; however, since client-server and off-line animation is difficult to combine with object inspectors (none of the listed simulation environments attempt to do so), they are significantly less efficient for debugging than integrated animation.

Performance

What performance can be expected from the simulation? Many network simulation scenarios require execution times of several hours. Probably the most important factor for execution speed is the programming language. C/C++-based simulation tools deliver good performance, esp. over Java-based ones. Most simulation tools examined (ns, Parsec, OPNET, C++SIM, NetSim++, SMURPH, Ptolemy) can be programmed in C, C++, or a language based on them.

Source Availability

Is the simulation library available in source? The availability of the source code is not only necessary for embedding or modifying the simulation engine, but often also provides significant help in debugging simulation models. Commercial tools, and even the non-commercial tool Parsec do not provide source code to the simulation kernel (although OPNET ships with the sources of the protocol models). OMNeT++ – like ns and most other non-commercial tools – is fully open-source.

CONCLUSIONS

The paper presented a discrete event simulation system which was designed to support the simulation of computer networks, parallel and distributed systems. The main advantage of the simulation system is that (1) it allows building models "in the large", (2) it implements an easy and natural programming model, (3) it has a powerful GUI execution environment, and (4) it is open-source. The authors feel that OMNeT++ has the potential to become a widely used network simulation package in academic and research environments.

ACKNOWLEDGMENTS

The author would like to thank the Department of Telecommunications, Technical University of Budapest, and especially Dr. György Pongor for supporting the development of

OMNeT++. The author also acknowledges Gábor Lencse for his helping hand on numerous occasions.

REFERENCES

Varga, A. 1997. *OMNeT++ Home Page.* http://www.hit.bme.hu/phd/vargaa/omnetpp.htm

Kaage, U., V. Kahmann, F. Jondral. 2001. "An OMNeT++ TCP Model". To appear in *Proceedings of the European Simulation Multiconference (ESM 2001)*, June 7-9, Prague.

Wehrle, K, J. Reber, V. Kahmann. 2001. "A Simulation Suite for Internet Nodes with the Ability to Integrate Arbitrary Quality of Service Behavior". In *Proceedings of the Communication Networks and Distributed Systems Modeling and Simulation Conference 2001*, Phoenix (AZ), USA, January 7-11.

Erdei, M., A. Wagner, K. Sója, M. Székely. 2001. "A Networked Remote Simulation Architecture and its Remote OMNeT++ Implementation". To appear in *Proceedings of the European Simulation Multiconference (ESM 2001)*, June 7-9, Prague.

Wagner, A., M. Erdei. 2001. "Agent-Based Resource Management for Remote Simulation Systems and an Implementation for Remote OMNeT++". To appear in *Proceedings of the European Simulation Multiconference (ESM 2001)*, June 7-9, Prague.

Zeigler, B. 1990. *Object-oriented Simulation with Hierarchical, Modular Models*. Academic Press.

Chow, A and B. Zeigler. 1994. "Revised DEVS: A Parallel, Hierarchical, Modular Modeling Formalism". In *Proceedings of the Winter Simulation Conference*, Lake Buena Vista, FL.

Varga, A. and Gy. Pongor. 1997. "Flexible Topology Description Language for Simulation Programs". In *Proceedings of the 9th European Simulation Symposium (ESS'97)*, pp.225-229, Passau, Germany, October 19-22.

Varga, A and B. Fakhamzadeh. 1997. "The K-Split Algorithm for the PDF Approximation of Multi-Dimensional Empirical Distributions without Storing Observations". In *Proc. of the 9th European Simulation Symposium (ESS'97)*, pp.94-98. October 19-22, Passau, Germany.

Varga, A. 1998. "Parameterized Topologies for Simulation Programs". In Proceedings of the *Western Multiconference on Simulation (WMC'98), Communication Networks and Distributed Systems (CNDS'98)*. San Diego, CA, January 11-14.

Jain, R, and I. Chlamtac. 1985. "The P^2 Algorithm for Dynamic Calculation of Quantiles and Histograms Without Storing Observations". Communications of the ACM, 28, no.10 (Oct.): 1076-1085.

Lencse, G. 1997. "Efficient Simulation of Large Systems - Transient Behaviour and Accuracy". In *Proc. of the 9th European Simulation Symposium (ESS'97)*. October 19-22, Passau, Germany.

Lencse, G. 1998. "Efficient Parallel Simulation with the Statistical Synchronization Method". In *Proceedings of the Western Multiconference on Simulation (WMC'98), Communication Networks and Distributed Systems (CNDS'98)*. January 11-14, San Diego, CA.

Bajaj, S., L. Breslau, D. Estrin, K. Fall, S. Floyd, P. Haldar, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kumar, S. McCanne, R. Rejaie, P. Sharma, K. Varadhan, Y. Xu, H. Yu and D. Zappala. 2000. "Improving simulation for network research". *IEEE Computer*. (to appear, a preliminary draft is currently available as USC technical report 99-702)

Bagrodia, R, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, B. Park, H. Song. 1998. "Parsec: A Parallel Simulation Environment for Complex Systems", *Computer*, Vol. 31(10), October, pp. 77-85.

Little, M. C. and D. L. McCue. 1993. "Construction and Use of a Simulation Package in C++". Computing Science Technical Report, University of Newcastle upon Tyne, Number 437, July (also appeared in the *C User's Journal* Vol. 12 Number 3, March 1994).

Marsan, M A, R. Lo Cigno, M. Munafò, A. Tonietti. 1994. "Simulation of ATM Computer Networks with CLASS". In *Proceedings of the 7th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, Vienna, Austria, May 4-6.

Gburzynski, P. 1996. *Protocol Design for Local and Metropolitan Area Networks.* Prentice Hall.

Davis, J, M. Goel, C. Hylands, B. Kienhuis, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay and Y. Xiong. 1999. "Overview of the Ptolemy Project". ERL Technical Report UCB/ERL No. M99/37, Dept. EECS, University of California, Berkeley, CA 94720, July.

OPNET Technologies, Inc. *OPNET Modeler.* http://www.opnet.com

Compuware Corp., *EcoPREDICTOR.* http://www.compuware.com/products/ecosystems/ecopredictor

Maranda, A., R. Ghilea, E. Lazar. 1996. "Project NetSim++: A Switched Packets Network Simulator". In *Proceedings of ROSE'96, the 4th Romanian Open Systems Conference*, Bucharest, Oct.

AUTHOR BIOGRAPHY

ANDRÁS VARGA received his M.Sc. degree in computer science from the Faculty of Electrical Engineering and Informatics, Technical University of Budapest in 1994. Between 1994 and 1998 he was doing Ph.D. studies at the Department of Telecommunications. His research interests include discrete event simulation of large communication systems, nonparametric density estimation and parallel discrete event simulation. Currently he is employed by Brokat Technologies in Budapest.