AN OVERVIEW OF THE OMNeT++ SIMULATION ENVIRONMENT

András Varga
OpenSim Ltd.
Szőlő köz 11, 1032
Budapest, Hungary
andras.varga@omnest.com

Rudolf Hornig
OpenSim Ltd.
Szőlő köz 11, 1032
Budapest, Hungary
rudolf.hornig@omnest.com

ABSTRACT

The OMNeT++ discrete event simulation environment has been publicly available since 1997. It has been created with the simulation of communication networks, multiprocessors and other distributed systems in mind as application area, but instead of building a specialized simulator, OMNeT++ was designed to be as general as possible. Since then, the idea has proven to work, and OMNeT++ has been used in numerous domains from queuing network simulations to wireless and ad-hoc network simulations, from business process simulation to peer-to-peer network, optical switch and storage area network simulations. This paper presents an overview of the OMNeT++ framework, recent challenges brought about by the growing amount and complexity of third party simulation models, and the solutions we introduce in the next major revision of the simulation framework.[1]

KEYWORDS

discrete simulation, network simulation, simulation tools, performance analysis, computer systems, telecommunications, hierarchical, integrated development environment

1. INTRODUCTION

OMNeT++[1][2] is a C++-based discrete event simulator for modeling communication networks, multiprocessors and other distributed or parallel systems. OMNeT++ is public-source, and can be used under the Academic Public License that 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 and research-oriented commercial institutions for the simulation of computer networks and distributed or parallel systems. OMNeT++ attempts to fill the gap between open-source, research-oriented simulation software such as NS-2 [11] and expensive commercial alternatives like OPNET [16]. A later section of this paper presents a comparison with other simulation packages. OMNeT++ is available on all common platforms including Linux, Mac OS/X and Windows, using the GCC tool chain or the Microsoft Visual C++ compiler.

SIMUTools, March 03 – 07, 2008, Marseille, France.
ISBN 978-963-9799-20-2

[1] The 4.0 release is scheduled to appear in Q1 2008.

OMNeT++ represents a framework approach. Instead of directly providing simulation components for computer networks, queuing networks or other domains, it provides the basic machinery and tools to write such simulations. Specific application areas are supported by various simulation models and frameworks such as the Mobility Framework or the INET Framework. These models are developed completely independently of OMNeT++, and follow their own release cycles.

Since its first release, simulation models have been developed by various individuals and research groups for several areas including: wireless and ad-hoc networks, sensor networks, IP and IPv6 networks, MPLS, wireless channels, peer-to-peer networks, storage area networks (SANs), optical networks, queuing networks, file systems, high-speed interconnections (InfiniBand), and others. Some of the simulation models are ports of real-life protocol implementations like the Quagga Linux routing daemon or the BSD TCP/IP stack, others have been written directly for OMNeT++. A later section of this paper will discuss these projects in more detail. In addition to university research groups and non-profit research institutions, companies like IBM, Intel, Cisco, Thales and Broadcom are also using OMNeT++ successfully in commercial projects or for in-house research.

2. THE DESIGN OF OMNeT++

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

- To enable large-scale simulation, simulation models need to be hierarchical, and built from reusable components as much as possible.
- The simulation software should facilitate visualizing and debugging of simulation models in order to reduce debugging time, which traditionally takes up a large percentage of simulation projects. (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 simulations 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.
- Should provide an Integrated Development Environment that largely facilitates model development and analyzing results.

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

2.1 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. Messages can be sent either via connections that span between modules or directly to their destination modules. The concept of simple and compound modules is similar to DEVS [46][47] atomic and coupled models.

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 a network module which is a special compound module type without gates to the external world. 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 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. The feasibility of model reuse is proven by the model frameworks like INET Framework [1] and Mobility Framework [17][18], and their extensions.

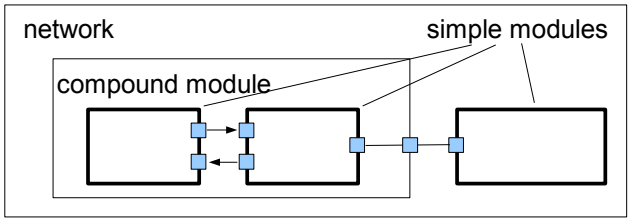

Figure 1. Model Structure in OMNeT++. Boxes represent simple modules (thick border), and compound modules (thin border). Arrows connecting small boxes represent connections and gates.

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. Connections spanning across hierarchy levels are not permitted, as it would hinder model reuse. 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. Properties such as propagation delay, data rate and bit error rate, can be assigned to connections. 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 boolean 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.

2.2 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. Network definitions are compound modules that qualify as self-contained simulation models.

The NED language has been designed to scale well, however, recent growth in the amount and complexity of OMNeT++-based simulation models and model frameworks made it necessary to improve the NED language as well. In addition to a number of smaller improvements, the following major features have been introduced:

Inheritance. Modules and channels can now be subclassed. Derived modules and channels may add new parameters, gates, and (in the case of compound modules) new submodules and connections. They may set existing parameters to a specific value, and also set the gate size of a gate vector. This makes it possible, for example, to take a `GenericTCPClientApp` module and derive an `FTPApp` from it by setting certain parameters to a fixed value; or derive a `WebClientHost` compound module from a BaseHost compound module by adding a `WebClientApp` submodule and connecting it to the inherited `TCP` submodule.

Interfaces. Module and channel interfaces can be used as a placeholder where normally a module or channel type would be used, and the concrete module or channel type is determined at network setup time by a parameter. Concrete module types have to "implement" the interface they can substitute. For example, the module types `ConstSpeedMobility` and `RandomWayPointMobility` need to implement `IMobility` to be able to be plugged into a `MobileHost` that contains an `IMobility` submodule.

Packages. To address name clashes between different models and to simplify specifying which NED files are needed by a specific

simulation model, a Java-like package structure was introduced into the NED language.

Inner types. Channel types and module types used locally by a compound module can now be defined within the compound module, in order to reduce namespace pollution.

Metadata annotations. It is possible to annotate module or channel types, parameters, gates and submodules by adding *properties*. Metadata are not used by the simulation kernel directly, but they can carry extra information for various tools, the runtime environment, or even for other modules in the model. For example, a module's graphical representation (icon, etc) or the prompt string and unit (milliwatt, etc) of a parameter are specified using properties.

The NED language has an equivalent XML representation, that is, NED files can be converted to XML and back without loss of data, including comments. This lowers the barrier for programmatic manipulation of NED files, for example extracting information, refactoring and transforming NED, generating NED from information stored in other system like SQL databases, and so on.

2.3 Graphical Editor

The OMNeT++ package includes an Integrated Development Environment which contains a graphical editor using 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 parametric 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 parametric topologies and associated design patterns have been investigated in [7][9]. With parametric topologies, NED holds an advantage in many simulation scenarios both over OPNET where only fixed model topologies can be designed, and over NS-2 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.

2.4 Separation of Model and Experiments

It is always a good practice to try to separate the different aspects of a simulation as much as possible. Model *behavior is captured in C++ files* as code, while *model topology* (and of course the parameters defining this topology) *is defined by the NED files*. This approach allows the user to keep the different aspects of the model in different places which in turn allows having a cleaner model and better tooling support. In a generic simulation scenario, one usually wants to know how the simulation behaves with different inputs. These variables neither belong to the behavior (code) nor the topology (NED files) as they can change from run to run. INI files are used to store these values. INI files provide a great way to specify how these parameters change and enable us to run our simulation for each parameter combination we are interested in. The generated simulation results can be easily harvested and processed by the built in analysis tool. We will explore later, in the Result Analysis paragraph, how the INI files are organized and how they can make experimenting with our model a lot easier.

2.5 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. OMNeT++ provides an Integrated C++ Development Environment so it is possible to write, run and debug the code without leaving the OMNeT++ IDE. 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, 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). Messages can be defined by specifying their content in an MSG file. OMNeT++ takes care of

creating the necessary C++ classes. MSG files allow the OMNeT++ kernel to generate reflection code which enables us to peek into messages and explore their content at runtime.

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 parametric internal topology can be created on the fly.

2.6 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 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 consistent use of object-oriented techniques makes the simulation kernel compact and slim. This makes it relatively easy to understand its internals, which is a useful property for both debugging and educational use.

Recently it has become more common to do large scale network simulations with OMNeT++, with several ten thousand or more network nodes. To address this requirement, aggressive memory optimization has been implemented in the simulation kernel, based on shared objects and copy-on-write semantics.

Until recently, simulation time has been represented as with C's `double` type (IEEE double precision). Well-known precision problems with floating point calculations however, have caused problems in simulations from time to time. To address this issue, simulation time has been recently changed to 64-bit integer-based fixed-point representation. One of the major problems that had to be solved here was how to detect numeric overflows, as the C and C++ languages, despite their explicit goals of being “close to the hardware”, lack any support to detect integer overflows.

2.7 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 [10] and the *k-split* [8] 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.

2.8 Parallel Simulation Support

OMNeT++ also has support for parallel simulation execution. Very large simulations may benefit from the parallel distributed simulation (PDES) feature, either by getting speedup, or by distributing memory requirements. If the simulation requires several Gigabytes of memory, distributing it over a cluster may be the only way to run it. For getting speedup (and not actually slowdown, which is also easily possible), the hardware or cluster should have low latency and the model should have inherent parallelism. Partitioning and other configuration can be configured in the INI file, the simulation model itself doesn't need to be changed (unless, of course, it contains global variables that prevents distributed execution in the first place.) The communication layer is MPI, but it's actually configurable, so if the user does not have MPI it is still possible to run some basic tests over named pipes. The figure below explains the logical architecture of the parallel simulation kernel:

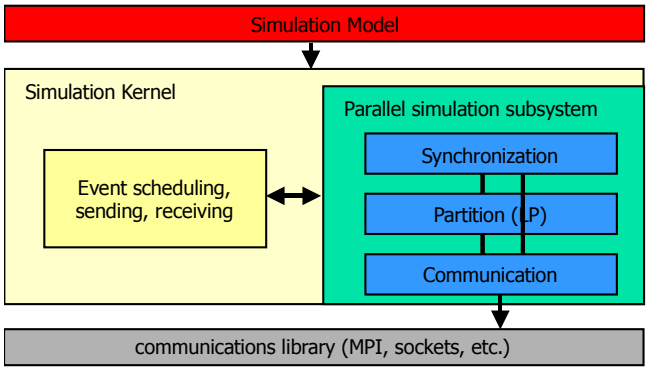

Figure 2. Logical Architecture of the OMNeT++ Parallel Simulation kernel

2.9 Internal Architecture

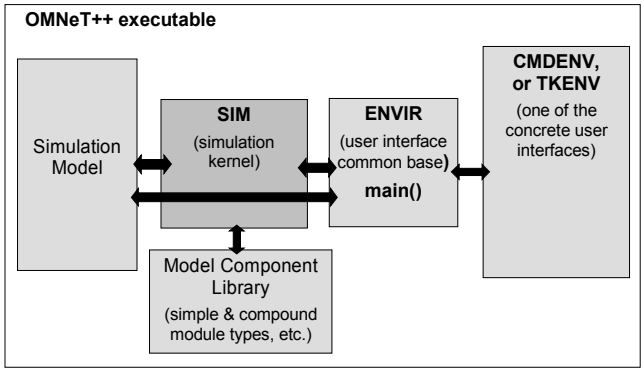

Figure 3. Logical Architecture of an OMNeT++ Simulation Program

OMNeT++ simulation programs possess a modular structure. The logical architecture is shown on Figure 3.

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.

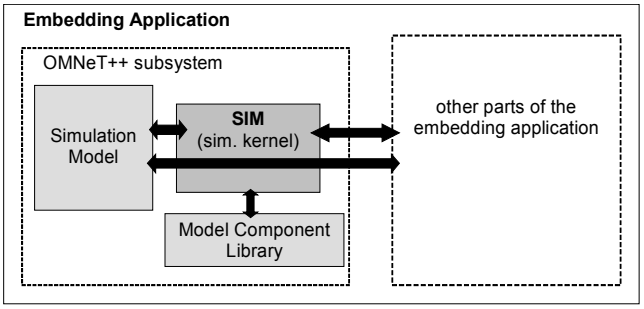

Figure 4. Embedding OMNeT++

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 (Figure 4). 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.

2.10 Real-Time Simulation, Network Emulation

Network emulation, together with real-time simulation and hardware-in-the-loop like functionality, is available because the event scheduler in the simulation kernel is pluggable too. The OMNeT++ distribution contains a demo of real-time simulation and a simplistic example of network emulation. Interfacing OMNeT++ with other simulators (hybrid operation) or HLA is also largely a matter of implementing one's own scheduler class.

2.11 Animation and Tracing Facility

An important requirement for OMNeT++ was easy debugging 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.

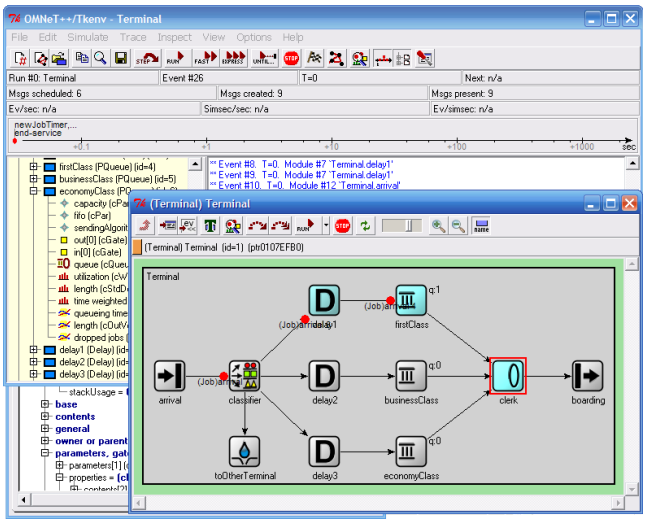

Figure 5. Screenshot of the *Tkenv* User Interface of OMNeT++

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.

2.12 Visualizing Dynamic Behavior

The behavior of large and complex models is usually hard to understand because of the complex interaction between different modules. OMNeT++ helps to reduce complexity by mandating the communication between modules using predefined connections. The graphical runtime environment allows the user to follow module interactions to a certain extent: one can animate, slow down or single-step the simulation, but sometimes it is still hard to see the exact sequence of the events, or to grasp the timing relationships (as, for practical reasons, simulation time is not proportional to real time; also, when single-stepping through events, events with the same timestamp get animated sequentially).

OMNeT++ helps the user to visualize the interaction by logging interactions between modules to a file. This log file can be processed after (or even during) the simulation run and can be used to draw interaction diagrams. The OMNeT++ IDE has a sequence chart diagramming tool which provides a sophisticated view of how the events follow each other. One can focus on all, or

just selected modules, and display the interaction between them. The tool can analyze and display the causes or consequences of an event, and display all of them (using a non-linear time axis) on a single screen even if time intervals between events are of different magnitudes. One can go back and forth in time and filter for modules and events.

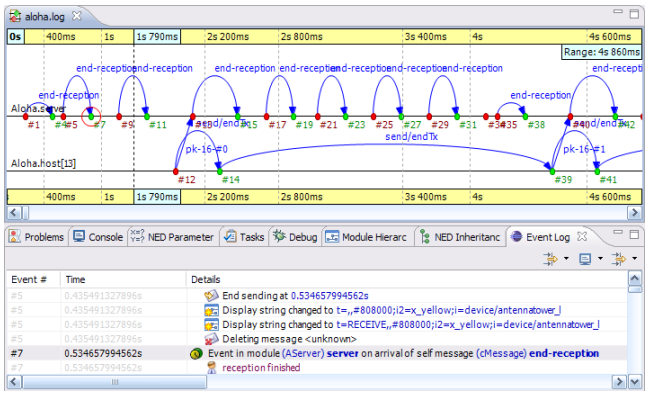

Figure 6.
Screenshot of a Sequence Chart from the OMNeT++ IDE

2.13 Organizing and Performing Experiments

The ultimate goal of running a simulation is to obtain results and to get some insight into the system by analyzing the results. Thorough simulation studies very often produce large amounts of data, which are nontrivial to organize in a meaningful way. OMNeT++ organizes simulation runs (and the results they generate) around the following concepts:

model – the executable (C++ model files, external libraries, etc.) and NED files. (INI files are considered to be part of the *study* and *experiment* rather than the *model*.) Model files are considered to be invariant for the purposes of experimentation, meaning that if a C++ source or NED file gets modified, then it will count as a different *model*.

study – a series of *experiments* to study some phenomenon on one or more *models*; e.g. "handover optimization for mobile IPv6". For a *study* one usually performs a number of *experiments* from which conclusions can be drawn. One *study* may contain *experiments* on different *models*, but one *experiment* is always performed on one specific *model*.

experiment – exploration of a parameter space on a *model*, e.g. "*the* `adhocNetwork` *model's behavior with numhosts=5,10,20,50,100 and load=2..5 step 0.1 (Cartesian product)*"; consists of several *measurements*.

measurement – a set of simulation *runs* on the same *model* with the same parameters (e.g. "numhosts=10, load=3.8"), but potentially different seeds. May consist of several *replications* of whose results get averaged to supply one data point for the *experiment*. A *measurement* can be characterized with the parameter settings and simulation kernel settings in the INI file, minus the seeds.

replication – one repetition of a *measurement*. Very often, one would perform several *replications*, all with different seeds. A replication can be characterized by the seed values it uses.

run – or actual run: one instance of running the simulation; that is, a run can be characterized with an exact time/date and the computer (e.g. the host name).

OMNeT++ supports the execution of whole (or partial) experiments as a single batch. After specifying the *model* (executable file + NED files) and the *experiment* parameters (in the INI file) one can further refine which *measurements* one is interested in. The simulation batch can be executed and its progress monitored from the IDE. Multiple CPUs or CPU cores can be exploited by letting the launcher run more than one simulation at a time. The significance of running multiple independent simulations concurrently is often overlooked, but it is not only a significantly easier way of reducing overall execution time of an experiment than distributed parallel simulation (PDES) but also more efficient (as it guarantees linear speedup which is not possible with PDES).

2.14 Result Analysis

Analyzing the simulation result is a lengthy and time consuming process. In most cases the user wants to see the same type of data for each run of the simulation or display the same graphs for different modules in the model, so automation is very important. (The user does not want to repeat the steps of re-creating charts every time simulations have to be re-run for some reason.) The lack of automation support drives many users away from existing GUI analysis tools, and forces them to write scripts.

OMNeT++ solves this by making result analysis rule-based. Simulations and series of simulations produce various result files. The user selects the input of the analysis by specifying file names or file name patterns (e.g. "adhoc-*.vec"). Data of interest can be selected into datasets by further pattern rules. The user completes datasets by adding various processing, filtering and charting steps, all using the GUI (Figure 7). Whenever the underlying files or their contents change, dataset contents and charts are recalculated. The editor only saves the "recipe" and not the actual numbers, so when simulations are re-run and so result files get replaced, charts are automatically up-to-date. Data in result files are tagged with meta information: experiment, measurement and replication labels are added to the result files to make the filtering process easy. It is possible to create very sophisticated filtering rules, for example, "all 802.11 retry counts of host[5..10] in experiment X, averaged over replications". In addition datasets can use other datasets as their input so datasets can build on each other.

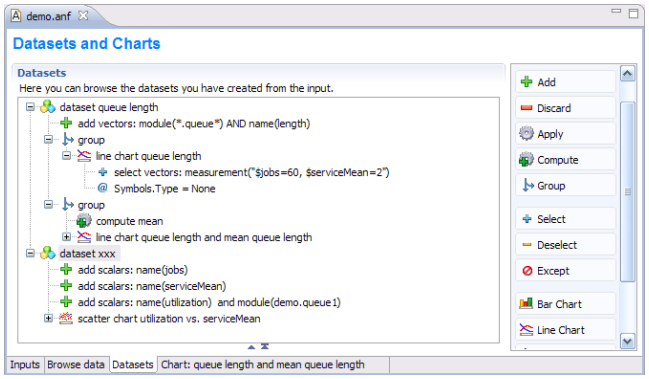

Figure 7. Rule based processing

OMNeT++ supports several fully customizable chart and graph types which are rendered directly from datasets (Figure 8). The visual properties of the charts are also stored in the "recipe".

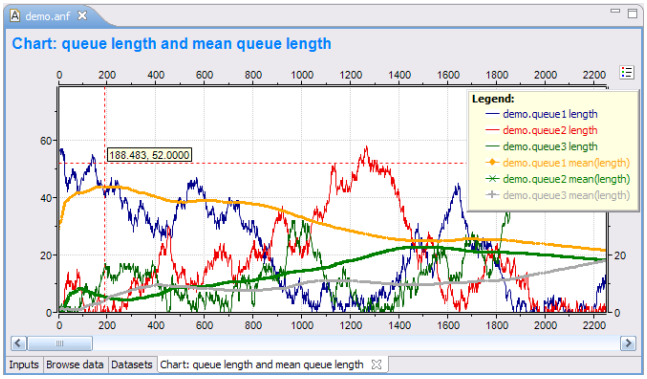

Figure 8. Charts in the OMNeT++ IDE

3. CONTRIBUTIONS TO OMNeT++

Currently there are two major network simulation model frameworks for OMNeT++: the Mobility Framework [17][18] and the INET Framework [1].

The Mobility Framework was designed at TU Berlin to provide solid foundations for creating wireless and mobile networks within OMNeT++. It provides a detailed radio model, several mobility models, MAC models including IEEE 802.11b, and several other components. Other model frameworks for mobile, ad-hoc and sensor simulations [26][33][13] have also been published (LSU SenSim [25][26] and Castalia [19][20], for example), but they have so far failed to make significant impact. Further related simulation models are NesCT for TinyOS [21] simulations, MACSimulator and Positif [13] which are continued in the MiXiM [5] project, EWsnSim, SolarLEACH, ChSim [27], AdHocSim, AntNet, etc.

The INET Framework has evolved from the IPSuite originally developed at the University of Karlsruhe. It provides detailed protocol models for TCP, IPv4, IPv6, Ethernet, Ieee802.11b/g, MPLS, OSPFv4, and several other protocols. INET also includes the Quagga routing daemon directly ported from Linux code base.

Several authors have developed various extensions for the INET Framework. OverSim [22][23][24] is used to model P2P protocols on top of the INET Framework. AODV-UU, DSR is also available as an add-on for the INET Framework. IPv6Suite [45] (discontinued by 2007) supported MIPv6 and HMIPv6 simulations over wired and wireless networks.

The OppBSD [44] model allows using the FreeBSD kernel TCP/IP protocol stack directly inside an OMNeT++ simulation. Other published simulation models include Infiniband [28], FieldBus [14] and SimSANs [43].

A very interesting application area of OMNeT++ is the modeling of dynamic behavior of software systems based on the UML standard, by translating annotated UML diagrams into OMNeT++ models. A representative of this idea is the SYNTONY project [30][31][32]; similar approach have been reported in [35] where the authors used UML-RT, and in [34] where performance characteristics of web applications running on the JBoss Application Server were studied.

The Simulation Library API can be mapped to programming languages other than C++. There is already 3rd party support for Java and C# which makes it possible to write simple module behavior in these languages.

4. COMPARISON WITH OTHER SIMULATION TOOLS

The network simulation scene has changed a lot in the past ten years, simulation tools coming and going. This section presents an overview of various commercial and noncommercial network simulation tools in wide use today, and compares them to OMNeT++. Specialized network simulators (like TOSSIM, for TinyOS simulations), and simulation packages not or rarely used for network simulations (such as Ptolemy or Ptolemy II) are not considered. Also, the discussion only covers the features and services of the simulation environments themselves, but not the availability or characteristics of specific simulation models like IPv6 or QoS (the reason being that they do not form part of the OMNeT++ simulation package.)

4.1 NS

NS-2 [11] is currently the most widely used network simulator in academic and research circles. NS-2 does not follow the same clear separation of simulation kernel and models as OMNeT++: the NS-2 distribution contains the models together with their supporting infrastructure, as one inseparable unit. This is a key difference: the NS-2 project goal is to build a *network simulator,* while OMNeT++ intends to provide a *simulation platform*, on which various research groups can build their own simulation frameworks. The latter approach is what called the abundance of OMNeT++-based simulation models and model frameworks into existence, and turned OMNeT++ into a kind of an "ecosystem".

NS-2 lacks many tools and infrastructure components that OMNeT++ provides: support for hierarchical models, a graphical editor, GUI-based execution environment (except for *nam*), separation of models from experiments, graphical analysis tools, simulation library features such as multiple RNG streams with arbitrary mapping and result collection, seamlessly integrated parallel simulation support, etc. This is because the NS-2 project concentrates on developing the simulation models, and much less on simulation infrastructure.

NS-2 is a dual-language simulator: simulation models are Tcl scripts[2], while the simulation kernel and various components (protocols, channels, agents, etc) are implemented in C++ and are made accessible from the Tcl language. Network topology is expressed as part of the Tcl script, which usually deals with several other things as well, from setting parameters to adding application behavior and recording statistics. This architecture makes it practically impossible to create graphical editors for NS-2 models[3].

NS-3 is an ongoing effort to consolidate all patches and recently developed models into a new version of NS. Although work includes refactoring of the simulation core as well, the concepts

[2] In fact, OTcl, which is an object-oriented extension to Tcl.

[3] Generating a Tcl script from a graphical representation is of course possible, but not the other way round: no graphical editor will ever be able to understand an arbitrary NS-2 script, and let the user edit it graphically.

are essentially unchanged. The NS-3 project goals [36] include some features (e.g. parallel simulation, use of real-life protocol implementations as simulation models) that have already proven to be useful with OMNeT++.

4.2 J-Sim

J-Sim [37][38] (formerly known as JavaSim) is a component-based, compositional simulation environment, implemented in Java. J-Sim is similar to OMNeT++ in that simulation models are hierarchical and built from self-contained components, but the approach of assembling components into models is more like NS-2: J-Sim is also a dual-language simulation environment, in which classes are written in Java, and glued together using Tcl (or Java). The use of Tcl in J-Sim has the same drawback as with NS-2: it makes implementing graphical editors impossible. In fact, J-Sim does provide a graphical editor *(gEditor)*, but its native format is XML. Although *gEditor* can export Tcl scripts, developers recommend that XML files are directly loaded into the simulator, bypassing Tcl. This way, XML becomes the equivalent of OMNeT++ NED. However, the problem with XML as native file format is that it is hard to read and write by humans.

Simulation models are provided in the Inet package, which contains IPv4, TCP, MPLS and other protocol models.

The fact that J-Sim is Java-based has some implications. On one hand, model development and debugging can be significantly faster than C++, due to existence of excellent Java development tools. However, simulation performance is significantly weaker than with C++, and it is also not possible to reuse existing real-life protocol implementations written in C as simulation models. (The feasibility and usefulness of the latter has been demonstrated with OMNeT++, where simulation models include port of the Quagga Linux routing daemon, the TCP stack from the FreeBSD kernel, the port of the UU-AODV routing package, etc. The NS-3 team has similar plans as well.)

Development of the J-Sim core and simulation models seem to have stalled after 2004 when version 1.3 was published; later entries on the web site are patches and contributed documents only. There are no independent (3rd party) simulation models for J-Sim.

4.3 SSFNet

SSFNet [39] (Scalable Simulation Framework) is defined as a "public-domain standard for discrete-event simulation of large, complex systems in Java and C++." The SSFNet standard defines a minimalist API (which, however, was designed with parallel simulation in mind). The topology and configuration of SSFNet simulations are given in DML files. DML is a text-based format comparable to XML, but has its own syntax. DML can be considered the SSFNet equivalent of NED, however it lacks expressing power and features to scale up to support large model frameworks built from reusable components. SSFNet also lacks OMNeT++'s INI files, all parameters need to be given in the DML.

SSFNet has four implementations: DaSSF and CSSF in C++, and two Java implementations (Renesys Raceway and JSSF). There were significantly more simulation models developed for the Java versions than for DaSSF. Advantages and disadvantages of using Java in SSFNet are the same as discussed with J-Sim.

As with J-Sim, development of the SSFNet simulation framework and models seem to have stalled after 2004 (date of the SSFNet for Java 2.20 release), and little activity can be detected outside the main web site as well.

4.4 JiST and SWANS

JiST [42][6] represents a very interesting approach to building a high performance Java based simulation environment. It modifies the Java Virtual Machine to run the programs in simulation time instead of real time. JiST is basically just a simulation kernel, and as such, it lacks most of the features present in the OMNeT++ package.

SWANS is a scalable wireless network simulator built atop the JiST platform as a proof of concept model, to prove the efficiency of the virtual machine based approach. It appears that no further simulation models have been created by the JiST team or independent groups. Development of JiST/SWANS seems to be halted after 2005.

4.5 OPNET Modeler

OPNET Modeler is the flagship product of OPNET Technologies Inc. [16]. OPNET Modeler is a commercial product which is freely available worldwide to qualifying universities. OPNET has probably the largest selection of ready-made protocol models (including IPv6, MIPv6, WiMAX, QoS, Ethernet, MPLS, OSPFv3 and many others).

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. OPNET's architecture is similar to OMNeT++ as it 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 parametric 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 C program that uses an OPNET API, while OMNeT++ models are simple text files which can be generated e.g. with Perl).

Both OPNET and OMNeT++ provide a graphical debugger and some form of automatic animation which is essential for easy model development.

OPNET does not provide source code to the simulation kernel (although it ships with the sources of the protocol models). OMNeT++ – like NS-2 and most other non-commercial tools – is fully public-source allowing much easier source level debugging.

OPNET's main advantage over OMNeT++ is definitely its large protocol model library, while its closed nature (proprietary binary file formats and the lack of source code) makes development and problem solving harder.

4.6 Qualnet

Qualnet [41] is a commercial simulation environment mainly for wireless networks, which has a significant client base in the

military. Qualnet has evolved from the Parsec parallel simulation "language"[4] [12] developed at the UCLA Parallel Computing Laboratory (PCL), and the GloMoSim (Global Mobile system Simulation) model written on top of Parsec. The Parsec language divides the simulation model into *entities*, and provides a minimalistic simulation API (timers, etc) for them. Entities are implemented with coroutines. Because coroutine CPU stacks require relatively large amounts of memory (the manual recommends reserving 200KByte each), it is rarely feasible to map the natural units of the simulation (say, hosts and routers, or protocols) one-to-one onto entities. What GloMoSim and Qualnet models do is implement the equivalent of the OMNeT++ model structure in model space, above the Parsec runtime. The Parsec kernel is only used to provide event scheduling and parallel simulation services.

Parsec provides a very efficient parallel simulation infrastructure, and models (GloMoSim and Qualnet simulation models) have been written with parallel execution in mind[5], resulting in an excellent parallel performance for wireless network simulations.

4.7 Summary

In this section we have examined the simulation packages most relevant for analysis of telecommunication networks, and compared them to OMNeT++. NS-2 is still the most widely used network simulator in the Academia, but it lacks much of the infrastructure provided by OMNeT++. The other three open-source network simulation packages examined (J-Sim, SSFNet and JiST/SWANS), have failed to gain significant acceptance, and their project web pages indicate near inactivity since 2004.

We have examined two commercial products as well. Qualnet emphasizes wireless simulations. OPNET has similar foundations as OMNeT++, but ships with an extensive model library and provides several additional programs and GUI tools.

5. CONCLUSIONS

In this paper we presented an overview of the OMNeT++ discrete event simulation platform, designed to support the simulation of telecommunication networks and other parallel and distributed systems. The OMNeT++ approach significantly differs from that of NS-2, the most widely used network simulator in academic and research circles: while the NS-2 (and NS-3) project goal is to build a network simulator, OMNeT++ aims at providing a rich simulation platform, and leaves creating simulation models to independent research groups. The last ten years have shown that the OMNeT++ approach is viable, and several OMNeT++-based open-source simulation models and model frameworks have been published by various research groups and individuals.

6. REFERENCES

[1] OMNeT++ Home Page. http://www.omnetpp.org [accessed on September, 2007]

[2] Varga, A. 2001. The OMNeT++ Discrete Event Simulation System. In the Proceedings of the European Simulation Multiconference (ESM2001. June 6-9, 2001. Prague, Czech Republic).

[3] 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.

[4] 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.

[5] MiXiM home page. http://sourceforge.net/projects/mixim/ [accessed on September, 2007]

[6] JiST home page. http://jist.ece.cornell.edu [accessed on September, 2007]

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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)

[12] 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.

[13] Consensus home page. http://www.consensus.tudelft.nl/software.html [accessed on September, 2007]

[14] FieldBus home page. http://developer.berlios.de/projects/fieldbus [accessed on September, 2007]

[15] 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.

[16] OPNET Technologies, Inc. *OPNET Modeler.* http://www.opnet.com [accessed on September, 2007]

[17] Mobility Framework. http://mobility-fw.sourceforge.net [accessed on September, 2007]

[18] W. Drytkiewicz, S. Sroka, V. Handziski, A. Koepke, and H. Karl, A Mobility Framework for OMNeT++. 2003. 3rd International OMNeT++ Workshop (Budapest University of Technology and Economics, Department of

[4] It extends the C language with some constructs, and Parsec programs are translated into C before compilation.

[5] Lookahead annotations, avoiding central components, etc.

Telecommunications Budapest, Hungary, January 2003). http://www.tkn.tu-berlin.de/~koepke/

[19] D. Pediaditakis, S. H. Mohajerani, and A. Boulis. 2007. Poster Abstract: Castalia: the Difference of Accurate Simulation in WSN. 4th European Conference on Wireless Sensor Networks, (Delft, The Netherlands, 29-31 January 2007).

[20] Castalia: A Simulator for WSN. http://castalia.npc.nicta.com.au. [accessed on September, 2007]

[21] NesCT: A language translator. http://nesct.sourceforge.net [accessed on September, 2007]

[22] OverSim:The Overlay Simulation Framework http://www.oversim.org [accessed on September, 2007]

[23] Ingmar Baumgart and Bernhard Heep and Stephan Krause. 2007. OverSim: A Flexible Overlay Network Simulation Framework. Proceedings of 10th IEEE Global Internet Symposium (May, 2007). p.79-84.

[24] Ingmar Baumgart and Bernhard Heep and Stephan Krause. 2007. A P2PSIP Demonstrator Powered by OverSim. Proceedings of 7th IEEE International Conference on Peer-to-Peer Computing (P2P2007, Galway, Ireland. Sep, 2007). pp. 243-244,

[25] C. Mallanda, A. Suri, V. Kunchakarra, S.S. Iyengar, R. Kannan, A. Durresi, and S. Sastry. 2005. Simulating Wireless Sensor Networks with OMNeT++ , submitted to IEEE Computer, 2005 http://csc.lsu.edu/sensor_web/publications.html

[26] Sensor Simulator. http://csc.lsu.edu/sensor_web [accessed on September, 2007]

[27] S. Valentin. 2006. ChSim - A wireless channel simulator for OMNeT++, (TKN TU Berlin Simulation workshop, Sep. 2006) http://www.cs.uni-paderborn.de/en/research-group/research-group-computer-networks/projects/chsim.html

[28] Mellanox Technologies: InfiniBand model: http://www.omnetpp.org/filemgmt/singlefile.php?lid=133

[29] I. Dietrich, C. Sommer, F. Dressler. Simulating DYMO in OMNeT++. Erlangen-Nürnberg : Friedrich-Alexander-Universität. 2007 Internal report.

[30] Isabel Dietrich, Volker Schmitt, Falko Dressler and Reinhard German, 2007. "SYNTONY: Network Protocol Simulation based on Standard-conform UML 2 Models," Proceedings of 1st ACM International Workshop on Network Simulation Tools (NSTools 2007), Nantes, France, October 2007.

[31] I. Dietrich, C. Sommer, F. Dressler, and R. German. 2007. Automated Simulation of Communication Protocols Modeled in UML 2 with Syntony. Proceedings of GI/ITG Workshop Leistungs-, Zuverlässigkeits- und Verlässlichkeitsbewertung von Kommunikationsnetzen und verteilten Systemen (MMBnet 2007), Hamburg, Germany, September 2007. pp. 104-115.

[32] Syntony home page. http://www7.informatik.uni-erlangen.de/syntony [accessed on September, 2007]

[33] Feng Chen, Nan Wang, Reinhard German and Falko Dressler, 2008. "Performance Evaluation of IEEE 802.15.4 LR-WPAN for Industrial Applications," Proceedings of 5th IEEE/IFIP Conference on Wireless On demand Network Systems and Services (IEEE/IFIP WONS 2008), Garmisch-Partenkirchen, Germany, January 2008.

[34] A. Hennig, D. Revill and M. Pönitsch. 2003. From UML to Performance Measures - Simulative Performance Predictions of IT-Systems using the JBoss Application Server with OMNET++. Proceedings of ESS2003 conference. Siemens AG, Corporate Technology, CT SE 1.

[35] Michael, J. B., Shing, M., Miklaski, M. H., and Babbitt, J. D. 2004. Modeling and Simulation of System-of-Systems Timing Constraints with UML-RT and OMNeT++. In Proceedings of the 15th IEEE international Workshop on Rapid System Prototyping (Rsp'04) - Volume 00 (June 28 - 30, 2004). RSP. IEEE Computer Society, Washington, DC, 202-209. DOI= http://dx.doi.org/10.1109/RSP.2004.30.

[36] T. R. Henderson, S. Roy, S. Floyd, G. F. Riley. ns3 Project Goals. WNS2 ns-2: The IP Network Simulator, Pisa, Italy - Oct. 10, 2006. http://www.nsnam.org/docs/meetings/wns2/wns2-ns3.pdf

[37] Ahmed Sobeih, Wei-Peng Chen, Jennifer C. Hou, Lu-Chuan Kung, Ning Li, Hyuk Lim, Hung-Ying Tyan, and Honghai Zhang,.J-Sim: a simulation and emulation environment for wireless sensor networks. IEEE Wireless Communications Magazine, Vol. 13, No. 4, pp. 104--119, August 2006.

[38] J-SIM home page: http://www.j-sim.org [accessed on September, 2007]

[39] Cowie, J. H., Nicol, D. M., and Ogielski, A. T. 1999. Modeling the Global Internet. Computing in Science and Engg. 1, 1 (Jan. 1999), 42-50. DOI=http://dx.doi.org/10.1109/5992.743621

[40] X. Zeng, R. Bagrodia, M. Gerla.GloMoSim: a Library for Parallel Simulation of Large-scale Wireless Networks. PADS '98, May 26-29, 1998 in Banff, Alberta, Canada.

[41] Qualnet home page: http://www.qualnet.com [accessed on September, 2007]

[42] R. Barr, Z. J. Haas, R. van Renesse. 2004. JiST: Embedding Simulation Time into a Virtual Machine. Proceedings of EuroSim Congress on Modelling and Simulation, September 2004. Computer Science and Electrical Engineering, Cornell University, Ithaca NY 14853.

[43] SimSAN home page. http://simsan.storwav.com/ [accessed on September, 2007]

[44] OppBSD home page. https://projekte.tm.uka.de/trac/OppBSD [accessed on September, 2007]

[45] E. Wu, S. Woon, J. Lai and Y. A. Sekercioglu, 2005. "IPv6Suite: A Simulation Tool for Modeling Protocols of the Next Generation Internet", In Proceedings of the Third International Conference on Information Technology: Research and Education (ITRE 2005), June 2005, Taiwan.

[46] Zeigler, B. 1990. Object-oriented Simulation with Hierarchical, Modular Models. Academic Press, 1990.

[47] Chow, A and Zeigler, B. 1994. Revised DEVS: A Parallel, Hierarchical, Modular Modeling Formalism. In *Proceedings of the Winter Simulation conference 1994.*