DISCRETE EVENT SIMULATION WITH UNIVERSAL PROGRAMMING LANGUAGES ON MULTICORE PROCESSORS

Thomas Wiedemann

University of Applied Science Dresden

wiedem@informatik.htw-dresden.de

ABSTRACT

The current hardware development is characterized by a in-creasing number of multi-core processors. The performance advantages of dual and quad core processors are already applied in high speed calculations of video streams and other multimedia tasks. This paper discusses possible applications of multi-core processors in discrete simulation. The implementation of parallel threads on more than one core requires massive changes in the software structure and software module interaction. Such changes are only possible inside the source code and can not be realized in COTS-simulation systems. The paper presents a special approach by using an assembler based, very fast multitasking routine combined with an additional multi-core runtime system. The basic system approach is realized with Standard C/C++ and Delphi-compilers and offers an high flexibility and a good runtime performance.

Keywords: Multicore processors, discret event simulation with universal languages

1. INTRODUCTION

The main algorithms and mathematical foundations of simulation systems are well defined and efficient (Wiedewitsch and Heusmann 1995). Nevertheless, the real application of simulation systems is still difficult (Kuljis and Paul 2000). Not more than 10% of all industrial firms use simulation tools by a number of reasons:

·The implementation of discrete event simulation mod-els with standard programming languages like C++ or Delphi is difficult. The main problem is the parallel execution of thousand or million small processes, which represent the simulation objects. The old concept of co-routine switching is not supported by modern programming languages.

·Especially in the area of optimization with simulation models exists a performance problem. It seems like a paradox, that an older simulation language like GPSS is significantly faster than modern simulation systems at run-time.

- In many areas of production planning it takes hours or days for finding useful solutions. Any speedup would improve the quality of simulation and optimization in real use cases.
- Commercial simulation packages like AutoMOD, Enterprise Dynamics, Arena or SLX are very complex. In fact of the small market for simulation, the prices of the systems are very high. Typical prices of more than $50,000 are too high for medium-sized firms.

In summary, it seems necessary to use cheap standard programming languages with fast scheduling algorithms on multi-core processor systems for much higher simulation speed and lower investment costs.

2. INSIDE DISCRETE EVENT SIMULATION

The foundations of discrete event simulation are already 30 years old. They are based on the basic principles of simulation, which are explained in detail in other papers (see former “How it works” sessions at the WSC, e.g. (Schriber 2003), (Kilgore 2001)).

In general, modeling and simulation of real world systems require parallel execution of a large number of processes in a specific order. This task is solved by all simulation systems. It is useful to discuss some details.

Process switching is the first task of parallel execution. The executing processor must switch from one process to an other process by preserving all states for future re-switching (see fig. 1). Often there are thousands of small processes with a high switching rate. Some operations are also conditionally. Switching inside basic functions is called co-routine switching. After a first realization in SIMULA such switching technologies were not integrated in C / C++ or similar languages. Other technologies, like pointer based functions calls and multi-threading are too slow and too complicated.

Process scheduling is the second task. The sequence of process switching must be determined by the simulation control unit. This is uncritical, if the schedule is simply determined by time or priority. It is critical, if the scheduling order depends on conditions, like blocking states in sequential organized queues.

Performance problems with simulation systems are often based on bad or non adequate switching and scheduling algorithms. Using standard multitasking algorithms from C/C++ or Delphi libraries are critical, because they are designed for switching a small number of large processes like tasks in operating systems. Often, the maximum number of threads is limited and the scheduling order can not be changed by the developer.

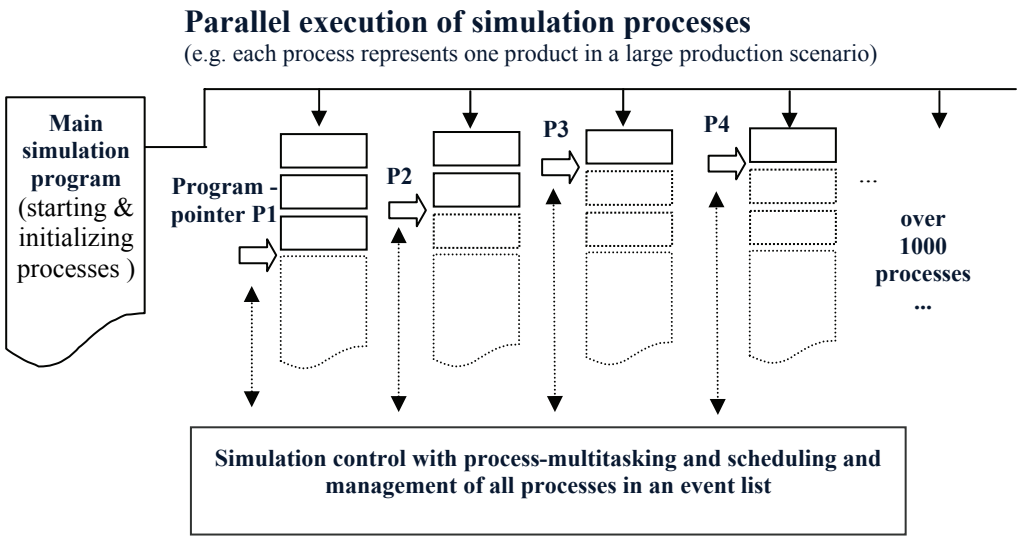

Figure 1 : Parallel execution and switching of simulation processes

Algorithms for switching and scheduling define different requirements :

- process switching is a quite simple task and defines the main performance,
- process scheduling is quite complex, and less critical in performance.

Although it seems possible to develop a very efficient switching implementation, it is nearly impossible to develop a optimal scheduling algorithm for all applications, because there are dozens of scheduling algorithms on trees, sorted lists etc., which differ in terms of performance and complexity.

From this view, a main design decision was made: The **switching should be separated from scheduling** by using an open and flexible interface, which allows the simulation model builder a free choice of possible switching and scheduling modules.

Because of the fact, that nearly all existing computers are based on sequential (non-parallel) processors, the switching will always change from the current to the next process. If the scheduler has determined the next process, the switching will need only the information of the current and next process by using the following interface (see fig. 2).

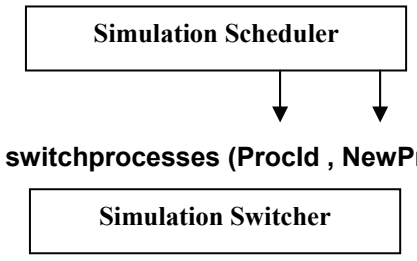

Figure 2 : Separation of switching and scheduling

This simple interface allows a wide spectrum of different switching and scheduling algorithms. The following pages will present some first implementations.

3. SWITCHING BY EXTREME MULTI-TASKING

3.1. Options for switching processes

The switching algorithm must save all local variables and the state of the processor of the current process, then he should load the new program and stack pointer address and must restore the processors register and local variables of the new process. Traditionally, the saving and restoring of the local variables is done by copying all memory blocks to backup areas, which is very time consuming.

Because of the fact, that in standard programming languages like C++ or Delphi all local variables are located on the stack, it seems possible to **switch all local data and the return address for the new process by only changing the current stack pointer address**. This simple change of the stack pointer value reduces the time for process switching significantly and allows very high rates of process multitasking. Otherwise there are some critical points of this approach:

- The change of the stack context is non trivial, because all local variables of all calling functions are switched off. In result, this method requires some special initialization of the stack during the start of each process.
- In general, the stack must provide memory space for an unknown number of functions calls. The size of stack space in standard implementations is between 16 Kbytes up to 64 Kbytes. The real used space is very different – efficient simulation functions need only some Hundred bytes of stack space, but Windows functions often require dozen Kilobytes of stack space. If any simulation process would use 64 Kilobytes of stack space, there would not be enough memory in the computer. For this reason the stack space is limited to 500 … 2000 Bytes per simulation process. If any simulation function calls an

expensive Windows function, this call is mapped to a larger stack space.

- Changing the stack pointer address could be dangerous for complex programming environments. The approach must be tested with each compiler and new version for avoiding stability problems.

In conclusion, the switching of processes by only changing the stack pointer is simple and very fast., but it has also some smaller disadvantages. For this reason, the attribute "extreme multitasking" is used to inform potential users about this specific approach.

3.2. Implementation results

The approach was tested by using DELPHI with the Object Pascal language. The stack pointer addresses are moved by assembler commands (see lines 7 – 10 of fig. 3) to and from a process address table. The push and pop commands save and restore the processor registers to the stack before switching. The number of POP/PUSH-operations depends on the specific processor and can change for other versions of compilers and languages.

```
procedure switchprocesses(OldProcId: integer;
NewProcID:integer );
begin asm  push eax // save calling environment
           push ebx
           push ecx
           push edi
           mov stackold,esp; end; // store old STACKP
      stacknew := cal[NewProcID];
      cal[OldProcId]:= stackold;
      asm mov  esp,stacknew; // get new STACKP
           pop edi
           pop ecx
           pop ebx
           pop eax  // get old environment
      end;
end; //AT THIS POINT THE SWITCHING HAPPENS !
```

Figure 3 : The code of the process switching module

Because of the fact, that there was no secure information about the possibility of changing the whole stack context by such a direct way, the author was impressed by the fact, that this code is also Debugger-safe. So if any application developer uses this code, he can still see all steps in step-wise execution: The old process enters this code sequence and after ending the switching code with the *end;* - statement (which is in practice a RETURN-assembler statement), the high level code–pointer will continue with the new process.

The necessary memory for this approach is simply the size of the stack of each process multiplied by the maximum number of processes. With a stack size of 2 Kilobytes about 500 processes are possible per Mbyte memory. If there are 100 Mbytes free memory, it allows 50.000 processes, which is a good value also for large models. If this size is too small, the simulation user should spend 100$ for an extra 1 Gigabyte RAM Memory.

In conclusion, we **PAY PERFORMANCE WITH MEMORY**, which is a cheap option today !

4. FLEXIBLE SCHEDULING

As defined by the interface (see fig. 2), the scheduler must select the next process for execution. This selection should be very fast for large numbers of processes and without long calculation times for inserting and deleting processes from the selection table. The kind of selection of course depends from the kind of simulation. In result, there will be different scheduling options for different simulation types.

Simple sequential scheduler

A simple sequential scheduler selects all processes one by one in the table and activates them. This kind of scheduler is only useful, if nearly all processes are executed in a strong periodical way. Related simulation models are used in traffic simulations, where all simulations objects (like cars or humans) are moving with small steps in every time step of simulation. The disadvantage of this scheduler is the bad performance in systems with very different activation rates.

Together with the switching module this scheduler allows a first test scenario for building up a simulation model. The resulting time for one whole cycle, measured over 1 Million switching / scheduling sequences was about 13 – 17 Nano-seconds on a 1,3 GHz Centrino PC and less than 10 Nano-seconds on a 2,5 GHz Desktop PC´s. In fact, that this time corresponds to about 30 basic assembler operations this cycle time seems to be the **lowest possible multitasking time cycle time**. Thread switching has cycle times from 500 ns up to some micro-seconds.

Future event list schedulers

For complex simulation models the sequential scheduler is not powerful enough. Better characteristics are possible with Future event list schedulers. They manage all processes in a sorted list. New processes are inserted by using their next activation time as the sort value. In result, the entry at the start of the list is always the next process for execution.

A simple list is critical for large amounts of processes, because the time for finding the place for insertion is linear growing with the number of processes. The current implementation task consists in finding algorithms with a better performance characteristic.

One option is an array-based tree with only 4 levels. In this scenario the time value is represented as a 32 bit long integer value. Each byte of this time is used as an index in one of the four levels (see fig. 5). With this approach, the insert time does not increase with a growing number of processes. The disadvantage is the same as before with the switcher – a high memory consumption. A test implementation shows, that about 3 Mbytes of RAM is necessary for running a typical production scenario.

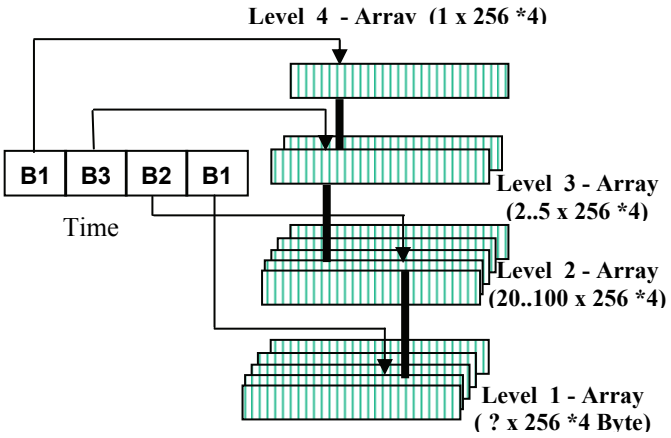

Figure 4 : An improved Future event scheduler

The main difference to existing simulation systems is the freedom of choice in the area of schedulers. While switching is assembler based and not very comfortable for High-level programmers, the development of new and much more improved scheduling algorithms is quite simple for experienced simulation kernel developers. After an initial time of building up different schedulers, the simulation user can select one of already existing schedulers. It is also possible to use different schedulers for different areas of a simulation model.

5. MULTI-CORE SUPPORT

The increasing number of multi-core processors in personal computers is very interesting also for simulation of large models, although it is not a new theme for the simulation community. Since many years distributed simulation is a well discussed topic in the simulation –community (see Perumalla 2006, PADS).
The main difference between the traditional distributed simulation and new opportunities of multi-core processors is defined by the wide availability of multi-core systems in the future :

- Instead of using specialized and very expensive hardware systems, nearly all future standard personal computers are equipped with 2,4 or more processor cores. So there is no cost overhead in hardware, when distributed simulation is used.
- Otherwise, standard computers are equipped only with standard operating system like Windows or Linux. In result, the implementation of distributed simulation must be realized with the methods of the existing operating system.

Implementation of distributed simulations on multi-core processors

The major number of multi-core systems will have two or four cores in the next few years. So the basic architecture of a distributed simulation should divide the algorithms on 2 or 4 or multiples of 2 cores.

The main experience from PADS-simulations shows, that a distributed execution of the simulation model itself is very complicated and the resulting speedup depends very heavily on the necessary communication between the distributed simulation modules. In bad cases, the speedup is below 1, which makes distributed simulation useless.

In the current situation with "only" 2 or 4 cores it seems more useful, **not to divide the model**, but to distribute the model and the simulation infrastructure. If there are more cores in the future, the cores should be used for a **pair based Hyper computing**, where one core is used for the simulation control and the other for the model. Of course also the traditional Hyper computing is possible and should be used if faster. Beside the model execution the simulation system must realize the following tasks:

- Scheduling of simulation processes with Future and Current event lists,
- Generation of a wide spectrum of random numbers (some random number types are quite expensive in terms of mathematical calculations)
- Storage of simulation results with basic statistical calculations (mean, standard deviation etc.) and compression of time series values.

On a 2-core system these tasks will be executed on the first core and the simulation model on the second core (see fig. 5). On a 4-core system the simulation control tasks will be executed on cores 1-3 and the simulation model on the fourth core (see fig. 6).

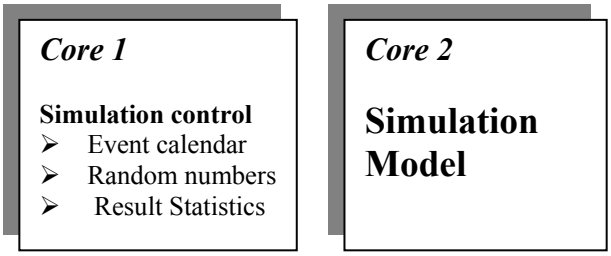

Figure 5 : 2-Core distributed simulation

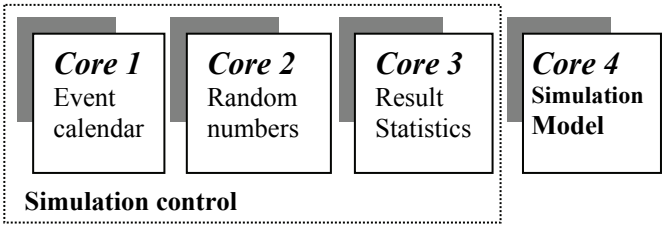

Figure 6 : 4-Core distributed simulation

The possible speedup of such an architecture depends very on the ratio between the model execution and the simulation control execution. In cases with small model functions, e.g. only random number based simulation of machining processes the time of model and simulation control execution could be nearly the same and the speedup could reach the number of existing cores 2 or 4, which means 50% or 75% less execution time.

In complex models with long running model functions the ratio between model and simulation control could be bad, so the speedup will decrease. In this case the cores should be used for a traditional Hyper computing, where each core executes one replication. In the case of Hyper computing the speedup equals nearly the number of processors. In applications where the speedup must be guaranteed, the usage of such parallel running simulations is the best and safe way. The described split of simulation control and model control seems only a interesting way of distributing simulation without dividing the models in very different and difficult ways.

Some additional measures could increase the speedup in a case with an oscillating ratio between model and simulation control:

- The generation of random numbers could be done in advance. So the next 200 or more random numbers could be generated and a model function with a burst usage (e.g. a Monte Carlo scenario inside a standard model) could use the numbers without waiting.
- The management of the event calendars is focused on delivering the next simulation events to the model. The storage of future events is of lower priority and is done after extracting the next future events from the list. A small secondary future event queue is possible.

If the simulation model runs always longer than the simulation control , some time expensive algorithms from the model (e.g. path-finding algorithms over a network or interpreting user-defined code) can be moved toward an free processor core.

In result of this options the ratio can be fine tuned towards similar time of model and simulation control execution , which maximizes the speedup. With some additional effort, this fine tuning can be automated in future systems.

First implementation results

The current simulation system is based on two (in the future also 4 or more) program threads. The first thread is started on the first processor and manages the simulation control functions. The second thread is started by the first thread and executes the simulation model. The interface between the threads is realized with shared memory.

The measurement of the speedup is quite simple : a first run is started with both threads only on one processor – which gives the single sequential time. The second run is executed with distributed threads on two cores and gives the time for distributed simulation. The first experiments with some simple queuing models with two lines of 4 machines show **speedups between 1.3 and 1.7** without special optimizations. Further work will analyze the effects of improvements of the interface and the discussed optimization measures.

6. THE SIMSOLUTION SYSTEM

All described basic routines will generate the kernel for a larger simulation environment, called "SIMSOLUTION". The whole picture of the future "SIMSOLUTION"-simulation environment shown in Figure 6 and is based on former development of the author (Wiedemann 2000, Wiedemann 2002). Above the Code-level are the GUI-interfaces or interfaces to other information systems. Possible interfaces could be traditional desktop forms or web based forms in a internet browser. The large block in the center of the system controls all processes. It is also an interfacing layer between the specific tools at the tool level and the universal and standardized modules at the Model level. The communication between all modules is based on file or network techniques. The communication protocol uses XML-coded information. In many cases the content of the XML-databases or XML-encoded simulation results is only wrapped by an additional XML-layer and transported over the network. Larger amount of data, for example simulation results, will be compressed by well-known compression algorithms for better transportation speed. For the end user this data conversions will be transparent. Data and model storage is realized with data bases, where a universal canonical data model is used for all simulation model. By using SQL-statements the elements of the model could be manipulated also group wise. This option allows quick and efficient changes of large simulation models.

7. SUMMARY

The application of a universal programming language as basic language offers new opportunities for the development of discrete event simulators.

Especially new hardware options like multi-core processors could be used without long waiting for new versions of COTS-simulation systems. The applied architecture of a distribution between simulation model and simulation control is not every time the best option, but it guaranties in opposite to traditional distributed models in any case a speedup larger than one. But if possible and useful, also the model could be distributed on future processors with more than two cores.

Well-known programming languages like C, C++ or PASCAL will reduce the learning effort and offer better flexibility than traditional simulation systems. Adding new functions or interfacing to database system or new web-based technologies like Web-services is less expensive.

An additional effect are low investment cost also in multi core environments, because the run time modules are free of charge.

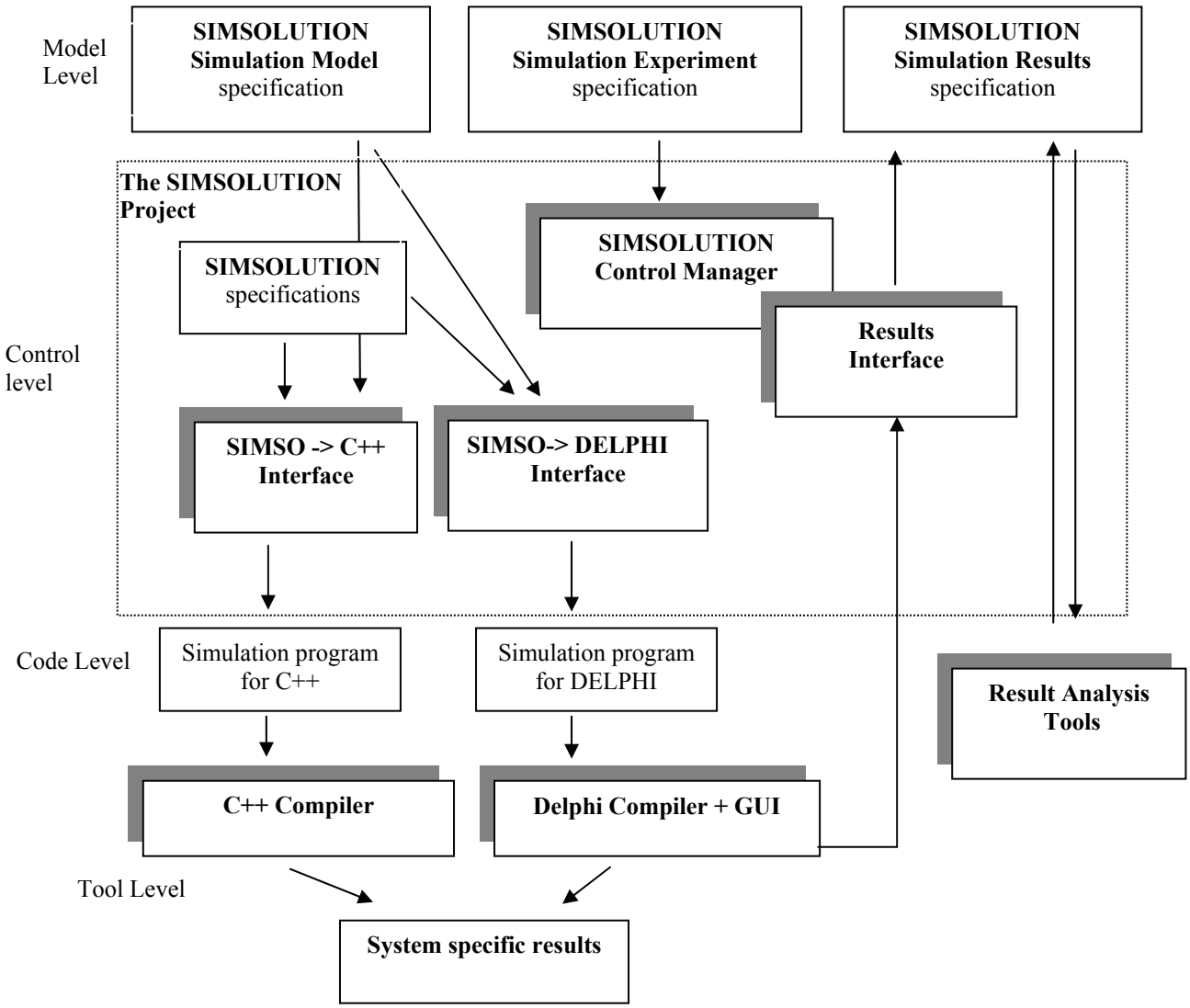

Figure 5 : The main architecture of the SIMSOLUTION - System

The usage of some specific Assembler-routines for switching could be seen as some disadvantage. But the resulting simulation speed is very high and offers new solutions especially in the area of optimization and simulation. For that reason, the current goal of development is to make the SIMSOLUTION-system some of the fastest simulation systems, even if there are some disadvantages or missing functions compared to other simulation systems.

In order to reduce the efforts for generating simulation models, the underlying programming language is managed by a universal modeling system, which generates universal, language independent XML-descriptions.

Code parsers and generators convert SIMSOLUTION-models to programs in C++, Delphi or .NET-languages.

In the future, with two sequential transformation processes a simulation model can be transferred between different platforms without manual changes.

Its future development will provide a universal and open simulation system. Any interested simulation expert or user is invited by the author for sharing his ideas, experience and cooperation inside the SIMSOLUTION-consortium.

REFERENCES

Kilgore, R. A. 2001. Open source simulation modeling language (SML*). In Proceedings of the 2001 Winter Simulation Conference*, ed., B. Peters, J. Smith. Piscataway, NJ: 2001

Kuljis, Jasna and Ray J. Paul, 2000: A Review of web based simulation: whiter we wander?, *Proceedings of the 2000 Winter Simulation Conference*, Orlando Florida, page 1872-1881

Jacobs, Peter, 2004: The DSO Simulation System. *Proceedings of the European Simulation Symposium*, Budapest, Hungary, October 2004

Perumalla K., 2006 "Parallel and Distributed Simulation: Traditional Techniques and Recent Advances". *Proceedings of Proceedings of the 2006 Winter Simulation Conference*

PADS : Website of the annual workshop for "Principles of Advanced and Distributed Simulation" http://www.pads-workshop.org/

Phillips, Lee Ann 2001. Special Edition using XML. Que Bestseller Edition, 2000

Schriber, Thomas J.; Brunner , Daniel T. : Inside Discrete-Event Simulation Software: How It Works and Why It Matters *Proceedings of the 2003 Winter Simulation Conference*, December 7-10, 2003, New Orleans, LA

Wiedemann, T., 2000. VisualSLX – an open user shell for high-performance modeling and simulation, *Proceedings of the 2000 Winter Simulation Conference*, Orlando Florida, pp. 1865-1871

Wiedemann, T., 2002. Next generation simulation environments founded on open source software and XML-based standard interfaces, *Proceedings of the 2002 Winter Simulation Conference*

Wiedewitsch J.; and Heusmann J. 1995. "Future Directions of Modeling and Simulation in the Department of Defense", *Proceedings of the SCSC'95*, Ottawa, Ontario, Canada, July 34-26, 1995

AUTHOR BIOGRAPHY

THOMAS WIEDEMANN is a professor at the Department of Computer Science at the University of Applied Science Dresden (HTWD). He has finished a study at the Technical University Sofia and a Ph.D. study at the Humboldt-University of Berlin. His research interests include simulation methodology, tools and environments in distributed simulation and manufacturing processes. His teaching areas include also intranet solutions and database applications. Email : <wiedem@informatik.htw-dresden.de>