

Profiling Simulation Performance: The Example of the German Toll System

Tommy Baumann, Bernd Pfitzinger and Thomas Jestädt

Abstract The execution performance of simulation model is an issue for large models, models used to give predictions in real-time and when the configuration space needs to be explored. These use cases apply to an existing large-scale simulation model of the German toll system which we use to identify typical workloads by profiling the run-time behavior. Performance hot spots are identified in the model and the execution environment and are related to the real-world application. In a benchmark approach we compare the observed performance to different simulation frameworks.

1 Introduction

Software evolution is a fact of life: The complexity, interconnectedness and heterogeneity of systems increases continually. Modeling and simulation techniques are one way to address the challenge of engineering, integrating and operating such systems. Especially in the design and specification stage executable models deliver tremendous value by lowering system design uncertainty and increasing specification speed and quality [1]. Hence, current system design approaches like Simulation Driven Design [2] are characterized by applying executable models to a large extend. In this context the performance in defining and executing models becomes vital due to the increased complexity of systems and processes as well as the customer requirement

T. Baumann

Andato GmbH and Co. KG, Ehrenbergstraße 11, 98693 Ilmenau, Germany
e-mail: tommy.baumann@andato.com

B. Pfitzinger (✉) · T. Jestädt

Toll Collect GmbH, Linkstraße 4, 10785 Berlin, Germany
e-mail: bernd.pfitzinger@toll-collect.de

T. Jestädt

e-mail: thomas.jestaedt@toll-collect.de

B. Pfitzinger

FOM Hochschule Für Oekonomie und Management, Zeltnerstraße 19, 90443
Nürnberg, Germany

© Springer International Publishing Switzerland 2015

M. Mach-Król et al. (eds.), *Advances in ICT for Business, Industry and Public Sector*,
Studies in Computational Intelligence 579, DOI 10.1007/978-3-319-11328-9_2

to create holistic, integrated, high accuracy models up to real world scale. Several use cases of simulations are only possible once the simulation performance is ‘good enough’: simulating the long-term dynamic behavior, iterative optimization loops, automatic test batteries, real-time models (higher reactivity to market demands and changes), and automated specification and modeling processes (including model transformation/generation) [3].

In this article we analyze the simulation performance of a large-scale Discrete Event Simulation (DES) [4] model of the German toll system implemented in MSArchitect [5]. The overall simulation performance emerges as the combination of the simulation framework (kernel) and the application domain simulation model. The outline of the article is as follows: Sect. 2 gives an overview of the automatic German toll system, the corresponding simulation model and typical simulation results. Section 3 introduces the simulation framework architecture and performance measurement techniques. Section 4 analyzes and evaluates the simulation performance of several DES kernels using small test models. Section 5 analyzes the performance within the application domain. Section 6 summarizes the results and describes future improvements and applications of our simulation model.

2 Executable Model of the German Toll System

For the application domain we use an existing simulation model of the German toll system [3, 6–8], a large-scale autonomous toll system [9] operated by Toll Collect GmbH. The tolls for heavy-goods vehicles (HGVs) driving on federal motorways—a total of 4.36 bn€ in 2012 [10]—is collected by the toll system, more than 90 % fully automatic using the more than 765,000 on-board units (OBUs) deployed in the HGVs.

In the Toll Collect example simulations are an additional method to ensure the service quality [11]. From a business perspective the simulation model is used to predict the dynamic system behavior for standard operational procedures (e.g. planned changes and updates), to determine the appropriate system sizing and to explore the system behavior close to the specification limits. Within the software development process the simulation model is used in the early design stages to implement prototypes in the context of the overall system allowing to gauge the dynamic system behavior already at the start of the software development cycle. The simulation model includes all subsystems necessary for the automatic tolling processes (Fig. 1) and for delivering updates to the OBU software, geo and tariff data as well as a model of the user interaction:

Fig. 1 High-Level simulation model of the Toll Collect system (*upper half*) and the model for the user interaction (*driving patterns*)

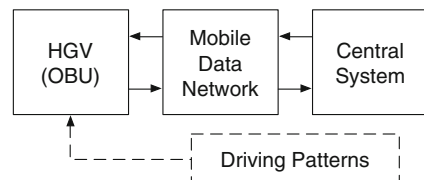


Table 1 Comparing fleet-wide updates (simulation results vs. data from April 2012 to January 2013, [13])

	Correlation
Software	0.99963
Geo data	0.99572
Tariff data	0.99475

- The vehicle fleet at a scale of 1:1 consisting of 765,000 instances of the OBU class responsible for tolling (collecting and forwarding to the central system) and for initiating the download of updates.
- The central systems necessary to receive the toll data and to serve download requests. A typical DMZ is used for authentication and accounting and the model includes a number of different resource constraints observed in the real-world system.
- The German mobile data networks provide the IP-based communication (via GPRS) between the OBUs and the central systems.
- The user interaction is modeled as statistically generated driving patterns (e.g. the points in time of HGV power cycles, toll events and mobile data network availability).

From the process perspective the simulation model covers business and system processes differing at least 7 orders of magnitude in time: All major technical processes with durations of 1 s and longer are included in the model aiming to predict the dynamic system behavior of fleet-wide updates. In fact, the model includes some processes with higher temporal resolution (down to 50 ms for the connection handling in the DMZ) and is used to simulate all updates occurring over a whole year. Using the Pearson correlation as metric to compare the simulation results with the observed update rates between April 2012 and January 2013 we find the correlation to be above (better than) 0,994 (see Table 1). The current investigation is to validate the simulation model using additional metrics and a time-scale of 1 h [13] (instead of one day).

However, a further improvement of the simulation results requires the use of an optimization algorithm to search the parameter space (an example is given in Fig. 2). A technical challenge even when using a scaled-down simulation model (1:1000): Even on the application level the user interaction (scenario generator) creates a large number of events to be processed by the simulation logic. On average each OBU will be powered-on for 16 % of the time and process tolls for 32,000 km annually ([14], one toll event per 4.2 km on average [15]) spread across some 1,300 power cycles (including three times as many periods of mobile data network). Of course, many more events are created from within the application logic.

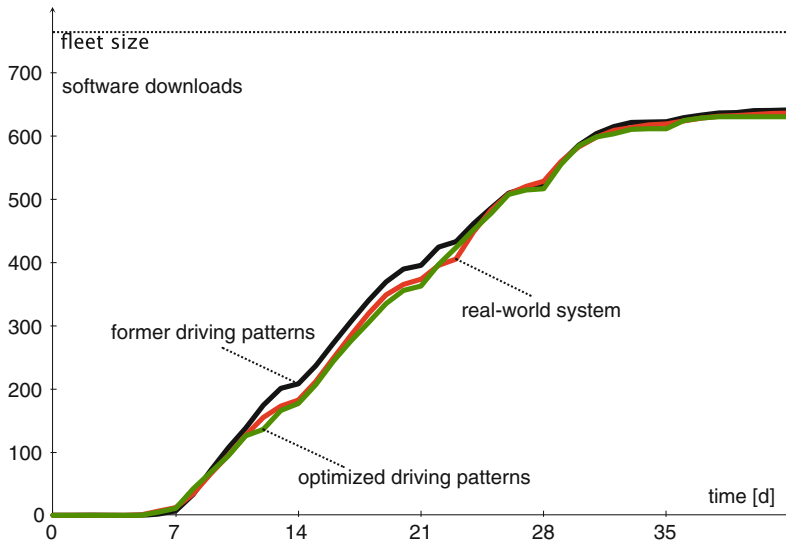


Fig. 2 Simulated software update before and after optimization [12]. The optimization algorithm is expected to improve the manually adjusted parametrization used previously (Table 1)

3 Simulator Architecture and Performance Measurement

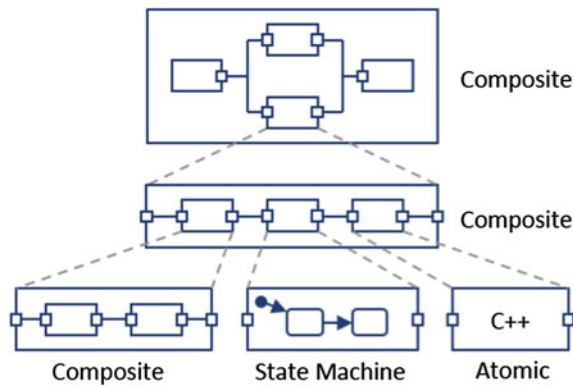
This section describes the architecture of the simulator and different possibilities to measure and evaluate simulation performance. The simulation model of the German toll system is implemented using the general purpose system design toolkit MSArchitect. The toolkit supports high performance DES and parallel DES (PDES), automated model reduction [16], hierarchical simulation models with the concept of executable simulation missions, performance probe and visualization capabilities and the generation and transformation of models during run-time.

MSArchitect distinguishes between atomic models and composite models (as most actor-oriented DES/PDES tools) to capture the behavior and structure of systems and processes. The composite models provide the structural composition of models whereas the atomics interact with the simulation kernel and are typically compiled into binary code. Combining both modeling types results in a hierarchical model tree with atomic models as leafs and composite models as nodes, as shown in Fig. 3.

The definition of atomic models and composite models depend on the application programming interface (API) to the simulation kernel. The following enumeration explains the most important components of the kernel interface [17]:

- A port represents a communication interface from or to another model entity by a logical communication channel (connection). Ports are characterized by data-type, direction, priority, and delay.

Fig. 3 Model hierarchy:
Composite models and
Atoms



- A state represents a data entity (memory) containing part of the model state during simulation. Depending on the visibility a state can be either only accessed locally or shared with other simulation entities.
- A parameter holds a data value, which is defined at modeling time and remains constant during simulation. A parameter is defined by data-type and value can be calculated from other parameters.
- A method encapsulates a user-defined functionality, which processes data or affect states. Methods are defined by signature and visibility.

Based on the kernel interface definition and the possibilities of the model description four technical representation layers can be differentiated in MSArchitect (Table 2, in principle applicable to all DES tools) with different factors impacting the performance on each layer. The *DES Composition Layer* describes the interconnection of models within a composite model and allows analyzing structural dependencies and properties [17]. On this layer, much of the performance depends on the application level behavior and its implementation as abstract simulation model. Starting with the *DES Atomic Interface Layer* the performance becomes independent of the application domain and is determined by the simulation toolset ([18], which describes the interface of atomic models as well as restrictions on features available

Table 2 Technical representation layers used in MSArchitect

Technical layer	Vocabulary
DES composition layer	Interconnection of ports, states and, parameters
DES atomic interface layer	Ports, states, parameters, lifecycle methods, inheritance, model/data types
Code layer	C++ types, variables, instructions, method calls, Control/data-flow, inheritance
Machine layer	Object code, register, memory, instructions (arithmetic, Jump, call)

for the functional description). The *Code Layer* contains the functional implementation of all atomics in the form of lifecycle methods and custom code sections. The performance is determined by the kernel event scheduling and the code generated by the compiler. The *Machine Layer* contains preprocessed and compiled code for model execution and references to external runtime libraries to be executed on a given CPU architecture.

The layers define the information available for profiling the run-time behavior, collected either by the simulation kernel, simulation logs or (external) performance profiling tools. This analysis aims first to assess the performance of simulation tools (i.e. DES Atomic Interface Layer and below). In a further analysis we take one application level example to benchmark a real-world simulation.

A kernel benchmark consists of several small test cases, which stress different aspects of a technical process. Thus they are useful to analyze low-level kernel mechanisms. The results are typically weighted according to their importance for a certain application domain. In Sect. 4 we evaluate the performance of several simulation frameworks by kernel benchmarking. Application benchmarks are characterized by use of real examples from the application domain. The selected applications should exhibit different characteristics and represent typical challenging workloads. In our case we rely on one application (the model of the German tolling system) and provide an in-depth performance analysis in Sect. 5.

4 Benchmark of Simulation Kernel Performance

In this section we perform a kernel benchmark based on test models. The models have been kept simple in order to assure universality regarding different kernels/tools and to avoid possible side effects. With regards to the technical representation layers in Table 2 the test models are defined on the *DES Composition Layer* and the *DES Atomic Interface Layer*. We included the most important performance influencing factors in our tests: The Future Event List (FEL) management, memory and data type management, pseudo-random number generator performance, and arithmetic operations performance [19]. Thereby the DES performance is characterized using the wall clock time of simulation runs, which represents the CPU time for executing a simulation model, as well as the peak memory allocated during the simulation run. Doing so, the simulation performance can be expressed as average events per second (ev/s) and memory usage. In the following we present five test models [7]. These models are applied later to investigate and compare DES kernel performance.

- **Runtime scaling:** A simulation model with several hierarchy levels is simulated with an increasing total number of events, while the size of the future event list remains fixed. This test determines whether the FEL performance is affected by the FEL size.
- **FEL Size Scaling:** The second test uses the simulation model with a delay-function. The delay is used to easily configure the (average) number of events waiting in the

future events list with minimal variance, while the total number of events processed remains constant. This test examines the overall performance of the FEL algorithm and data management. We used a uniform distribution of events in the FEL list.

- **FEL Adaption:** This test extends the future events list size during one test run by changing the parameter of the delay-function dynamically during the simulation execution. The test extends the previous test model by additional single events used to change the parameter of the delay-function. As a consequence the size of the FEL changes during the test run forcing the simulation kernel to adapt the FEL size (e.g. allocating and freeing memory) during the simulation run.
- **Data Type Management:** The test creates large data arrays of different sizes and passes the data through the simulation model in sequential or parallel order. When executing the model the memory management of the simulation kernel should recognize the passing of unmodified data and use references to this data. Ideally only one datum should be created and sent as reference through the model. As long as the delays are set to zero, no difference between serial and parallel passing should be recognizable.
- **Random Number Generation:** A large number of random values is generated using different distributions. The model uses a constant function as a reference to measure relative performance of the built-in pseudo-random-number generators.

Each test model is simulated with a set of simulation parameters using different system design tools. Currently more than 80 tools listed for DES [20]. We selected six system design tools for evaluation: Ptolemy II, Omnet++, AnyLogic, MLDesigner, SimEvents and MSArchitect. All tools were run in serial mode on an Intel Core i7 X990 at 3.47 GHz with 24 GB RAM using either Windows 7 Enterprise (64 bit) or openSuse 11.4 (32 bit, kernel 2.6.37.6). Performance data was recorded with Perfmon on Windows and sysstat and the Gnome System Monitor on the Linux system.

Figure 4 gives the results of the Runtime Scaling test. The upper chart shows the event processing performance for different simulation lengths and the bottom chart the private memory consumed during simulation. The tests show that neither the memory consumption nor the event processing performance is affected by increasing the simulation runtime. Looking at the sensitivity of running the tests with additional hierarchy levels we find that only OMNeT++ is sensitive to the additional hierarchy levels. However, from the test results it is already obvious that the different tools vary in event processing performance by an order of magnitude: MSArchitect provides the highest performance. MLDesigner, AnyLogic, and OMNeT++ provide 25 % of the speed (compared to [7] the MSArchitect performance improved by more than 30 %). Ptolemy II is twenty times slower. Looking at the memory usage during the simulation the difference between the tools is again more than an order of magnitude—the slowest tool using the most memory and the fastest tool using the least. Two of the tools (Ptolemy II and AnyLogic) are based on the Java programming language, where explicit memory deallocation is not possible. Apparently the Ptolemy II test run triggers the JVM garbage collection during the simulation run and is able to free 90 % of its memory.

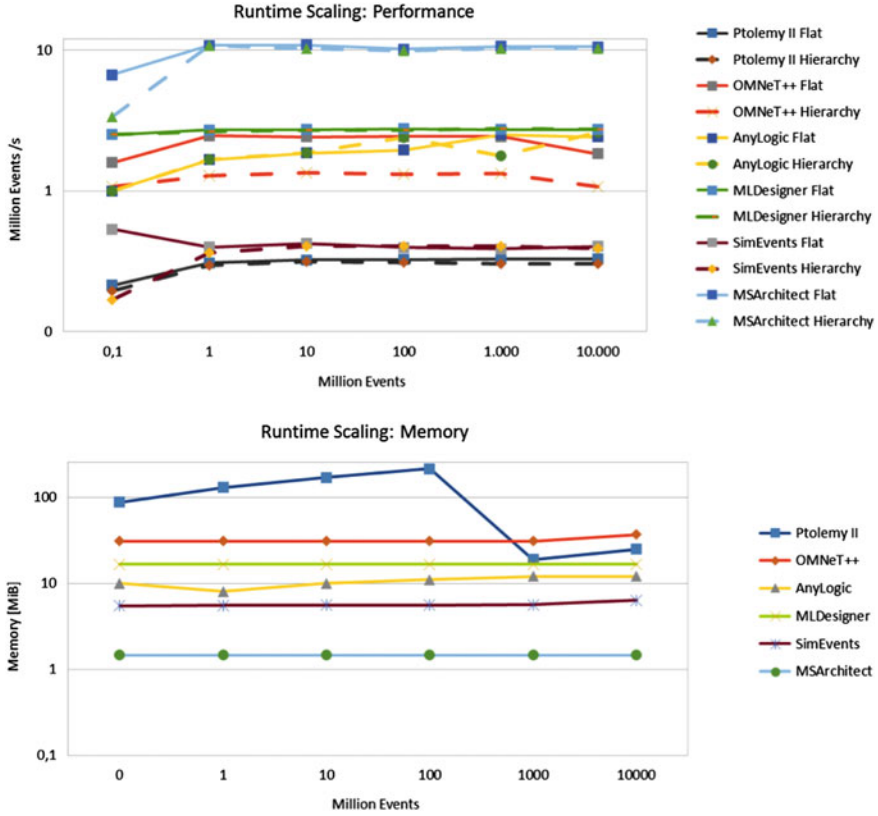


Fig. 4 Runtime performance (*top*) and memory usage (*bottom*) scaling of different DES tools

Figure 5 gives the results of the FEL Size Scaling test. As expected, a systematic performance reduction can be observed with increasing FEL size, due to the increasing overhead for FEL management. Most of the tools tested initially start with relative constant performance (on a log-log scale). With increasing FEL size three of the five tools develop drastic performance degradation. This coincides with a rapid grow of memory consumption with increasing FEL size. We propose that the performance reduction is correlated with increased FEL memory usage due to a performance penalty of calendar queue based schedulers for large queue sizes.

In the FEL Adaption test the simulation kernel is subjected to a varying demand to its FEL. Beyond the run-time needed for the test the main result is the memory consumption during the test run as given in Fig. 6. The simulation took 2,276 s with MLDesigner, 673 s with AnyLogic, 192 s with MSArchitect, 395 s with OMNeT++ and over 2 h with Ptolemy II. During that time the dynamically changing memory usage varies widely between the different tools. Most tools tend to allocate memory in chunks visible as steps in Fig. 6.

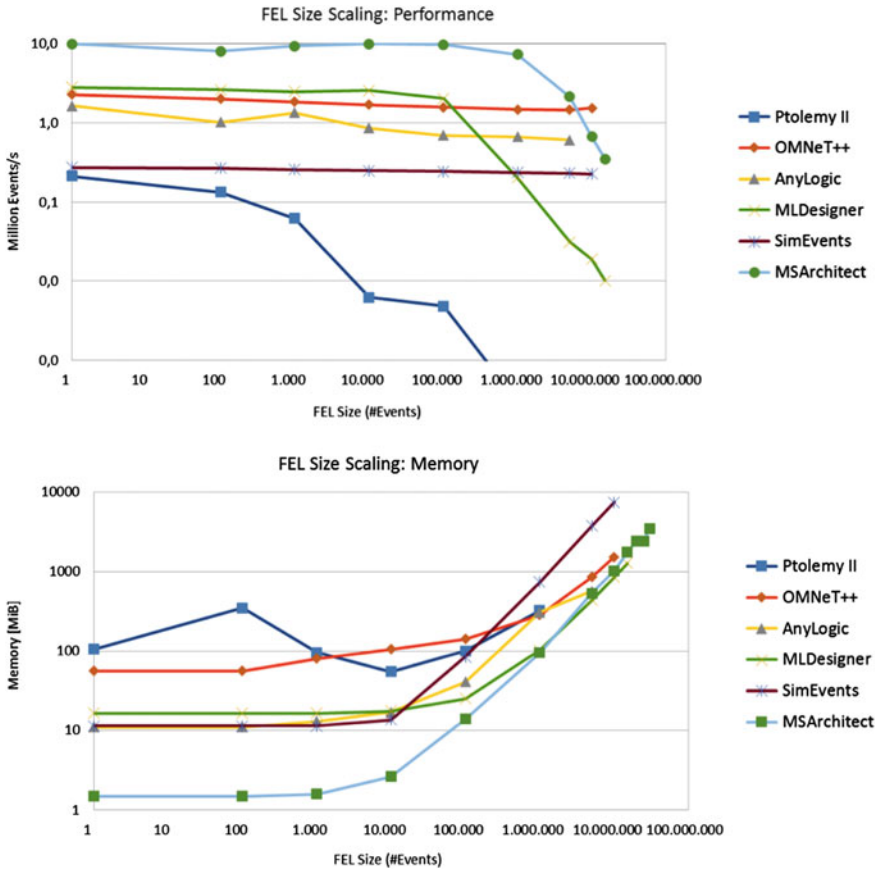


Fig. 5 Future event list size scaling test results for runtime performance (*top*) and memory usage (*bottom*) scaling of different DES tools

The Data Type Management test passes large arrays of data through the simulation running either in a parallel or serial configuration. The memory consumption during the test run is shown in Fig. 7. Most tools handle serial and parallel passing of token data in a different way, which can be recognized by the gap in memory consumption between both serial and parallel versions. Ptolemy II and MSArchitect do not show a difference between the parallel and serial version, only references are passed when only delays are used. However, Ptolemy II requires more memory and shows a different behavior according to the memory allocation: a large portion of the memory is allocated at initialization time with standard modeling elements.

The last test is the Random Number Generation test. As depicted in Fig. 8 the performance does not depend on the type of the generated distribution. Since this test relies on the correct implementation of the PRNG, i.e. we do not check the

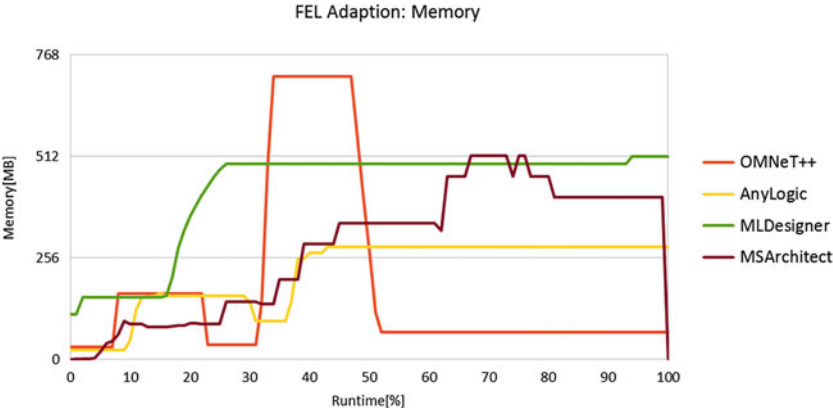


Fig. 6 Test results for future events list adaption

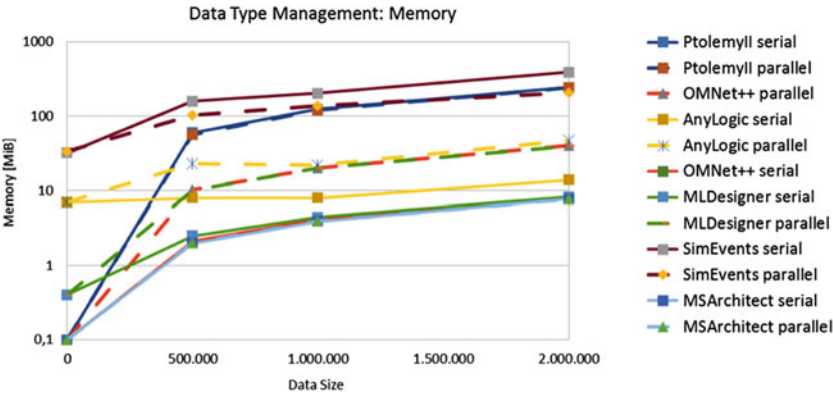


Fig. 7 Test results for the memory consumption during Data Type Management test case

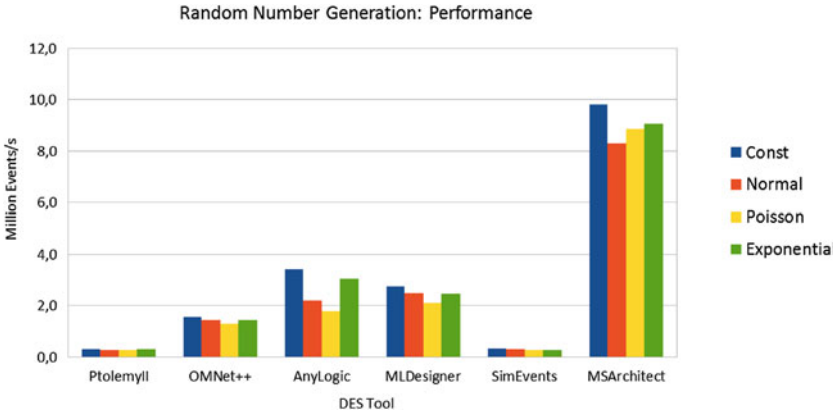


Fig. 8 Test results for random number generator scaling

statistical quality of the random numbers generated, the test results might not be fair if one of the tools were to use low-quality but high-speed generators.

It can be concluded, that Ptolemy II is the slowest tool in all simulation kernel benchmarks performed. MLDesigner is equal to or better than AnyLogic in all categories but FEL adoption. Due to the utilization of the JVM, AnyLogic requires more memory in equivalent models and therefore scales worse with increasing FEL size. Both, MSArchitect as well as OMNeT++ show the best performance in some categories. MSArchitect is the fastest simulation kernel in most categories and requires least memory for data handling.

5 Profiling of the Simulation Model

To evaluate the application-level simulation performance of our model of the German toll system, we use both the kernel logging capabilities of MSArchitect and an external profiling application (Intel VTune). Kernel logging allows to count the number of calls of atomic models as well as the total number of samples (corresponding to a processor cycle). The external profiler allows measuring the space complexity (memory), the time complexity (duration, CPU time), and the usage of particular instructions of a target program by collecting information on their execution. The most common use of a profiler is to help the user evaluate alternative implementations for program optimization. Based on their data granularity, on how profilers collect information, they are classified into event based or statistical profilers [21]. We chose the statistical profiler Intel VTune Amplifier XE and connected it to the generated C++ run-time representation of our model. As test environment, an Intel Core i7 K875 at 2.93 GHz with 8 GB RAM and Windows 7 Professional (64 bit) installed has been used.

To profile the simulation model we take the simulation scenario used to verify the simulation model against real-world data. We take statistical data to model the user interaction (driving patterns) of the vehicle fleet (700,000 OBUs used in the simulation run) to simulate the tolling system and to predict the daily rates of updates to the fleets' software, geo and tariff data. In the simulation run each OBU is treated individually and has on average 1,300 power cycles annually, the majority being very brief (i.e. less than 30 min) but some taking up to several days. The number of toll events created for the driving patterns is calibrated to correspond to the total toll amount collected in 2012 ($\approx 32,000$ km annually per OBU, using an average segment length of 4.2 km [15] and average speed of close to 80 km/h). The less-than-perfect availability of the mobile data networks is taken into account by adding (many brief) periods of network unavailability to the driving patterns. On average each power cycle has 3 such periods corresponding to almost 10 % of the power-on time (comparable to numbers seen in nation-wide network comparisons [22, 23]). In our simulation we calibrate the power-on time using the real-world update rates and choose a total power-on time of 16 % (of the whole year on average per OBU) of which 29 % are spent driving on toll roads.

Taking this scenario we use the hardware setup described above to perform the simulations with pre-calculated user interaction (driving patterns, stored as ASCII file on disk). The simulation model reads and parses the driving patterns (containing the events for power cycles, tolling and network unavailability at a granularity of 1 s) and feeds the events into the simulation model. The simulation model in turn creates many more events (with a time resolution below 1 s) to be processed by the simulation kernel. Using a single CPU core the simulation run encompassing a fleet of 700,000 OBUs and a simulated time period of 45 weeks takes less than 10h to compute.

5.1 Profiling with MSArchitect

Applying the kernel logging capabilities of MSArchitect resulting in a file with profiling information at the end of the simulation run. Table 3 shows an excerpt of the file, containing all atomic blocks relevant to analysis (15 out of 65). Since during simulation all composite blocks are resolved to directly communicating atomic blocks (see Fig. 3), the table only contains atomic blocks and is therefore related to the *DES Atomic Interface Layer*. For each atomic block the table shows the number of calls, the accumulated count of samples, the time required in relation to other atomic blocks, and the samples needed for one call.

First of all, the atomic block “AccessSessionStateSwitch” is striking, since it consumes a large amount of time due to the high number of calls. The block is

Table 3 Kernel logging results

Atomic block	Calls (M)	Samples (G)	Time (%)	Samples per call
AccessSessionStateSwitch	19,980	10,760	10,89	539
ExternDStxt	0,0007	9,565	9,68	13,665 M
StaHandling	482	6,503	6,58	13,483
EinzelbuchungsHandling	4,660	6,442	6,52	1,382
IpAutomat	7,323	5,749	5,82	785
Delay (Standard)	8,874	5,522	5,59	622
CheckComponentState	7,363	3,859	3,91	524
NetzverlustHandling	3,020	3,479	3,52	1,152
AccessSessionStateWrite	5,841	3,215	3,26	551
MfbSwitch	5,525	3,196	3,24	579
Nutzdaten	3,563	2,694	2,73	756
TcmessageCopy	2,030	2,010	2,03	990
TcpAutomat	1,291	1,533	1,55	1,187
TimedAllocate	2,570	1,484	1,50	578
SimOutObuVersions	0.017	1,509	1,53	89 M

responsible for switching OBU data structures in response to its state to one of the output ports. As the block switches between 34 states, 539 samples per call are acceptable. Nevertheless the number of calls could be reduced for performance improvement by changing the model architecture—especially once the model is ported to the parallel DES core, it is an obvious block for introducing parallelism (considering that more than 700,000 instances are running simultaneously).

The next conspicuous atomic block is “ExternDStxt”, reading the pregenerated files provided by the scenario generator model as ASCII file. The block consumes 13,665 M samples/call and is rarely executed 700 times (twice per simulated day) resulting in a time consumption of 9,681 %. In order to reduce the load, scenarios should be computed on the fly. The atomic block “StaHandling” is responsible for generating and controlling status requests, which may result in update processes. The block consumes 6,581 % of simulation time. We see potential for improvements in changing the implementation (e.g. conversion of formulas to save operations, replacing divisions by multiplications with reciprocal and using of compare functions from standard libraries). With 4,660 M calls “EinzelbuchungsHandling” is a frequently executed atomic block. After analyzing the implementation we find 1,382 samples/call acceptable. The block depends on the random number generator and would benefit from faster random number generation algorithms. The atomic block SimOutObuVersions cyclically writes the software, region, and tariff version of all OBUs to an output file. In our scenario we simulate 50 weeks and write data every 30 min, resulting in 16,801 calls. 89 M samples/call seems to be quite costly and offers room for improvement.

In summary the simulation of the scenario took 98,811,263 M calls. Of these, the model components consumed 84,51 % and the simulation kernel (logical processor) 15,49 %.

5.2 Profiling with Intel VTune

In the second step we apply the profiling application Intel VTune [24], which operates at functional level resp. Code Layer (see table 4). The external profiler catches the activities of both the simulation kernel and the simulation model (denoted as “K” or “M” in Table 4).

An excerpt of the results is shown in Table 4. For each function the CPU time in percent, the amount of needed instructions (instructions retired), the estimated instruction call count, the instructions per call on average, and the last level cache miss rate (0,01 means one out of hundred accesses takes place in memory) is shown.

Most of the CPU time is consumed by kernel functions responsible for data transport. These functions are grouped by component (resp. namespace `msa.sim.core`, denoted as “K” in the first column of Table 4), as `Port.send`, `EventManager(en|de)queueEvent` and `LogicalProcessor.mainLoopFast`. In sum the functions consume 61,1 % of the CPU time. Conspicuous is the relative high last level cache miss rate of function `EventManager.enqueueEvent` with

Table 4 VTune profiling results for simulation kernel (K) and model (M) ordered by CPU time

	Function	Time (%)	IR (G)	eCC (M)	IPC	MR (%)
K	Port.send	9,0	44	689	65	0,4
K	EventManager.enqueueEvent	7,7	21	92	237	3,2
K	LogicalProcessor.mainLoopFast	7,0	17	7	2,379	0,3
K	EventManager.dequeueEvent	6,3	104	2,517	41	1,1
K	big._mul<unsigned int>	5,0	103	2,611	40	0,3
M	StaHandling.Dice	4,8	12	11	1,097	0,1
K	EventManager.scheduleEvent	3,5	53	1,286	42	0,2
K	Any.extractToken	3,0	70	1,805	39	1,7
K	Pin.popFrontToken	2,8	49	1,234	40	0,2
K	EventManager.bucketOf	2,7	17	327	55	0,0
K	Any.operator=	2,5	54	1,403	39	0,2
K	Any.create	2,3	64	1,689	38	0,4
K	random.tr1.UniformRng.getNextV	2,2	39	961	41	0,2
K	Any.doClear	2,1	22	497	45	0,2
K	Tokenizer.nextToken	1,8	22	606	36	0,3
K	TemplatePort<Tcmessage>.re-ceiveToken	1,7	29	726	40	0,3
K	TemplateTypeInfo<EventData>.createToken	1,6	84	2,326	36	2,1
M	AccessSessionStateSwitch.run	1,5	13	287	48	0,2
M	EinzelbuchungsHandling.run	1,4	5	66	87	7,2
M	IpAutomat.run	1,4	7	103	70	3,4
K	Pin.popFront	1,3	6	89	74	0,3

Shown are the CPU instructions retired (IR), estimated call count (eCC), instructions per call (IPC) and last level cache miss rate (MR)

3,2% and the needed instructions per call of function `LogicalProcessor.mainLoopFast` with 2379. However, the number of calls depends on the dispatch of data within atomic model components, which are grouped in form of user libraries. In our model we have two user libraries: `GPRSSimulation` (`GPRSSimulation.Components.Atomics`, denoted as “M” in the first column of Table 4) and `Standard` (`msa.Standard.Control`). The latter is a support library included in `MSArchitect`. Combined they are responsible for 20,1% of CPU time consumption. Performance critical and starting point for improvement is function `StaHandling.Dice` with 1,097 instructions per call and a CPU time consumption of 4,80%. This function is part of the update process where each OBU determines randomly (hence ‘Dice’) whether to check for any available updates.

Both, kernel logging and profiling showed that most of the resources are utilized by functions responsible for data input/output (data mining), and functions responsible for transmission and processing of tolling information. By doing the analysis we located multiple components with potential for optimization, e.g. `AccessSessionStateSwitch` and `StaHandling`. Furthermore we came to the conclusion to generate scenarios on the fly since the reading of pre-generated

scenario files is as time consuming. Relating the resource utilization of model components to real-world applications we could recognize a weak correlation. Model components like `STAHandling`, `IPAutomat` and `Einzelbuchungs-Handling` are abstractions of important real word system applications and crucial to performance in both worlds.

6 Summary

Extending [7, 8] we have shown how to analyze the performance of DES simulations: Generic benchmark test-cases allow a simple and direct comparison of different simulation tools. Not surprisingly the tools differ vastly as to their time and memory consumption. Yet the real-world impact depends on the application: The workload generated in typical simulation runs and the necessity of high-performance simulations.

Particularly in real-time scenarios and optimization runs the use of large-scale simulations depends critically on the performance achieved. We have shown the benefit of profiling to identify bottlenecks and to choose the appropriate tool. `MSArchitect`, the simulation kernel used in the application benchmark, is currently extended to allow the automatic model reduction and (semi-) automatic parallelization of simulation runs. The single-core benchmark performed here will be the baseline to measure the improvements against.

References

1. Baumann, T.: Automatisierung der frühen Entwurfsphasen verteilter Systeme. Südwest-deutscher Verlag für Hochschulschriften, Saarbrücken (2009)
2. Baumann, T.: Simulation-driven design of distributed systems. SAE Technical Paper, no. 2011-01-0458 (2011)
3. Pfitzinger, B., Baumann, T., Jestädt, T.: Network resource usage of the german toll system: Lessons from a realistic simulation model. In: 46th Hawaii International Conference on System Sciences (HICSS), pp. 5115–5122 (2013)
4. Lee, V., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.* **C 36**(1), 24–35 (1987)
5. Andato GmbH & Co. KG: `MSArchitect`. <http://www.andato.com/>. Accessed 12 May 2013
6. Baumann, T., Pfitzinger, B., Jestädt, T.: Simulation driven development of the German toll system—simulation performance at the kernel and application level. *Advances in Business ICT*, pp. 1–25. Springer, Heidelberg (2014)
7. Baumann, T.: Simulation driven design of the German toll system—evaluation and enhancement of simulation performance. In: 2012 Federated Conference on Computer Science and Information Systems (FedCSIS). IEEE, pp. 901–909 (2012)
8. Baumann, T.: Simulation driven design of the German toll system—profiling simulation performance. In: 2013 Federated Conference on Computer Science and Information Systems (FedCSIS). IEEE, pp. 923–926 (2013)
9. CEN: ISO/TS 17575-1:2010 electronic fee collection—application interface definition for autonomous systems—part 1: Charging (2010)

10. Bundesministerium der Finanzen: Sollbericht 2013, Monatsbericht des BMF, 2, pp. 6–22, Feb. 2013. http://www.bundesfinanzministerium.de/Content/DE/Monatsberichte/2013/02/Downloads/monatsbericht_2013_02_deutsch.pdf?_blob=publicationFile&v=4. Accessed 20 Mar 2013
11. Pfitzinger, B., Jestädt, T.: Service governance from the perspective of service quality: Notes from the German toll system. In: *Smart Services and Service Science*, vol. 36, pp. 141–150 (2012)
12. Baumann, T., Pfitzinger, B., Macos, D., Jestädt, T.: Using parameter optimization to calibrate a model of user interaction. In: *2014 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE (2014)
13. Pfitzinger, B., Jestädt, T., Baumann, T.: Simulating the German toll system: Choosing “good enough” driving patterns. In: *Proceedings of the mobil.TUM 2013 - International Conference on Mobility and Transport* (2013)
14. Bundesamt für Güterverkehr. Mautstatistik Jahrestabellen 2011, Bonn, Feb. 2012. http://www.bag.bund.de/SharedDocs/Downloads/DE/Statistik/Lkw-Maut/Jahrestab_11_10.pdf?_blob=publicationFile. Accessed 10 Mar 2012
15. Dettmar, M., Rottinger, F., Jestädt, T.: Achieving excellence in GNSS based tolling using the example of the German HGV tolling system. *ITS Europe*, Dublin (2013)
16. Pacholik, A., Baumann, T., Fengler, W., Rath, M.: A model reduction approach for improving discrete event simulation performance. In: *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Brussels, Belgium, pp. 101–108 (2013)
17. Zhou, Y., Lee, E.A.: Causality interfaces for actor networks. *ACM Trans. Embed. Comput. Syst. (TECS)* 7(3), 29 (2008)
18. Pacholik, A., Baumann, T., Fengler, W., Grüner, D.: Discrete event simulation performance—benchmarking simulators. In: *International Simulation Multi-Conference (SummerSim)*, Genoa, Italy (2012)
19. Fishman, G.S.: *Discrete-Event Simulation: Modeling, Programming and Analysis*. Springer, Berlin (2001)
20. Albrecht, M.C.: Introduction to discrete event simulation. <http://www.albrechts.com/mike/DES/Introduction> (2010)
21. Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: a call graph execution profiler. In: *ACM Sigplan Notices—Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, vol. 17, no. 6, pp. 120–126, ACM (1982)
22. Theiss, B.: *Der große Netztest in Deutschland* (2010)
23. Theiss, B.: *Mobilfunk in Deutschland—Der Netztest 2012* (2012)
24. Intel: Intel VTune amplifier. <http://software.intel.com/en-us/intel-vtune-amplifier-xe>. Accessed 12 May 2013

Advances in ICT for Business, Industry and Public Sector

Mach-Król, M.; M. Olszak, C.; Pełech-Pilichowski, T. (Eds.)

2015, VIII, 189 p. 76 illus., 9 illus. in color., Hardcover

ISBN: 978-3-319-11327-2