Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 923–926

Simulation driven design of the German toll system – profiling simulation performance

Tommy Baumann*, Bernd Pfitzinger†‡ , Thomas Jestädt†
*Andato GmbH & Co. KG, Ehrenbergstraße 11, 98693 Ilmenau, Germany. Email: tommy.baumann@andato.com
†Toll Collect GmbH, Linkstraße 4, 10785 Berlin, Germany. Email: {bernd.pfitzinger|thomas.jestaedt}@toll-collect.de
‡FOM Hochschule für Oekonomie & Management, Bismarckstraße 107, 10625 Berlin, Germany

***Abstract*—Taking an existing large-scale simulation model of the German toll system we identify the typical workload by profiling the runtime behavior. Crucial performance hot spots are identified and related to the real-world application to analyze and evaluate the observed efficiency. In a benchmark approach we compare the observed performance to different simulation frameworks.**

I. Introduction

As technology advances, systems and processes with higher complexity, interconnectedness and heterogeneity can be developed. Simultaneously, user requirements are constantly increasing: Software evolution is a fact of life. Modeling and simulation techniques are applied to design, analyze, evaluate, validate, and optimize such systems. The article analyzes the performance of a large-scale Discrete Event Simulation (DES, [1]) simulation model of the German toll system implemented in MSArchitect [2] – similar to and larger than existing simulation models [3, 4].

The next section gives an overview of the automatic German toll system, the corresponding simulation model and typical simulation results. Section III introduces the simulation framework architecture and performance measurement techniques. Section IV analyzes and evaluates the simulation performance of several DES kernels using small test models. Section V analyses the performance within the application domain followed by a summary in section VI.

II. Executable model of the German toll system

For the application domain we use an existing simulation model of the German toll system [5–7], a large-scale autonomous toll system [8] operated by Toll Collect GmbH. The tolls for heavy-goods vehicles (HGVs) driving on federal motorways – a total of 4.36 bn € in 2012 [9] – is collected by the toll system, more than 90% fully automatic using the more than 750 000 on-board units (OBUs) deployed in the HGVs. 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 [10].

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 one second and longer are included in the model aiming to predict the dynamic system behavior of fleet-wide

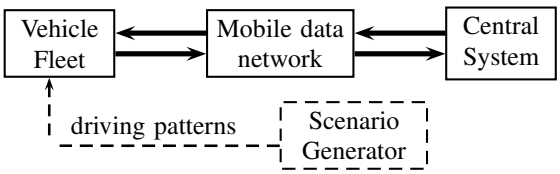

Fig. 1. High-Level simulation model of the Toll Collect system (upper half) and the model for the user interaction (scenario generator, lower half).

updates (taking weeks to months). In fact, the model includes some processes with higher temporal resolution (down to 50 ms for the connection handling by the firewalls). Using the Pearson correlation as metric to compare the simulation results with the observed update rates between 04/2012 and 01/2013 we find the correlation to be above (better than) 0,994.

Even on the application level the user interaction (precalculated driving patterns) 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 ([11], one toll event per 4.2 km on average [12]) spread across 1 300 power cycles (including three times as many periods of loss of access to the mobile data network). Of course, many more events are created from within the application logic, e.g. to forward tolls to the central systems or to run error recovery protocols in the case of network unavailability.

III. Simulator architecture and performance measurement

This section describes the architecture of the simulator and different possibilities to measure and evaluate simulation performance. 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. Atomics interact with the simulation kernel and contain the whole behavioral description, including event consumption and creation, time advance of events, and manipulation of data entities. They are typically written in C++ (all common programming languages work as well) and compiled into binary code for execution. The composite models provide the structural composition of models, but do not themselves contribute to the execution semantics. Combining both modeling types results in a hierarchical model tree with atomic models as leafs and composite models as nodes, as shown in fig. 2.

978-1-4673-4471-5/$25.00

TABLE I
TECHNICAL REPRESENTATION LAYERS USED IN MSARCHITECT

Technical layer	Vocabulary	Performance factors
DES Composition	interconnection of ports, states and, parameters	model hierarchy, summable model structure, data exchange via ports, forks and merges, sharing of states, port multiplicities
DES Atomic Interface	ports, states, parameters, lifecycle methods, inheritance, model/data types	data type conversion, reuse of data objects, memory organization
Code	C++ types, variables, instructions, method calls, control/data-flow, inheritance	kernel event scheduling, C++ type resolution, utilization of external code, memory allocation, caching, code granularity/distribution
Machine	Object code, register, memory, instructions (arithmetic, jump, call)	CPU capabilities (instruction set, latencies, cache), memory, code structure

The definition of atomic models and composite models relies on a well-defined application programming interface (API) to the simulation kernel: Typically consisting of ports (communication interfaces), states, parameters and methods [13].

Based on the simulation kernel API definition and the possibilities for model description four technical representation layers can be differentiated in MSArchitect (tab. I, 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 [13]. 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 ([14], which describes the interface of atomic models as well as restrictions on features available 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 *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 runtime behavior, collected either by the simulation kernel, simulation logs or (external) performance profiling tools. This analysis repeats the kernel benchmarks presented in [7] and expands the performance analysis to the application level taking the example of a real-world simulation in section V.

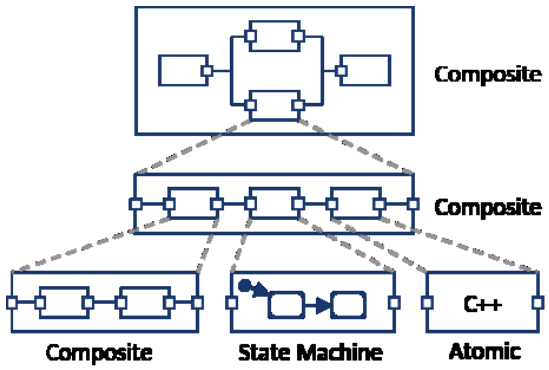

Fig. 2. Model hierarchy: Composite models and Atomics.

IV. BENCHMARK OF SIMULATION KERNEL PERFORMANCE

In this section we perform a kernel benchmark using the five test models introduced in [7]. With regards to the technical representation layers in tab. I 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 [15].

Each test model is simulated with a set of simulation parameters using different system design tools. 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 GiByte RAM using either Windows 7 Enterprise (64 bit) or openSuse 11.4 (32 bit, kernel 2.6.37.6).

Fig. 3 gives the results of the event processing performance of the Runtime Scaling test. The tests show that neither the event processing performance nor the memory consumption is affected by increasing the simulation runtime and 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 speed. 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

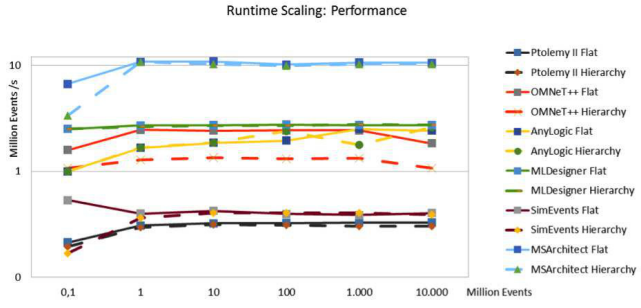

Fig. 3. Runtime performance (top) and memory usage (bottom) scaling of different DES tools.

of magnitude – the slowest tool using the most memory and the fastest tool using the least.

Repeating the FEL Size Scaling test we observe – as expected – a systematic performance reduction with increasing FEL size, due to the increasing overhead for FEL management. With increasing FEL size three of the five tools develop drastic performance degradation coinciding with a rapid grow of memory consumption. In absolute numbers, MSArchitect has the best test performance and the lowest memory usage until FEL size 10^6. Subsequently the memory usage of MLDesigner is lower since MSArchitect runs in 64 bit mode only. However, in our tests MLDesigner stopped to work for FEL sizes above $15 \cdot 10^7$. We tested MSArchitect successfully with a FEL size of 10^8.

The comparison of the DES tools using the FEL Adaption, Data Type Management and Random Number Generation tests yield the same findings as presented previously ([7], with some improvements in MSArchitects' RNG performance): Both, MSArchitect and OMNeT++ show the best performance in some categories. MSArchitect is the fastest simulation kernel in most categories and requires least memory for data handling.

V. 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 time and space complexity as well as chip-level details on the instruction execution.

To profile the simulation model we take the simulation scenario used to verify the simulation model of ther automatic German toll system against real-world data (see section II) using pre-calculated user interaction. The simulation model reads and parses the driving patterns (containing the events for power cycles, tolling and network unavailability) and feeds the events into the simulation model. The simulation model in turn creates many more events (e.g. for scheduling timeouts) 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 10 hours to compute.

As mentioned above, in a first step we apply the kernel logging capabilities of MSArchitect resulting in a file with profiling information at the end of the simulation run. Tab. II shows an excerpt of the file, containing the top-10 (out of 65) atomic blocks by runtime.

First of all, the atomic block "AccessSessionStateSwitch" is striking, consuming a large amount of time with a high number of calls. The block is 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.

TABLE II
Kernel logging results: Top 10 atomic blocks by runtime.

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

The next conspicuous atomic block is "ExternDStxt", reading the pre-generated files provided by the scenario generator model as ASCII file. The block consumes 9,681% of the runtime for 13 665 M samples/call and is rarely executed (twice per simulated day). 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 minutes, resulting in 16801 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%.

In the second step we apply the Intel VTune [16] profiler, which operates at functional level resp. Code Layer (see table III). The external profiler catches the activities of both the simulation kernel and the simulation model (denoted as "K" or "M" in tab. III).

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 tab. III), as Port.send, EventManager.enqueueEvent, LogicalProccesor.mainLoopFast, and EventManager.dequeueEvent. 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 3,2% and the needed instructions per call of function LogicalProcessor.mainLoopFast with 2 379. However, the

TABLE III
VTUNE PROFILING RESULTS FOR SIMULATION KERNEL (K) AND MODEL (M): TOP 10 FUNCTIONS BY RUNTIME

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

	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

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 tab. III) 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%.

Both, kernel logging and profiling showed that most of the resources are utilized by functions responsible for data input/output and functions responsible for transmission and processing of tolling information. Relating the resource utilization of model components to real-word applications we could recognize a weak correlation: The application-level performance is determined by the real-world behavior of the simulated system.

VI. SUMMARY

Extending [7] 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. However, the benchmark results cannot be transferred to the application domain: The workload generated by a given simulation model determines in large part its performance. Taking an existing simulation model of a large-scale technical system we performed an in-depth performance analysis for one simulation tool using both the performance analysis methods provided by the simulation kernel and an external profiler with access to the CPU hardware profiling support.

Both profilers immediately identify the same bottleneck: Reading the ASCII-formatted pre-calculated driving patterns from disk. Consequently the simulation model is now integrated with the scenario generator. This in turn will allow implementing an optimization algorithm to fit the driving patterns to the observed system behavior – a feature that we expect to drastically improve the accuracy of the simulation results for the short-term behavior [10].

The hardware profiler catches both the application-level methods as well as the atomics provided by the simulation kernel (with or without access to its source code). Looking e.g. at the cache miss rate we find some simulation kernel routines and several application-level methods with a considerable probability of needing access to the main memory. We take this as starting point for future improvements.

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] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. 100, no. 1, pp. 24–35, 1987.

[2] Andato GmbH & Co. KG, "MSArchitect," [accessed 12-May-2013]. [Online]. Available: http://www.andato.com/

[3] K. Lunde and L. Kieble, "Simulating communication within a satellite-based automated toll collection system," *Proceedings of the 55th International Scientific Colloquium*, pp. 318 – 323, 2010.

[4] K. Lunde, L. Kieble, and M.-A. Funk, "Prediction strategies in a service level granting prefetching cache for version-controlled gis data," *ISAST Transactions on Computers and Intelligent Systems*, vol. 2, no. 2, pp. 46–51, 2010.

[5] B. Pfitzinger, T. Baumann, and T. Jestädt, "Analysis and evaluation of the german toll system using a holistic executable specification," *45th Hawaii International Conference on System Sciences (HICSS)*, vol. 0, pp. 5632–5638, 2012.

[6] ——, "Network resource usage of the german toll system: Lessons from a realistic simulation model," in *46th Hawaii International Conference on System Sciences (HICSS)*. IEEE, 2013, pp. 5115–5122.

[7] T. Baumann, B. Pfitzinger, and T. Jestädt, "Simulation driven design of the German toll system – evaluation and enhancement of simulation performance," in *Computer Science and Information Systems (FedCSIS), 2012 Federated Conference on*. IEEE, 2012, pp. 901–909.

[8] CEN , "ISO/TS 17575-1:2010 electronic fee collection - application interface definition for autonomous systems - part 1: Charging," 2010.

[9] Bundesministerium der Finanzen, "Sollbericht 2013," *Monatsbericht des BMF*, vol. 2, pp. 6–57, Feb. 2013. [Online]. Available: http://www.bundesfinanzministerium.de/Content/DE/Monatsberichte/2013/02/Downloads/monatsbericht_2013_02_deutsch.pdf?__blob=publicationFile&v=4

[10] B. Pfitzinger, T. Jestädt, and T. Baumann, "Simulating the German toll system: Choosing 'good enough' driving patterns," in *Proceedings of the mobil.TUM 2013 – International conference on mobility and transport*, Lehrstuhl für Verkehrstechnik, Ed. Technische Universität München, 2013.

[11] Bundesamt fùr Gùterverkehr, "Maut-Jahresstatistik 2011/2010," 2012, [accessed 10-March-2012]. [Online]. Available: http://www.bag.bund.de/SharedDocs/Downloads/DE/Statistik/Lkw-Maut/Jahrestab_11_10.pdf?__blob=publicationFile

[12] M. Dettmar, F. Rottinger, and T. Jestädt, "Achieving excellence in GNSS based tolling using the example of the german HGV tolling system," in *Proceedings of the 9th ITS Europe Congress*, Jun. 2013.

[13] Y. Zhou and E. A. Lee, "Causality interfaces for actor networks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 29, 2008.

[14] A. Pacholik, T. Baumann, W. Fengler, and D. Grüner, "Discrete event simulation performance – benchmarking simulators," in *International Simulation Multi-Conference (SummerSim)*, Genoa, Italy, 2012.

[15] G. S. Fishman, *Discrete-Event Simulation: Modeling, Programming and Analysis*. Berlin: Springer, 2001.

[16] Intel, "Intel VTune Amplifier," [accessed 12-May-2013]. [Online]. Available: http://software.intel.com/en-us/intel-vtune-amplifier-xe