Continuous Constant-Memory Monitoring of Embedded Software Timing

Johan Kraft and Thomas Nolte
School of Innovation, Design and Technology
Mälardalen University
Västerås, Sweden
{johan.kraft, thomas.nolte}@mdh.se

***Abstract*—A method is presented for generating statistical models of timing data continuously over very long monitoring sessions. This method is intended for memory-efficient runtime modeling of timing properties in embedded software systems, such as execution times or inter-arrival times, but is a quite generic method that should be applicable for other purposes and domains as well. Specifically, we intend to use this method as a component in automatic generation of simulation models for probabilistic timing analysis of complex embedded software systems. Given a stream of data as input, this method gradually builds up a statistical model capturing the approximate distribution of the data. The method uses a modest and fixed amount of on-target RAM, decided by the desired accuracy of the model, and allows for long monitoring sessions covering billions of data points. The paper presents the motivation, algorithm, a prototype implementation and evaluation using real execution time data from an ARM7 microcontroller.**

I. Introduction

If developers of embedded software systems were able to predict runtime properties related to timing and resource usage before implementing new software designs, they could identify and avoid unsuitable software designs at an early stage and thereby also avoiding the associated costs and delays as a result of having to redesign the software when the system verification (hopefully) detects the problems. Predicting task response times is possible for many systems using established methods for schedulability and response time analysis [1], [2], [3] or tools for formal verification using model checking, such as UPPAAL [4] and KRONOS [5]. However, there are embedded systems in industry today which have real-time and performance requirements, but hardly can be analyzed using such methods. An example is the industrial robot control system IRC 5 developed by ABB Robotics which contains some 3 million lines of code and 50-100 tasks. Model checking tools do not come near to scaling to systems of this size. The existing analytical methods for schedulability or response-time analysis are not suitable for the IRC 5 system, due to the complex runtime behavior which does not comply with the system assumptions required to use these methods. In the IRC 5 system, tasks trigger each other using asynchronous messages and the contents of the messages often have a major impact on the temporal behavior of the receiver. The telecom domain poses similar even greater analysis challenges, for instance the radio base stations and radio network controllers developed by Ericsson. Such systems are event triggered, massively parallel, contain many millions lines of code, thousands of processes and also has several layer of fault tolerance which together gives very high system complexity. An alternative for analysis of complex embedded systems is to use simulation-based timing analysis [6]. This approach is more related to testing than formal verification as it is only possible to show the presence of potential errors, not their absence. However, simulation has the advantage of not posing restrictions regarding the design of the software system. Moreover, simulation allows for analysis of any measurable run-time property, in contrast to the analytical methods for schedulability and response time analysis [1], [2], [3] which have strict assumptions and focus on specific properties. Since a simulation model can be much more abstract than the modeled system, and since a PC is typically much faster than embedded hardware, many thousands of simulations can often be performed in short time where each simulation is given more or less random variations in execution times and other parameters. This way, many scenarios are explored which otherwise might be hard to generate on a real system. Note that the type of simulation in focus is not cycle-accurate low-level simulation like, e.g., Simics from Wind River, which are much slower than normal execution. In the perspective of analyzing an existing software system, simulation does not require manual modeling, since a simulation model typically is implemented using common programming languages and can be automatically extracted from the system implementation, either using a combination of program analysis and runtime measurements [6] or using runtime measurements [7] alone. This paper focuses on a specific problem in enabling automatic model generation: how to obtain execution time data for simulation models from recordings, without storing each data point individually. This is specifically in the context of the RTSSim simulation framework presented in Section II. The specific contribution of this paper is the proposed algorithm and a brief evaluation on real timing data, with diagrams visualizing the results. The algorithm has been implemented in an offline trace visualization tool, which feeds the algorithm one data point at a time, like when used in runtime monitoring. A proper implementation on an embedded system is planned in future work. As the session length is only limited by counter overflows, for 32-bit CPUs a session may cover at least $2^{32}-1$ (4.294.967.295) observations of each specific property, since each interval may hold up to $2^{32} - 1$ data points. If assuming a data production rate of 100 Hz per measured property, this solution allows for at least 30 years of continous monitoring before wrapping occurs. This is in practice unlimited.

This solution is presented in Section III, and we present an implementation and evaluation of the method in Section IV. Section V puts this paper in perspective of related work, and Section VI concludes the paper and outlines future work.

II. RTSSim and Tracealyzer

RTSSim [6] is a simulation environment which emulates a generic real-time operating system on a PC. It works as a "sandbox" with respect to timing. All time-triggered events in RTSSim are driven by an integer simulation clock incremented by explicit *Execute* calls, using timing data recorded from the modeled system. The timing of the modeled system is thereby preserved in the simulation even though it runs on a PC instead of the embedded hardware. The simulated timing is however approximate, it is not guaranteed to be 100 % identical to the real timing when executing the code on the intended hardware, since the simulation model abstracts from the details of instruction-level timing. It instead describes the execution times between relevant program points, in a probabilistic manner. RTSSim focuses on the timing and resource usage of tasks, i.e., threads in a real-time operating system. An RTSSim task is a C function which is registered in the RTSSim scheduler together with attributes such as priority. An RTSSim task contains at least one *Execute* statement, which models CPU time usage. A simulation model can be specified at different abstraction levels. In one extreme is the most basic RTSSim task containing a single Execute statement only, i.e., only execution time is specified, no behavior. In the other extreme is to include the full source code of the analyzed software system, extended with Execute statements. Our intention of the RTSSim framework is however to use it together with a *model extraction* tool, which produces an abstraction of the implementation focusing on behavior of relevance for timing and resource usage. The tasks of an RTSSim simulation model are by default scheduled using fixed priority preemptive scheduling, and are assumed to share a single CPU core. In future work we intend to extend RTSSim to support parallel systems as well. RTSSim can produce two kinds of output, a detailed simulation trace focusing on the scheduling and other logged events, and a text file with selected statistics on task timing, such as highest response time observed for the selected task. The simulation trace is produced using a *trace recorder* described in Kraft et al. [8], which outputs a simulation trace for the *Tracealyzer* tool, including task scheduling, task communication and synchronization events. Tracealyzer is trace visualization tool for studying scheduling, resource usage and interaction of tasks and interrupts in embedded software. This tool originated in a research project back in 2004, performed in collaboration with ABB Robotics. They have used it since then in every industrial robot, and the Tracealyzer is now product in the author Johan Kraft's company Percepio AB[1]. Tracealyzer is today sold by Quadros Systems, Inc. as RTXCview and is the official trace tool for the real-time operating system RTXC Quadros. A description of the trace recorder used in RTXC Quadros is found in Kraft et al [8].

III. Statistical Runtime Modeling

Classic theoretical distributions are often hard to fit to timing data from software systems, since the distributions often are discontinuous and complex. Even linear code may result in discontinuities due to hardware effects such as cache misses, and if the measurements cover code with conditional jumps that is another cause of discontinuities in the resulting distribution. The most exact method of logging timing data from a program is to store every observation separately, i.e., an empirical distribution. However, over a long monitoring session this would consume vast amounts of RAM. Another approach is to only store watermarks (lowest and highest value) and use them as bound for a uniform distribution, but this way the shape of the distribution is lost which may result in incorrect simulations.

We suggest to use an established statistical approach known as *stratification*, where the data points are divided into intervals and where each such interval has a counter attribute, which is incremented at each matching data point. The individual observations are attributed to an interval (or used to create a new interval), and then forgotten. Each interval represents a uniform distribution, so the shape of the distribution within each interval still is lost, but given enough intervals this may give a sufficiently accurate statistical model, using a modest amount of memory.

The amount of RAM needed for a particular measurement is $3wi$, where i is the number of intervals used and w is the width of the variables, measured in bytes. In many cases $w = 2$ is sufficient, otherwise $w = 4$ allows for over four billion observations per interval. The constant (3) corresponds to the number of properties stored per interval, i.e., lower bound, upper bound and count. When using this simulation model for replay of timing data in a simulation, a two step sampling approach is used. At first the interval is selected, with the probability indicated in its count property, i.e., the number of observations in the interval divided by the total number of intervals. Once the interval has been selected, its bounds are used as a uniform distribution from which the final value is sampled.

A. Modeling Algorithm

The proposed algorithm takes as input a sequence of integer values, and produces as output a set of intervals, where each interval I has:

- $I.min$: the lower interval bound
- $I.max$: the upper interval bound, and
- $I.count$: the number of inputs x observed matching the interval, i.e., where $I.min \leq x \leq I.max$.

[1]http://www.percepio.se

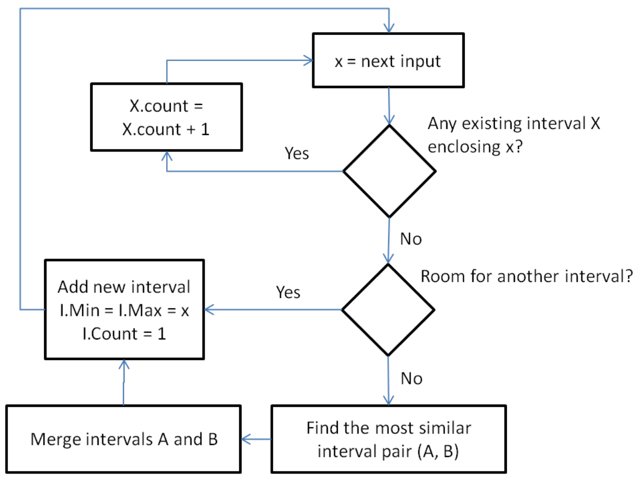
x = next input
X.count = X.count + 1
Any existing interval X enclosing x?
Yes
No
Add new interval I.Min = I.Max = x I.Count = 1
Room for another interval?
Yes
No
Merge intervals A and B
Find the most similar interval pair (A, B)

Fig. 1. The Modeling Algorithm

If there is no existing interval that match a specific input x and the maximum number of intervals has not yet been reached, a new interval I is created where

$$I.min = I.max = x$$

$$I.count = 1$$

If the maximum number of intervals has been reached, an analysis is performed in order to find and merge the two most similar adjacent intervals, in order to make room for a new interval according to above. The overall algorithm is illustrated in Figure 1. Initially there are no intervals and the first input thus result in the first interval created. Most early inputs will produce new intervals (until the maximum number of intervals have been reached), unless a specific input value is repeated. The merging of intervals will cause them to grow, which increases the probability of later inputs falling within the bounds of the interval. When merging two intervals A and B, they are replaced by a new interval C where

$$C.min = Min(A.min, B.min)$$

$$C.max = Max(A.max, B.max)$$

$$C.count = A.count + B.count$$

Only neighbor intervals are merged and intervals never overlap. The crucial aspect of this algorithm is how to select what intervals to join. We propose the following selection heuristics: Calculate a fitness value for each interval apart from the first, indicating its suitability for a merge with the lower neighbor interval. Then find the interval with best fitness value and merge it, i.e., with its lower neighbor. We suggest to base the fitness value on three properties:

- Proximity
- Density
- Count

Proximity is the absolute distance between the intervals of the candidate pair, i.e., the size of the gap in between the intervals. The smaller the gap, the more suitable merge. Density indicates the number of observations in relation to the width of the interval. A "spike", where many observations have been made in a narrow band, should not be merged with a "plateau" with lower density, i.e., where observations are more sparse. This is indicated by a density measure

$$I.density = I.count/(I.max - I.min)$$

The density fitness value of two merge candidate intervals A and B, a value between 0 and 1, indicates the similarity in density. The closer to 1, the better fitness for merge. The density fitness is calculated by

$$Min(A.density, B.density)/Max(A.density, B.density)$$

Finally, Count is the absolute number of observations accounted to the interval, i.e., $I.count$. If this is very low, the density becomes very sensitive to random variations in $I.count$. We therefore skip the density comparison if the count is below a certain threshold for either of the two intervals, we have used 5 as minimum in the prototype evaluation. Only if both intervals have at least 5 data points is the density property used, otherwise the fitness is based on proximity only. The density property is the dominant one when used, the proximity only matters in case the densities of two pairs are identical. To relax this a bit, we represent the density using an 8-bit integer in order to allow minor differences in actual density. We do not claim this to be the optimal merge selection heuristic, but the overall method proposed is however quite generic and can be used with different heuristics. In future work we plan evaluate the accuracy of this approach in greater depth and probably develop improved methods for selecting what intervals to merge.

B. *Suggested Implementation*

When implemented in an embedded system, we envision a solution where the intervals are stored in a linked list and kept sorted using insertion sort. The input is read from a circular RAM buffer which the measurements probes (code instrumentation) writes to. The modeling algorithm runs periodically on a low priority thread. This means that the modeling computations do not interfere with higher priority processes, but may cause loss of data if the system is under high load over a longer period of time. This might however be acceptable, since it is easaly detectable and can be compensated by increasing the rate of the modeling task, the size of the buffer, or both. The buffering will cause some additional RAM usage, but should typically only need to hold a few thousand observations. Consider a system with 100 tasks, each producing on average 100 observations per second. Assuming 4 bytes per observation, this would require about 40 KB for one second of execution. The buffer can however be eliminated by processing each data point when captured, i.e., the code instrumentation would update the statistical model directly, without buffering. This will cause CPU overhead at higher priority levels, thereby decreasing system performance, but might be an option for systems with severely limited amount of RAM and where a few percent lower system performance can be tolerated.

IV. Evaluation

An implementation of the algorithm has been made within the Tracealyzer tool; a trace visualization tool developed by Johan Kraft in his company Percepio AB. Note that the proposed algorithm has not yet been implemented in the embedded software recorder, but has been added as a prototype feature in the Tracealyzer tool itself, i.e., in the offline analysis of the recorded trace. However, the prototype implementation works in the same way as intended when integrated in an embedded software recorder; it gradually constructs and updates the statistical model for each data point given as input.

In our prototype implementation, Tracealyzer has also been extended to visualize the resulting timing models (i.e., the intervals) together with a plot of the raw data. Figure 2 show four diagrams generated using the prototype implementation, on a specific task using different configurations of the algorithm; 2, 4, 6 or 8 intervals allowed. The horizontal axis represents the value domain and the vertical axis the relative frequency of the different values. The rectangles represent the resulting intervals of the generated timing model, i.e., the output of the modeling algorithm, while the thin vertical bars (some are very short) represent the individual data points used as input, i.e., the recorded execution times of the analyzed task. The height of the interval rectangles correspond to the relative number of matching data points, and the absolute number of matching inputs of the interval (the count property) is shown on the top of the rectangle.

The execution time data used in the evaluation comes from a demo trace for RTXCview provided by Quadros Systems, Inc, which has been recorded on a development board with a microcontroller using the ARM7 core. The four diagrams of Figure 2 illustrate how the interval merging is affected by the interval limit. The most significant gap is around 120 μs, when allowing only two intervals, only this gap survives the merging of intervals. When allowing four intervals, a second major gap is recognized at around 135 μs, which is divided in two parts by an interval of size 1. The reason this single data point is not merged with the large interval to the right, is since it was not forced to by the interval limit, as four intervals were allowed. When increasing to six and eight intervals, additional larger gaps are recognized as the merging process is less aggressive. Figure 3 show diagrams from two other tasks from the same trace file (TMR0ISR is actually an interrupt service routine). In this case, four intervals are allowed in both cases, a relatively aggressive setting but which represents the distributions fairly well. In future work we plan to evaluate the accuracy of this method by running simulations using the statistical model as base for generating execution times and compare the resulting distribution with the real distribution used as input for the modeling. Such a comparison can be made using established statistical methods, such as the two-sample Kolmogorov-Smirnoff test [9], or KS test for short. A good overview of this test can be found at the U.S. NIST website [10]. The KS test is non-parametric and makes no assumptions on the underlying distribution of the data, which is important for this type of data. As a basic verification of the correctness of the implemented algorithm, traces have been generated using RTSSim, with a known distribution, and given as input to the algorithm. The resulting diagrams are shown by Figure 4. The distribution of the input data in the first case, for the task TEST, is uniformly distributed in two intervals, 30% between 200–300 μs and 70% between 400–450 μs. If allowing for two intervals only, the generated intervals fits exactly. The right diagram of Figure 3 shows a similar experiment using a different reference distribution.

V. Related Work

Several methods have been proposed for stochastic representation of task execution time. Bernat et al. [11] use Execution Profiles (EPs) to represent execution times of sections of a task. The EPs can then be combined to Joint Execution Profiles (JEPs) which represents the execution time of a whole task. This work resulted in the founding of Rapita Systems, Ltd. [12] and their RapiTime tool for execution time profiling. They use a hardware recorder device, the RTBx, to log execution times, which uses a large hard drive to store very long traces of execution time data. This is however fairly large and expensive equipment which hardly can be deployed in the field.

Hansen et al. [13] presented an approach for probabilistic estimation of the worst-case execution time (WCET) of tasks using extreme value theory, based on earlier work by Edgar and Burns [14]. They divide the sample data into blocks, take the maximum of each block (an independent random variable) and use these maximas to derive a Gumbel distribution, which essentially have a skewed bell-shape. Such an approach could possibly be combined with the statistical model proposed in this paper; instead of modeling the intervals as uniform distributions, we could fit the data within each interval to a theoretical distribution such as the Gumbel distribution. If this is possible in a memory efficient manner during continuous runtime monitoring remains to be investigated in future work.

RTSSim is not claimed a novel contribution conceptually, several similar simulation frameworks have been proposed. The most similar ones are Virtual Time, from Rapita Systems, Ltd. [12], ARTISST [15] and DRTSS [16]. An earlier result is STRESS [17], which inspired our first simulator implementation ART-ML back in 2002.

Nolte et al. [18] outlined how execution-time profiles can be used for probabilistic timing analysis in general and in the context of component-based software engineering. The idea was to let components keep track of their own execution times and they highlighted the problem of how to generate execution time profiles, which is effectively solved by this paper.

Kraft et al. [19] (at the time named Andersson) made a case study of practical application of simulation-based timing analysis in collaboration with ABB Robotics, and studied in particular how to model execution time data measured in runtime. A concept of *instance equivalence classes*, or IECs,

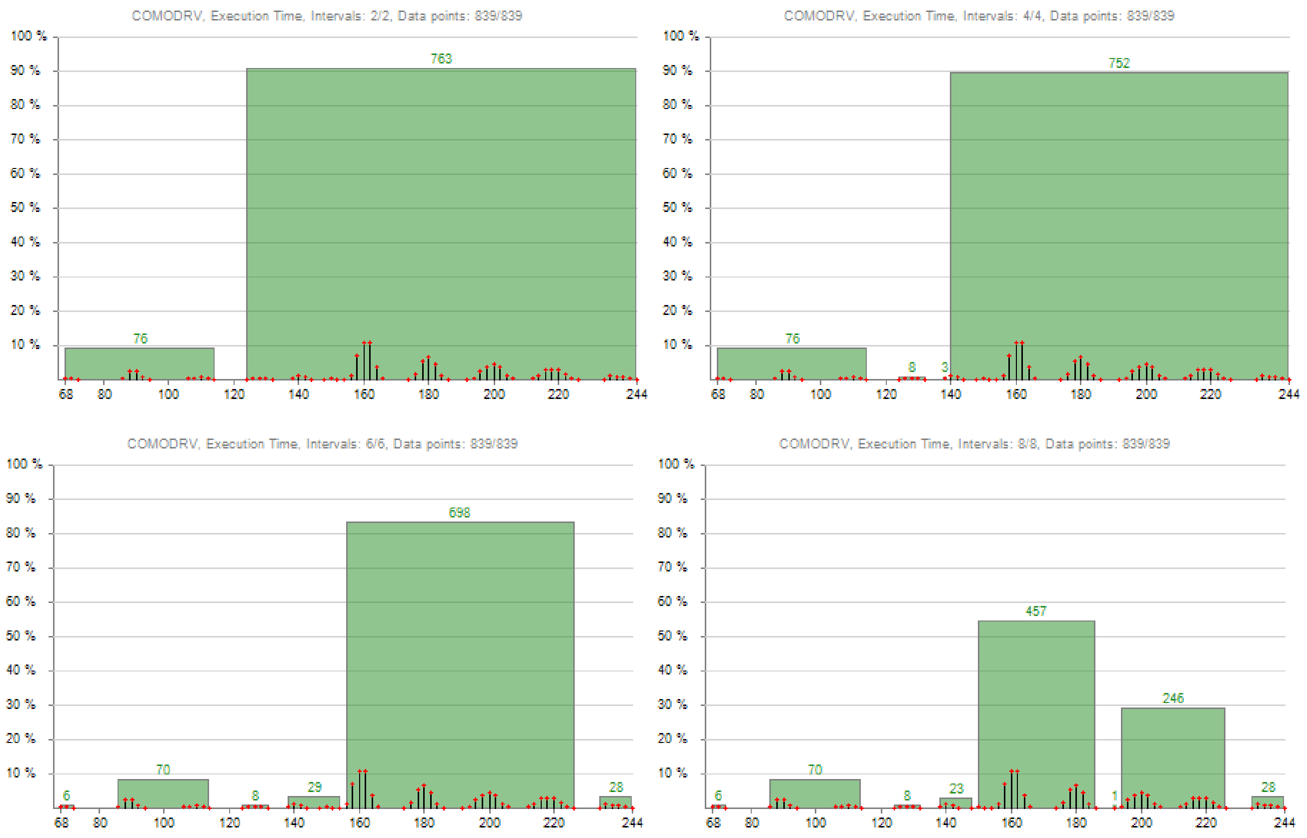

Fig. 2. COMODRV execution times, using 2, 4, 6 and 8 intervals

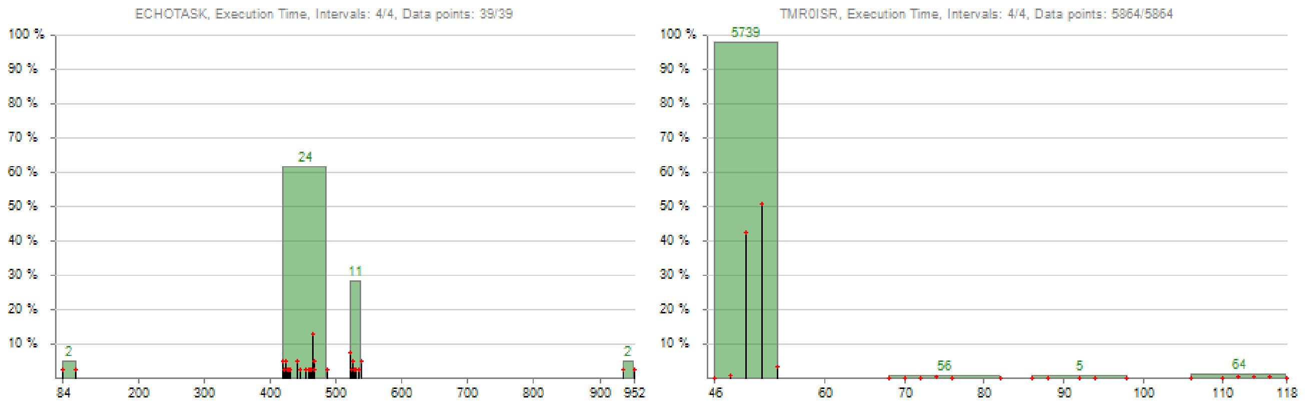

Fig. 3. ECHOTASK and TMR0ISR execution times, using 4 intervals

was introduced which is very similar to the statistical model proposed in this paper, and an algorithm was presented for offline identification of IECs. This was however not designed for use during continuous monitoring, but was intended for offline analysis of scheduling trace files.

VI. Conclusions

We have presented an approach for constant-memory monitoring of embedded systems, intended for continuous profiling of execution times and other runtime properties, which allows for profiling of systems in live operation over extended periods of time, possibly for years, using no hardware tracing equipment but only onboard software mechanisms. Due to the modest memory requirements, this approach is feasible even for microcontrollers with limited amounts of RAM. We have presented a prototype implementation of the algorithm and an evaluation which indicates that the method performs satisfactorily, although improvements in the selection heuristics most likely are possible. In future work we intend to implement,

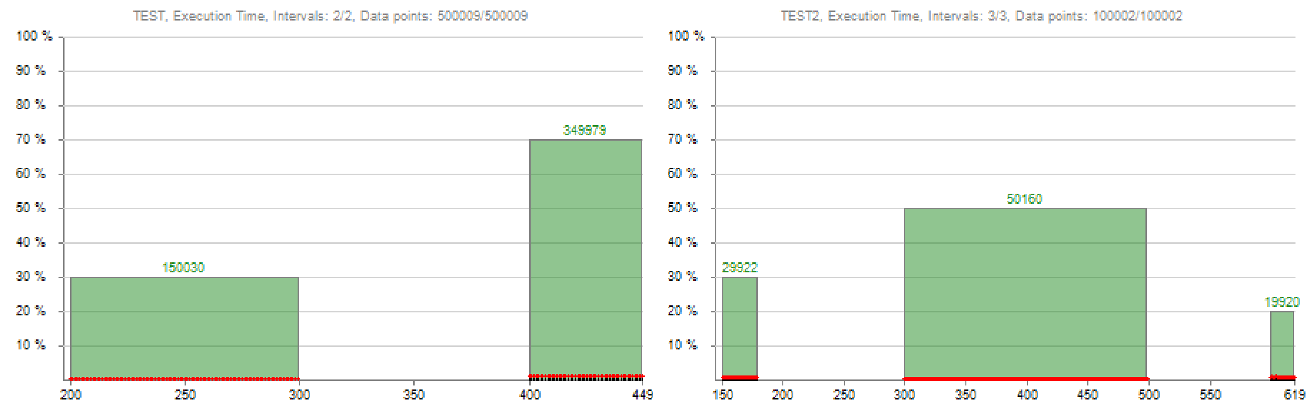

Fig. 4. Verification cases from RTSSim

evaluate and refine the modeling method using industrial cases; the ABB Robotics control system and the Ericsson telecom platform CPP.

Acknowledgment

This work is supported by the Artemis-JU project *CHESS*, performed in collaboration with Ericsson AB and ENEA AB, and the Swedish Foundation for Strategic Research.

References

[1] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in hard-real-time environment," *Journal of the Association for Computing Machinery*, vol. 20, no. 1, pp. 46–61, 1973.

[2] M. Joseph and P. K. Pandya, "Finding Response Times in a Real-Time System," *The Computer Journal*, vol. 29, no. 5, pp. 390–395, 1986.

[3] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings, "Fixed priority pre-emptive scheduling: An historical perspective," *Real-Time Systems Journal*, vol. 8, no. 2/3, pp. 173–198, 1995.

[4] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "Uppaal – a tool suite for automatic verification of real-time systems," in *4th DIMACS Workshop on Verification and Control of Hybrid Systems*, 1995.

[5] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, "Kronos: A Model-Checking Tool for Real-Time Systems," in *10th Intl. Conf. on Computer Aided Verification*, vol. 1427, 1998, pp. 546–550.

[6] J. Kraft, "Enabling timing analysis of complex embedded software systems," Ph.D. dissertation, Mälardalen University Press, August 2010.

[7] J. G. Huselius, J. Andersson, H. Hansson, and S. Punnekkat, "Automatic generation and validation of models of legacy software," in *12th IEEE Intl. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)*, Aug. 2006, pp. 342–349.

[8] J. Kraft, A. Wall, and H. Kienle, "Trace recording for embedded systems: Lessons learned from five industrial projects," in *Proceedings of the First International Conference on Runtime Verification (RV 2010)*. Springer-Verlag (Lecture Notes in Computer Science), November 2010.

[9] I. M. Chakravarti, J. Roy, and R. G. Laha, *Handbook of methods of applied statistics*. Wiley, New York, 1967.

[10] "U.S. National Institute of Standards and Technology (NIST) e-Handbook of Statistical Methods, Section 1.3.5.16: Kolmogorov-Smirnov Goodness-of-Fit," http://www.itl.nist.gov/div898/handbook.

[11] G. Bernat, A. Colin, and S. M. Petters, "WCET analysis of probabilistic hard real-time systems," in *23rd IEEE Real-Time Systems Symposium (RTSS'02)*, Dec. 2002, pp. 289–300.

[12] "Rapita Systems, Ltd." http://www.rapitasystems.com.

[13] J. Hansen, S. Hissam, and G. Moreno, "Statistical-based WCET estimation and validation," in *9th Intl. Workshop on Worst-Case Execution Time Analysis (WCET'09)*, Jun. 2009, pp. 123–133.

[14] S. Edgar and A. Burns, "Statistical analysis of wcet for scheduling," in *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, ser. RTSS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 215–. [Online]. Available: http://portal.acm.org/citation.cfm?id=882482.883802

[15] D. Decotigny and I. Puaut, "Artisst: An extensible and modular simulation tool for real-time systems," in *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Washington D.C.*, 2001.

[16] M. Storch and J.-S. Liu, "DRTSS: A Simulation Framework for Complex Real-Time Systems," in *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium (RTAS'96)*. Dept. of Computer Science, Illinois Univ., Urbana, IL, USA, 1996.

[17] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings, "STRESS: A Simulator for Hard Real-Time Systems," *Software-Practice and Experience*, vol. 24, no. 6, pp. 543–564, June 1994.

[18] T. Nolte, A. Möller, and M. Nolin, "Using components to facilitate stochastic schedulability analysis," in *WIP at 24th IEEE Real-Time Systems Symposium (RTSS'03)*, Dec. 2003, pp. 7–10.

[19] A. Wall, J. Andersson, J. Neander, C. Norström, and M. Lembke, "Introducing Temporal Analyzability Late in the Lifecycle of Complex Real-Time Systems," in *9th Intl. Conf. on Real-Time and Embedded Computing Systems and Applications (RTCSA'03)*, 2003.