Predictability Analysis of Interruptible Systems by Statistical Model Checking

Josef Strnadel
Faculty of Information Technology, Centre of Excellence IT4Innovations, Brno University of Technology

Editor's note:
This article proposes a model comprising of a network of stochastic timed automata for predictability analysis of interruptible systems via statistical model checking.

—Tulika Mitra, National University of Singapore
—Jürgen Teich, Friedrich-Alexander-Universität Erlangen-Nürnberg
—Lothar Thiele, ETH Zurich

Predictability is one of the most important attributes of many systems. The problem is that a designer of a predictable system must face many sources of unpredictability. Among others, they result from detecting an event through interrupts. The main advantage of such a detection is that no CPU time is consumed regarding an event until the corresponding interrupt is triggered. Although this may not be so, the effects of interrupts may look like random variables, elimination of which is not possible in practice. However, if we analyze a system carefully then we can bound the effects.

To facilitate the analysis at a higher level of abstraction, we need sufficiently precise and credible information from lower levels. In particular, the analysis of real-time (RT) systems is simplified if the values of parameters such as the best-case execution time (BCET), worst-case execution time (WCET) or worst-case response time (WCRT) are available. In the area of RT systems, many techniques exist for evaluating the parameters analytically. Typically, the techniques restrict themselves to systems with periodic behaviors. However, perfect periodicity does not exist in practice, which results in two way-outs. The first one extends the periodic perspective by concepts such as sporadic servers, allowing us to analyze the parameters statically by analytical instruments. The second wayout lies in using a real platform, its credible simulator, emulator, etc., for analyzing the parameters dynamically.

Due to the complexity of the predictability analysis problem, this manuscript restricts itself only to issues in the digital CPU-based systems that are driven by events initiated by interrupts. Predictability of these systems is difficult to analyze, especially when interrupts are prioritizable, may be nested or unmasked at runtime. Moreover, an interrupt may occur at an arbitrary time, asynchronously to a program that the CPU executes. The facts lead to an explosion of situations entering the analysis process. If the analysis rests on the simulation over a credible model of a system, then the number of states needed to represent and

Digital Object Identifier 10.1109/MDAT.2017.2766568
Date of publication: 25 October 2017; date of current version: 23 March 2018.

Copublished by the IEEE CEDA, IEEE CASS, IEEE SSCS, and TTTC

analyze the system accurately may easily exceed the amount of available computational resources [1].

This manuscript addresses some challenges of both creating such a model and its application for predictability analysis purposes. To the best of my knowledge, this manuscript is the first one that addresses the challenges in such a complex form.

Toward the problem

The following paragraphs present key issues that relate to interrupts. Without any loss of generality, they only focus on a single-CPU system.

Principles

For simplicity, let us assume that the CPU is either disabled or continuously busy by processing instructions. This manuscript distinguishes three classes of the CPU business, based on the execution level. In Figure 1, the symbols L_{main}, L_{ctx}, and L_{isr} indicate the levels. A dotted area indicates the period of time in which the CPU is busy due to executing instructions at the given level. The illustration starts at L_{main} where the CPU processes instructions from the main program loop, "main()" in brief. While executing part (1) of main(), an interrupt request (IRQ) may occur at t_{IRQ}. However, this may happen under various scenarios, where each one delays a reaction to the IRQ. Examples of such scenarios are the masking of the IRQ or the start of processing an instruction. After such a scenario is over, the level switches to L_{ctx}, starting the process (2) of saving the CPU's state/context onto the stack ("stacking"). The process typically ends by disabling all maskable interrupts and then consecutive arbitration of IRQs that are pending at that moment. The arbitration fetches the vector of the highest-priority pending IRQ and puts it into the program counter (PC) of the CPU. The placement starts the associated interrupt service routine (ISR) at L_{isr}.

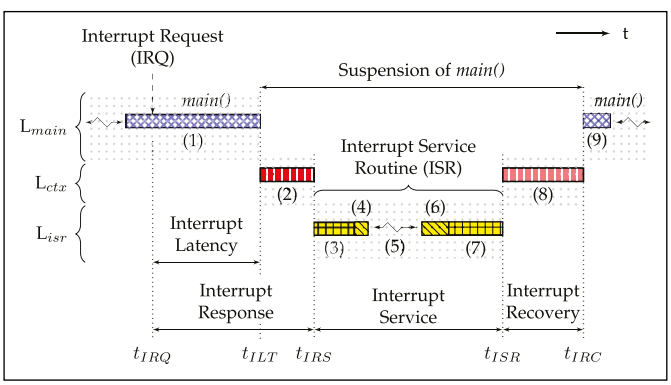

Figure 1. Interrupt-related mechanisms.

An ISR may be started by a prologue (3) able to perform actions such as running an ISR-entry code for an operating system (OS). Then, the service itself runs within (4)–(6), followed by an epilogue (7) running an ISR-exit code. The levels L_{ctx}, L_{isr} may nest (embed) in a recurrent way; particularly, if an IRQ of a sufficiently high priority becomes enabled in (4), then the execution of (4) at L_{isr} stops during (5). The stop is an effect of nesting of further levels L_{ctx}, L_{isr} of higher priority between (4) and (6) of lower priority. After an ISR is completed, the CPU context typically switches back (8) to the state before entering the ISR ("unstacking"). Some of the principles may vary if the nesting of ISRs is off or interrupts are subject to techniques such as tail-chaining or late-arriving in ARM Cortex.

Issues and perspectives

Figure 1 clarifies the following issues regarding interrupts. First, the CPU processes IRQs prior to main(), resulting in stopping the execution of main() during processing an IRQ. Second, the processing is not immediate, but delayed due to mechanisms such as executing an instruction, masking of interrupts, stacking of the CPU context or nesting of ISRs. Third, each level of stacking or nesting needs a memory to store the CPU context or local data of the ISR. Finally, an IRQ may occur anytime and the priority of a nonmaskable interrupt (NMI) is typically one of the highest ones.

From the practical point of view, the following perspectives are important. Each of them concentrates on various attributes of predictability. The first one relates to time the CPU spends by executing main() under the given interrupt scenario. The time decreases with factors such as interarrival rate of interrupts ($f_{interrupt}$) or the duration of ISRs. If a factor exceeds a certain level, then a system may stop working correctly or collapse suddenly. Such an excess is typically denoted as the interrupt overload (IOV) problem, the seriousness of which grows with the criticality of main(). The second perspective relates to ISRs, for which the inter rupt response and interrupt service times are particularly important. The third one relates to the dynamics of utilizing a random access memory. It typically focuses on analyzing the worst-case utilization of stacks or buffers.

The model proposed in this manuscript is able to cope with all of the issues and perspectives.

State of the art

Various mechanisms of increasing predictability under scenarios such as IOV exist. Many of them try to increase it by preventing $f_{interrupt}$ from exceeding the critical level [2]. In our previous research [3], we dealt with the modeling and analysis of the preventive mechanisms as well. But our approach has not reflected facts such as NMIs or variability in executing ISRs. Under such facts, the predictability analysis remains a problem despite all of the mechanisms. Similar limitations can be found in existing approaches as well. For example, Chattopadhyay et al. [4] analyzed the predictability of a system under disabled interrupts. Further approaches, such as [5] and [6], limit themselves to a simplified static model of a system and analytical solution of the problem. In particular, they support just periodic interrupts, no nesting of ISRs, no execution jitters, no un/masking, no priorities, and no arbitration of IRQs at the runtime. Kidd et al. [7] expect that an IRQ cannot occur anytime, but after the so-called hyperperiod. Contrary to that, the approach in [8] supposes that an IRQ arrives whenever an instruction is completed. Finally, Kroening et al. [9] consider nesting and priorities of interrupts, but did not deal with the unpredictability of IRQ arrival times, un/masking of interrupts at the runtime and variability in executing ISRs.

Proposed approach

This section outlines the applicability of the publicly available toolset UPPAAL SMC [10] with respect to the modeling a system from the interrupt perspective and its analysis from the predictability viewpoint. The toolbox allows us to create a model of an RT system on the basis of stochastic timed automata (STAs) and to analyze its properties in the given stochastic environment. To check whether a property holds or not, UPPAAL SMC applies the so-called statistical model checking (SMC) technique. SMC monitors simulation runs over a model in order to process them statistically. This continues until the probability of satisfying a specified property meets the predefined degree of confidence.

Works such as [11] show that methods based on SMC easily scale to industrial size systems. These works conclude that the methods scale logarithmically in the size of the analyzed models and that they are trivially parallelizable and still scale sublinearly in the time domain.

Models

For the predictability analysis purposes, we have proposed a set of STA models (Figure 2) that are related to the following parts of a system: a) CPU, b) IRQ controller, c) ISR, and d)–f) IRQ sources. Each of them is parameterizable and represents a template that may be instantiated many times. Despite our models comprise components such as pipelines and caches, we skip further details to the

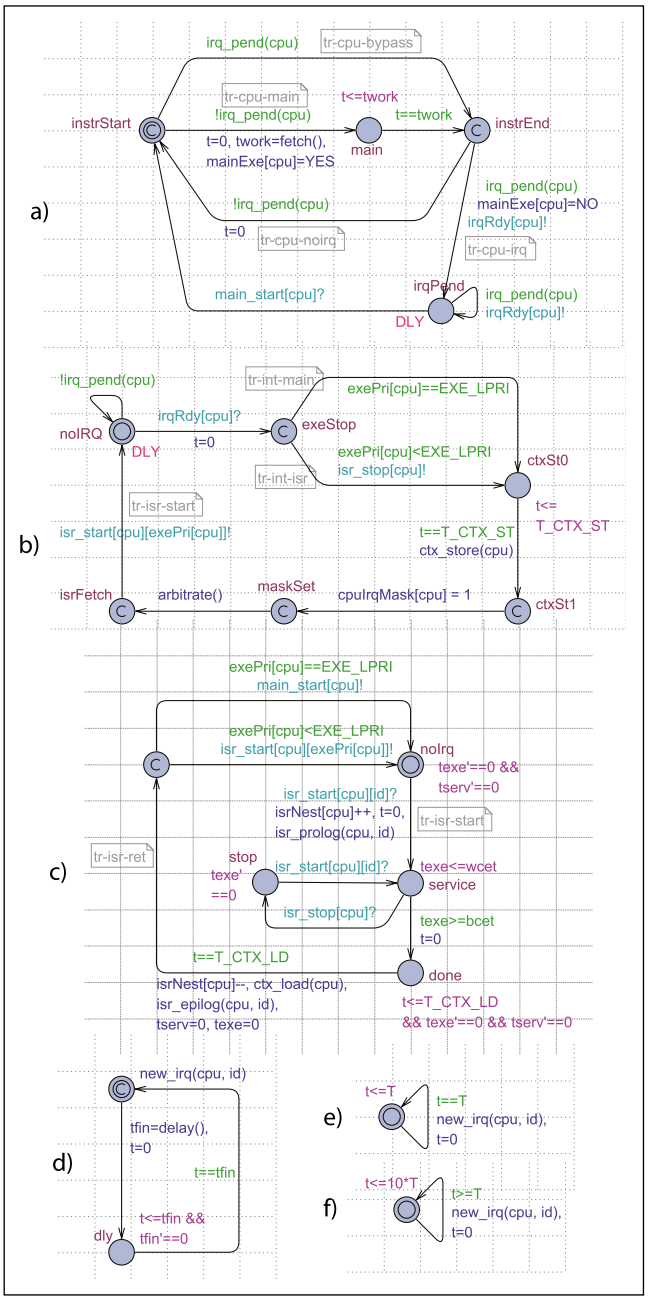

Figure 2. Proposed STA models of a) CPU, b) IRQ controller, c) ISR, and d)–f) IRQ sources.

components as their presentation is beyond the scope of this manuscript. For more details, refer to [12].

The model of a CPU (Figure 2a, parameter "cpu") starts in its initial location ("instrStart") where it prepares for fetching an instruction of main(). Then, either of the transitions "tr-cpu-main," "tr-cpu-bypass" is fired before transiting to "instrEnd." A pending IRQ is detected by the function "isr_pend()." If no IRQ is pending ("tr-cpu-main"), then the process of fetching an instruction starts, which ends by entering "main." The model stays here until the instruction is completed, which takes "twork" units of time. In fact, "twork" is a random variable with the uniform distribution of probability within [1,10). Otherwise, if an IRQ is pending, no fetching starts ("tr-cpu-bypass"). In "instrEnd," two situations may occur depending on whether an IRQ is pending ("tr-cpu-irq") or not ("tr-cpu-noirq"). During "tr-cpu-irq," the CPU sends a message ("irqRdy[cpu]!") to the corresponding IRQ controller to let it consume the CPU time for IRQ processing purposes. In "irqPend," the model stays while a pending IRQ exists.

The model of an IRQ controller (Figure 2b, parameter "cpu") starts in "noIRQ." Here, it stays until the corresponding CPU agrees to process an IRQ. Then it moves to "exeStop" with the two outgoing transitions. The first one ("tr-int-main") represents the interruption at a level corresponding to main(). As the execution of main() is already stopped, no action is associated with the transition. The second transition ("tr-int-isr") relates to the interruption at the level corresponding to an ISR. Here, the actual ISR must stop ("isr_stop[cpu]!") as an ISR of a more significant priority, represented by a lower numerical value, is pending. Then, the model moves to "ctxSt0" where the stacking starts. After "T_CTX_ST" units of time, the stacking is completed ("ctxStl"). Then, the CPU sets the global interrupt mask ("maskSet"), arbitrates pending IRQs ("arbitrate()"), fetches the vector of the ISR that has won the arbitration, and finally, starts the ISR ("tr-isr-start").

The model of an ISR (Figure 2c) has four parameters ("cpu," "id," "bcet," and "wcet"), where "id" is a unique identifier of the corresponding source of IRQs; "bcet" and "wcet" are the best-case and worst-case execution times of the ISR, respectively. The model uses the local clocks "texe" and "tserv" to measure the execution and servicing times of an ISR, respectively. The model starts in "noIrq," where it waits until it receives a signal to start the ISR. Until then, neither of the clocks "texe," "tserv" measures time. Afterward, the model increments the CPU's level of nesting executions due to ISRs performs the prologue of the ISR and enters "service." Here it consumes the CPU time, the amount of which ranges from "bcet" to "wcet." Implicitly, the model allows the nesting of ISRs. So, if a higher-priority IRQ becomes pending, then the servicing stops ("stop") until it receives the signal to start again. By then the ISR does not consume the CPU time, so the measurement using "texe" stops too. But "tserv" still runs to measure how much time the ISR spends in the "stop." From "service," the model moves to "done" in which the measurement using "texe," "tserv" stops. Then the return from an ISR starts ("tr-isr-ret") by the unstacking and consequent resumption of either main() or a lower-priority ISR.

Finally, we have created models of various sources of interrupts. Their representatives (Figure 2d–f) include a maskable periodic IRQ from a timer (TMR), unmaskable aperiodic IRQ from a software interrupt (SWI), and maskable aperiodic IRQ from a keyboard interface (KBI). The model of an IRQ source has three parameters ("cpu," "id," and "T"), where "id" is a unique identifier of the IRQ and "T" defines the interarrival time of the IRQ. IRQs from TMR arrive exactly after "T" units of time. For KBI, they are uniformly distributed in the interval ["T," "10×T"). For SWI, the distribution is given by a user-defined "delay()" function that returns a random number given by the normal distribution of probability. This is made possible by using the stopwatch concept, details of which follows. First, "delay()" returns the time to produce a new IRQ. The (final) time captures into the clock "tfin." Then the model moves to "dly," where it waits until the clock "t" reaches "tfin" ("tfin" will stop until then). The function "new_irq(cpu, id)" simply sets the request for the given CPU and IRQ sources.

Scenarios

Applicability of our models is demonstrated by the four scenarios. For each of them, properties of the models are analyzed by various SMC means. The implicit setup represents the scenario Sc(0), denoted by Sc(0)/N+ as well ("N+" means that the nesting of ISRs is allowed). Further scenarios (Sc(0)/N-, Sc(1), Sc(2)) result from modifications of Sc(0) that are emphasized in the next paragraphs.

In all the scenarios, both the CPU and IRQ controllers are instantiated by "cpu=0."

Using the templates from Figure 2d–f, Sc(0) instantiates the four sources of IRQs: SWI0, TMR1, TMR0, and KBI0. Their priorities are fixed and their values are 1 (the most significant) for SWI0, 3 for TMR1, 4 for TMR0, 7 (the least significant) for KBI0. The priority of main() is set to 8. Masking of IRQs is possible just for TMR1, TMR0, and KBI0. After the CPU resets, all maskable IRQs are disabled. Parameters of the IRQ sources are as follows: SWI0 (cpu=0, id=1), TMR1 (cpu=0, id=3, T=500), TMR0 (cpu=0, id=4, T=1000), and KBI0 (cpu=0, id=7, T=50), with the results shown in Figure 3a–d. In relation to SWI0, "delay()" returns a random number of the normal distribution of probability with its mean set to 500 and the standard deviation of 50. Parameters of the corresponding ISRs are also adjusted so: SWI0 (cpu=0, id=1, bcet=35, wcet=35), TMR1 (cpu=0, id=3, bcet=170, wcet=230), TMR0 (cpu=0, id=4, bcet=250, wcet=250), and KBI0 (cpu=0, id=7, bcet=95, wcet=120). For the constants, it holds DLY=1, EXE_HPRI=0, EXE_LPRI=8, T_CTX_ST=5, TCTX_LD=5, CTX_BYTES=6, STK_SIZE=256. Implicitly, the analysis has been performed within 25,000 μs.

In all the scenarios, main() calls an initialization subroutine, the execution of which takes 1000 μs. Then, it enters the infinite loop, an iteration of which operates as follows. If the time equals to 1100 μs, then the IRQ mask for KBI0 is cleared. 100 μs later, the mask for TMR0 is cleared and after 100 μs, the mask for TMR1 is cleared. 500 μs later, masks for TMR1 and TMR0 are set up again along with the CPU's mask. In 200 μs, the mask for TMR0 is cleared while the mask for KBI0 is set up and the CPU's mask is cleared. 100 μs later, the mask for TMR1 is cleared, the iteration ends, and the time is reset. For the resulting dynamics, see Figure 3e. Details of the stack manipulation are as follows. In main(), the probability of calling or returning from a subroutine by the corresponding instruction is 25%. Each such a call or return adds two bytes to the stack or removes them from the stack, respectively. The stack sizes (in bytes) for particular ISRs are as follows: 8 for SWI0, 16 for TMR1, 24 for TMR0, and128 for KBI0. Such a space is allocated or released after the corresponding ISR starts or before it ends, respectively.

Scenarios Sc(0)/N-, Sc(1), and Sc(2) result from Sc(0) as follows. Comparing Sc(0), Sc(0)/N- does not

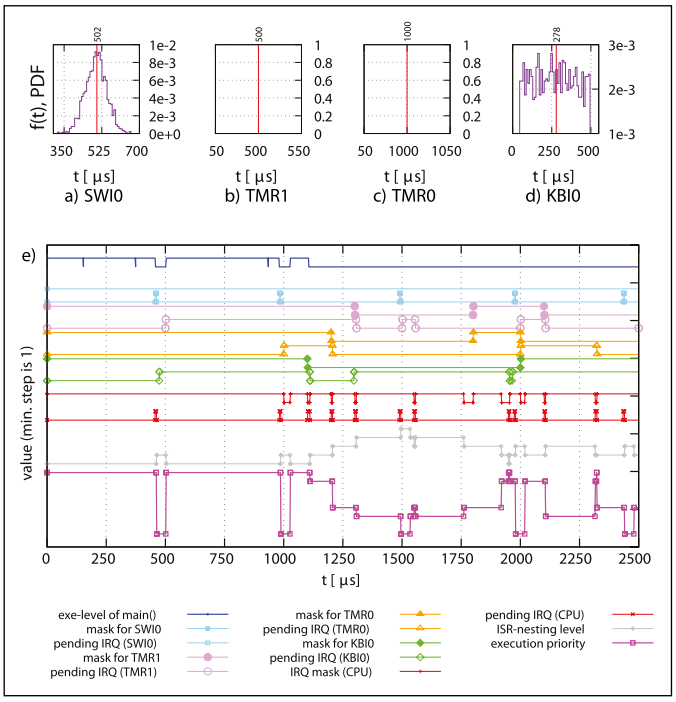

Figure 3. PDFs of the IRQ arrival times for particular IRQ sources (a–d) and a cutout of selected 2500 μs-wide waveforms resulting from a single simulation (e).

allow ISRs to nest. Sc(1) decreases both the mean and deviation in SWI0's delay() function (Figure 2d) 10 times. This allows IRQs from SWI0 to arrive more frequently. Finally, Sc(2) decreases "bcet," "wcet" of ISRs for TMR1, TMR0, respectively, 10 times. This allows the ISRs to finish sooner.

Queries and results

Regarding the perspectives mentioned at the end of "Issues and perspectives," this section tries to demonstrate that our approach is capable of solving practical problems. Despite the generality of the approach, this section is just limited to some representatives of the problems. For a given scenario ("Scenario"), each of the problems relates to an analysis of the property/attribute that is important from the predictability viewpoint. Such an analysis may be related to the CPU load due IRQs, stack utilization, interrupt service, and latency times or throughput of produced/serviced IRQs. Details of such an analysis are shown in Figure 4.

Figure 4a results from the SMC query "simulate 1 [<=25,000] 100.0*(t_main[0]/t)," where 1 represents the number of simulation runs, 25,000 represents

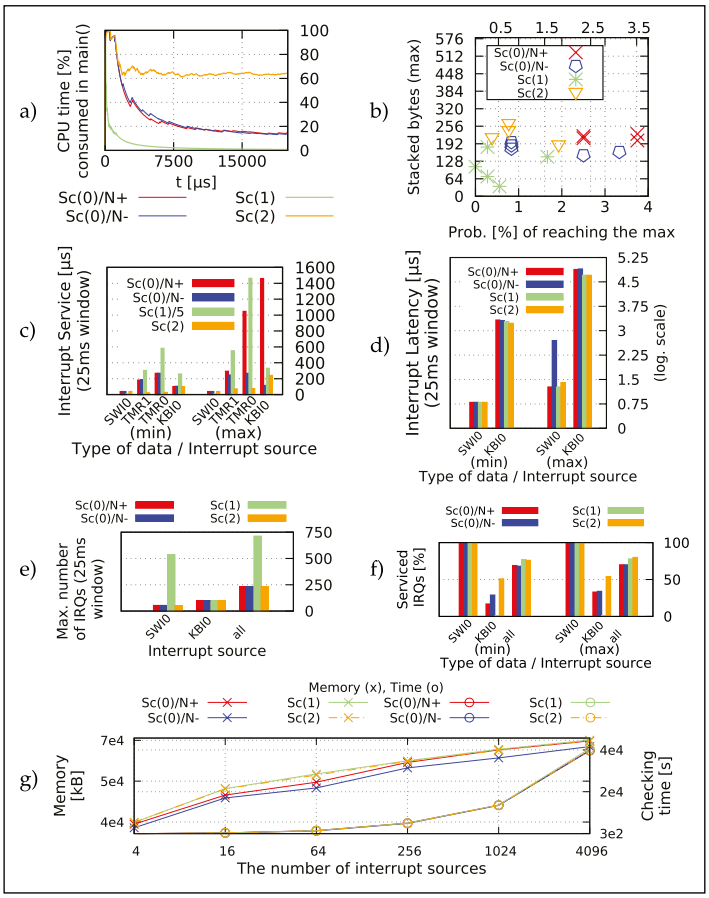

Figure 4. Results of analyzing predictability of selected properties attributes under various scenarios. a) The CPU time spent by executing the main program loop. b) Maximum utilization of the stack. c) Jitter of interrupt service times. d) Interrupt latency jitter. e) Maximum number of interrupt requests. f) Jitter of the percentage of serviced interrupt requests. g) Scalability of the proposed approach in memory and time domains.

the maximum value of the simulation time and "t_main[0]" and "t" represent the time spent by the CPU, indexed by 0, in main() and the total time that has elapsed, respectively. It shows that turning the nesting of ISRs on or off has a negligible impact to the observed attribute, while an impact of SC(1), SC(2) is significant.

Figure 4b results from the SMC query "E[<=25,000; 10] (max: isrSP[0])" capable of finding the maximum increment of the CPU's stack pointer (isrSP[0]). It evaluates both the maximum within 10 simulation runs, each taking 25,000 μs, and the probability of reaching the maximum. The results show that the maximum is lower under Sc(0)/N-, Sc(1) and that the probability of reaching the maximum is smaller for Sc(1) and Sc(2).

Figure 4c results from the SMC queries "E[<=25,000; 10] (max: X.tserv)," "E[<=25,000; 10] (min: X.tserv)," where X represents one of SWI0, TMR1, TMR0, and KBI0. It shows that, for lower-priority IRQs, the time is prolonged significantly under Sc(0) and yet more under Sc(1).

Figure 4d results from the measuring time between *i*th occurrence of an IRQ and starting the corresponding ISR of that IRQ. A special STA must exist for that purpose and an instance of the STA must be created dynamically, after an IRQ occurs. As further explanation of the concept needs many illustrations and comments, it is omitted.

Figure 4e results from the SMC query "E[<=25,000; 10] (max: numIRQs[0])," where numIRQs[0] represents the number of IRQs that has arrived to the CPU till a particular instant.

Figure 4f results from the SMC queries "E[<=25,000; 10] (min: 100.0*numberOfDoneISRs/numberOfIRQs)," "E[<=25,000; 10] (max: 100.0*numberOfDoneISRs/numberOfIRQs)," sequentially applied to the associated sources of IRQs (SWI0, KBI0, all). It shows that, for KBI0, the observed ratio changes significantly under Sc(0).

Figure 4g illustrates the scalability of our approach (mean memory and mean time consumed during the process of checking SMC queries, respectively) as a function of the number of interrupt sources. Even though the number of interrupt sources in today's systems typically does not exceed 256, we have analyzed the scalability up to 4096 interrupt sources. The figure shows that, approximately, our approach scales sublinearly in the memory domain and linearly in the time domain.

THIS PAPER PRESENTS a novel model of an interrupt-driven CPU-based system. Its novelty relies on stochastical timed automata, which allows us to model sources of unpredictability not covered by existing approaches. The manuscript shows that by means of a statistical model checker, it is possible to handle a variety of aspects with respect to analyzing predictability under various scenarios. Such an approach is capable of facilitating the analysis of parameters such as interrupt latency or interrupt servicing time and minimizing the over/underestimation of their values. ■

Acknowledgments

This paper was supported by The Ministry of Education, Youth and Sports of the Czech Republic from the National Programme of Sustainability (NPU II); project IT4Innovations excellence in science - LQ1602.

References

[1] C. Baier and J. P. Katoen, *Principles of Model Checking*, Cambridge, UK: MIT Press, May 2008.

[2] J. Regehr and U. Duongsaa, "Preventing interrupt overload," *SIGPLAN Not.,* vol. 40, no. 7, pp. 50–58, 2005.

[3] J. Strnadel and M. Risa, "On analysis of software interrupt limiters for embedded systems by means of UPPAAL SMC," in *Proc. Austrochip Workshop Microelectronics (Austrochip),* Villach, Austria, Oct. 2016, pp. 45–50.

[4] S. Chattopadhyay, M. Tresina, and S. Narayan, "Worst case execution time analysis of automotive software," *Procedia Engineering*, vol. 30, pp. 983–988, 2012.

[5] J. Kotker, D. Sadigh, and S. A. Seshia, "Timing analysis of interrupt-driven programs under context bounds," in *Proc. Formal Methods Comput. Aided Des. (FMCAD)*, Austin, TX, USA, Oct. 2011, pp. 81–90.

[6] L. E. Leyva-del Foyo, P. Mejia-Alvarez, and D. de Niz, "Integrated task and interrupt management for real-time systems," *ACM Trans. Embed. Comput. Sys.*, vol. 11, no. 2, pp. 32:1–32:31, 2012.

[7] N. Kidd, S. Jagannathan, and J. Vitek, "One stack to run them all: reducing concurrent analysis to sequential analysis under priority scheduling," in *Proc. Int. SPIN Conf. Model Checking Software,* Enschede, The Netherlands, Sept. 2010, pp. 245–261.

[8] X. Wu, Y. Wen, L. Chen, W. Dong, and J. Wang, "Data Race detection for interrupt-driven programs via bounded model checking," in *Proc. IEEE Int. Conf. Software Security Reliab. Companion,* Gaithersburg, MD, USA, 2013, pp. 204–210.

[9] D. Kroening, L. Liang, T. Melham, P. Schrammel, and M. Tautschnig, "Effective verification of low-level software with nested interrupts," in *Proc. 2015 Des. Autom. Test Europe Conf. Exhibition,* Dresden, Germany, 2015, pp. 229–234.

[10] A. David, K. Larsen, A. Legay, M. Mikuionis, and D. Poulsen, "Uppaal SMC Tutorial," *Int. J. Software Tools Tech. Transfer*, vol. 17, no. 4, pp. 397–415, 2015.

[11] J. H. Kim, A. Boudjadar, U. Nyman, M. Mikucionis, K. G. Larsen, and I. Lee, "Quantitative schedulability analysis of continuous probability tasks in a hierarchical context," in *Proc. 2015 18th Int. ACM SIGSOFT Symp. Component-Based Software Eng. (CBSE),* Montréal, QC, Canada, May 2015, pp. 91–100.

[12] A. E. Dalsgaard, M. C. Olesen, M. Toft, R. R. Hansen, and K. G. Larsen, "METAMOC: Modular execution time analysis using model checking," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010),* B. Lisper, Ed. Dagstuhl, Germany: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010, pp. 113–123.

Josef Strnadel is an Assistant Professor at the Centre of Excellence IT4Innovations, Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic. His research interests include the dependability of embedded and real-time systems. He has a PhD in information technology from the Brno University of Technology.

■ Direct questions and comments about this article to Josef Strnadel, Brno University of Technology, Brno, Czech Republic; e-mail: strnadel@fit.vutbr.cz.