On Analysis of Software Interrupt Limiters for Embedded Systems by Means of UPPAAL SMC

Josef Strnadel, Michal Riša
Brno University of Technology, Faculty of Information Technology, Centre of Excellence IT4Innovations
Brno, Czech Republic
strnadel@fit.vutbr.cz, irisa@fit.vutbr.cz

***Abstract*—The paper deals with a novel method of modeling and analysis of software interrupt managers for event-driven embedded systems by means of the stochastic timed automata and statistical model checking instruments. The above-mentioned system is typically formed of a real-time part expected to produce correct responses and meet all predetermined timing constraints at runtime, even in adverse conditions such as an excessive rate of events caused by interrupts. Because of the asynchronous nature of interrupts, their impact to the system being interrupted must be modeled and analyzed very carefully for various interrupt scenarios – either using classical analytical/formal approaches able to cover systems and interrupts with deterministic behavior or using probabilistic ones able to deal with a stochastic behavior too. The paper is focused on the latter (probabilistic) approaches to show a style of such a modeling and show how and that both the analysis phase of a system can be facilitated and the information about a system behavior under particular configuration/scenarios can be produced using the instruments.**

***Index Terms*—event-driven; embedded; system; interrupt; overload; management; software; interrupt limiter; statistical model checking; stochastic timed automaton**

I. Introduction

The method presented in this paper is applicable to systems that are i) *embedded*, i.e., typically control systems having very limited computational resources, but able to comply criteria such as low power consumption, low weight, small size etc., ii) *real-time* (RT), i.e., they must react to events both correctly and on-time, according to their specification [1] and iii) *event-driven*, i.e., stimuli entering the system are signaled by *events*, each of which being typically triggered by an associated *interrupt* (INT). Moreover, it is supposed that RT properties are guaranteed by an extra software (typically, an operating system) layer being executed by a device such as a *central processor unit* (CPU). To avoid an unexpected failure of the layer (and consequently, of RT properties), one must specify assumptions [2] about faults the system must be resilient to (so-called *fault hypothesis*) and load induced by the environment (*load hypothesis*). This paper abstracts from the fault hypothesis (not reducing its weight nohow) and focuses just on problems related to the latter.

The paper is organized as follows. Section II outlines basic terms w.r.t. problem being solved (II-A) and basic instruments (II-B). Section III, presents concepts of existing solutions to the problem (III-A) and of modeling timed systems and their statistical model checking in the UPPAAL SMC tool (III-B). Section IV describes our approach to modeling the problem and checking the model. Section V summarizes results achieved on basis of our approach while section VI concludes the paper.

II. Preliminary

A. Towards the Problem Formulation

Basically, an event can be detected either on basis of a i) *polling-loop* for which it is typical that a special, event-related flag is continuously tested by a CPU to detect whether the event has occurred or not, or ii) an *interrupt* (INT), main advantage of which is that no CPU time is consumed w.r.t. the event until an INT related to the event is triggered.

To better understand and define the problem, it should be noted that if an INT occurs then an extra CPU time and a memory space are required to store the recent CPU context; after it is stored, a *service routine* (ISR) associated with the (highest-priority) pending INT starts to be executed by the CPU. After the ISR is completed (and if there is no pending INT request, i.e. IRQ, at the moment), the context is restored back in order to resume the CPU execution being interrupted due to the INT triggering. Otherwise, further ISR – i.e., ISR corresponding to the highest-priority pending IRQ – is started.

Since the CPU executes an ISR prior to the main program loop (*main()* in brief), occurrence of each enabled, i.e., unmasked, INT delays the loop for a certain time (let it be denoted by $t_{INTover}$) needed to process the CPU context and service the INT. If $t_{INT} \leq t_{INTover}$, where t_{INT} is the time between successive INTs, then no CPU time remains to execute the main program loop because of the excessive *INT interarrival rate* ($f_{INT} = t_{INT}^{-1}$). Let it be emphasized herein that – excluding an ISR code – any code such as an RT application code is executed within *main()*. Consequently, a system may stop working correctly or collapse suddenly as f_{INT} increases – this is typically denoted as the *interrupt overload* (*IOV*) problem, seriousness of which grows with the criticality of *main()*.

B. Statistical Model Checking

Various techniques can be utilized to check whether particular (typically, formally specified) properties are guaranteed for a given behavior of a system; in this paper it is supposed that so-called *model checking* (MC) [3] technique is utilized for that purpose; it has been implemented in several powerful tools such as SPIN [4] or SMV [5] being successfully applied

TABLE I
PARAMETERS OF SOFTWARE INTERRUPT LIMITERS ACCORDING TO [12]

Parameter	*Description*
t_{poll}	time needed to test an event flag
t_{ictx}	overhead of saving/restoring the ISR context
t_{flip}	time needed to disable/enable INTs
t_w	ISR execution time
t_{adj}	overhead of adjusting and start a timer
t_{exp}	overhead of timer expiration service
t_{cnt}	time to increment the counter value and test if it is below the threshold value + overhead of clearing the counter
t_{tmr}	time to expire a timer
t_{cri}	time to disable INTs after t_{INT} drops below the $t_{arrival}$ value; during the time, unlimited number of INTs can occur to overload the system

in practice. However, even though various optimizations and/or heuristics exist, MC techniques suffer from the state-space explosion. To avoid the explosion, so-called *statistical model checking* (SMC) has been proposed – and implemented in several tools such as PRISM [6] or UPPAAL SMC [7] – as a compromise between testing and classical (binary, exhaustive) MC techniques. Simply, SMC is based on monitoring some simulations of the system and their statistical processing to estimate the satisfaction probability of a specified property under some degree of confidence. The SMC approach has been applied to problems that are far beyond the scope of classical MCs and has been widely accepted in various areas such as biology [7], software engineering [8], or system analysis [6].

III. BASIC CONCEPTS

A. Solutions to the IOV Problem

Mechanisms for softening impacts implying from the IOV problem can be classified according to the sub-problem they solve, i.e., the *timing disturbance* or *predictability* problem. While impacts of the timing disturbance problem can be minimized using special instruments such as common (joint) ISR/task priority space [9] or resource access protocols [10], effects of the predictability problem cannot be minimized in such an easy way due to their aperiodical nature. Thus, the rest of this paper is devoted just to the solutions of the latter (predictability) problem.

Solutions to the predictability problem such as [11], [12], [13], [14], [15] are typically designed to bound t_{INT} (or, f_{INT}). In [12] so-called *interrupt limiters* (*ILs*) are designed to prevent RT systems from the problem – they are divided into the two types: *software* (SW) limiters (*SILs*) and *hardware* (HW) limiters (*HILs*). While our previous research activities [16], [17], [18], [19] have been focused on theoretical analysis, architectural and realization details of our adaptive HW/SW solution to HIL, this paper is dedicated just to statistical model checking of existing SIL solutions. Typical parameters w.r.t. SILs are summarized in the Tab. I. SILs can be classified [12] to the *polling*, *strict* and *bursty* sub-types (for their overheads, see Fig. 1), the principle of which follows.

Polling SIL (Fig. 1a) checks periodically (each $t_{arrival}$ units, at the t_{poll} cost) whether an event flag is set or not (for the purpose, either a special on-chip timer able to trigger an INT (t_{tmr}, t_{exp}) is typically utilized, but it can be replaced by a well-tuned polling loop); if the flag is set then a code (such as an ISR or a task) is executed to service the event (t_w) and INTs become disabled for further $t_{arrival}$ units of time. Drawback of the (polling) principle implies directly from its active waiting construction (the CPU is consumed by checking a flag although no event occurs);

The drawback can be removed e.g. by the *strict SIL* principle (Fig. 1b) utilizing the ISR prologue – being executed at the end of the context switch w.r.t. an INT (t_{ictx}) – to disable any further INT (except those from timers) at the t_{flip} cost. After the INT is serviced (t_w), it configures a one-shot timer (t_{adj}) to expire after t_{tmr} units (the expiration cost is t_{exp}). After the expiration, INTs are re-enabled (t_{flip}) to let a further INT to be processed within the consecutive $t_{arrival}$ period. Main disadvantages of the approach can be seen in the following facts: i) INTs are practically doubled as each IRQ triggers the (further) INT utilized to signalize the expiration and ii) INTs are disabled each time an INT is triggered – this degrades reactivity of an RT system.

The disadvantages w.r.t. strict SIL can be minimized using the *bursty SIL* mechanism (Fig. 1c) being configured by the *maximum arrival rate* of INTs ($f_{arrival} = 1/t_{arrival}$) and the *maximum burst size* (N) parameters; the idea of the mechanism is to disable INTs after a burst of N IRQs (where $N \geq 2$) rather than after each IRQ (the strict SIL behavior). A special counter is needed to count (t_{cnt}) the number of triggered INTs (to be serviced within the $t_{arrival}$ interval) until it reaches N; then, INTs are disabled (t_{flip}) to be re-enabled again in the new $t_{arrival}$ period; for that purpose, a timer must be configured again (t_{adj}) to expire (t_{tmr}) at the cost t_{exp} and then, are INTs re-enabled (t_{flip}). This approach represents the actual state of industrial practice applied e.g. in AUTOSAR [15]. Although the (CPU-)load to the (INT-)throughput can be well-tuned comparing to the strict SIL, an extra CPU is still needed especially to compare t_{cnt} to N after an INT occurs as well as to service the start/end of the $t_{arrival}$ period and overload of t_{cnt}.

B. Statistical Model Checking in UPPAAL SMC

UPPAAL [20] is a toolbox primarily designed for formal verification of *real-time* (RT) systems modeled by (a network of) *Timed Automata* (TA) extended with instruments such as typed variables and channel synchronization. SMC extension of UPPAAL (denoted as UPPAAL SMC) has been proposed [21] to avoid the state-space explosion w.r.t. checking

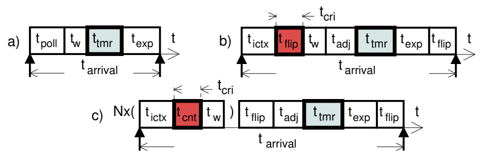

Fig. 1. Overheads w.r.t. the (a) polling, (b) strict and (c) bursty SIL principles. The light-blue (t_{tmr}) boxes represent an action running in parallel with the CPU, red boxes bound the t_{cri} intervals and white boxes represent executions performed by the CPU [19]

properties of an RT system model. The modeling formalism of UPPAAL SMC is based on a stochastic extension of the original TA formalism. On basis of the extension – called *Stochastic Timed Automata* (STA) –, one can validate properties of a given deterministic or stochastic system. In the next paragraph(s), concepts of (S)TA-based modeling are informally outlined.

First of all, it should be noted that a single TA [22] is formed of at least the start state, being represented by two concentric circles (for illustration, see state a in Fig. 2); a TA state is called a *location* too. A transition between two locations (let us say from a to b and denote it by $a \to b$) is represented by an oriented edge from a to b. Transition in Fig. 2a can be made anytime (but the concrete time is unknown), while transition in Fig. 2b – being conditioned by so-called *guard* (where x is a variable of the clock type) – can be made if x is 5 or later, but again: no upper bound is specified for x.

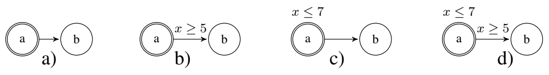

Fig. 2. Illustration to basic TA terms: place, transition, guard, invariant

In Fig. 2c, time of staying in a is limited by so-called *invariant*, i.e., a condition defined for a location; the transition must be made before the invariant becomes false. In Fig. 2d, a guard/invariant combination is utilized to model a transition that can be made if $x \geq 5$, but must be made if $x \leq 7$, i.e., the transition is possible if $5 \leq x \leq 7$. Further TA-related instruments related e.g. to communication via channels, location types etc. are omitted herein because of the limited scope of this paper and no meaning for planned illustrative examples.

The above-mentioned principles as well as related non-deterministic behavior of TAs (such as non-deterministic choice among parallel transitions between the same locations) are refined in STAs by stochastic ones, being briefly illustrated in the next. E.g., weight annotations on locations are extended to model the staying in a location using a probability distribution; e.g., in Fig. 3a, the staying in a (i.e., entering b) is given by the exponential distribution with the rate (λ) set to $\frac{1}{2}$. In Fig. 3b, the probabilistic uniform-distribution choice between $a \to b$ (with probability $\frac{1}{5}$) and $a \to c$ (with probability $\frac{4}{5}$) is modeled. In Fig. 3c, the (so-called *stopwatch*) concept, able to determine the exact time that has elapsed, is illustrated. First, the clock x is reset along with a user-defined (function f) adjustment of the clock $delay$ (during $a \to b$). Then, staying in b cannot take longer than $delay$ units of time, being measured by x while $delay$ is stopped ($delay' == 0$) in b. Finally, $b \to c$ is possible just if x matches $delay$.

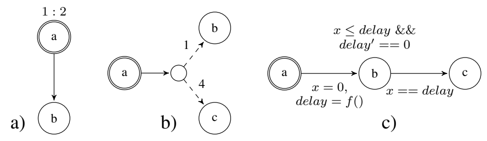

Fig. 3. Illustration to basic STA terms: place, transition, guard, invariant

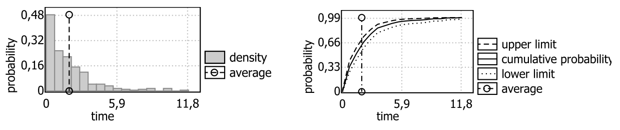

Fig. 4. Illustration to probability of entering b (left) and cumulative probability with confidence intervals (right) of that for the STA model from Fig. 3a

Properties of an STA-based model can be verified (checked) using special queries a user can post in the UPPAAL SMC tool w.r.t. model; e.g., for Fig. 4a one can post the `Pr[<=500000](<> STA.b)` query to get the probability of eventual entering the state b within 500000 units of the simulation time. A possible (probabilistic) result of the query is visualized in Fig. 4. For further examples, please see [21].

IV. Proposed Solution

Our solution to modeling and analysis of the IOV problem and its consequences is composed of i) a collection of STAs, each utilized to express the behavior of a key part of a system – as a CPU, INT subsystem etc. – and ii) a set of queries utilized to check properties of the system and its parts.

A. Behavioral Models

Since all the behavioral models were created in the UPPAAL SMC tool, they are going to be expressed by means of STAs supported by the tool.

First, let a CPU model be presented (see Fig. 5). In the model, the local clock (*cpu_clk*) is utilized to model progress of the local CPU time. Its STA starts in the *S_halt* state; herein, the CPU stays until it is reset – that takes T_RESET units of time; afterwards, it transits to *S_running*. If there is no pending and enabled INT at the moment, i.e. if (irq_pend & irq_mask) $==$ 0, then the CPU randomly select an instruction to be executed – since a particular program is not important for our purpose – and then transits to S_exe_instr. The execution makes the CPU busy for $instr\_delay$ units of time; then, the STA transits to *S_running*.

If a pending and enabled INT is detected ((irq_pend & irq_mask) != 0) while the STA is in *S_halt* then the arbitration of pending INT requests (IRQs) is modeled in S_irq_arb, followed by the INT-related context store (S_ctx_st), service routine (ISR) execution (S_irq_exe) and context restore (S_ctx_rst) phases. Then, the STA transits to *S_halt*.

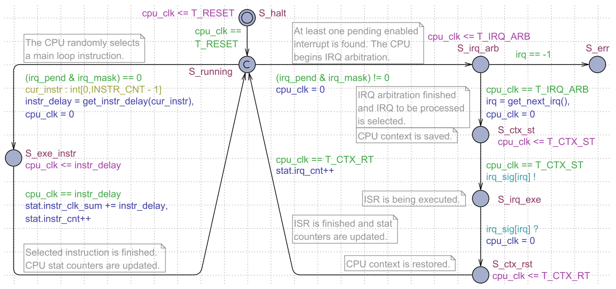

Fig. 5. STA model of a CPU behavior

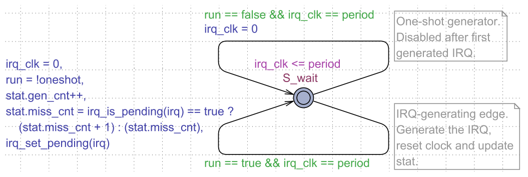
run == false && irq_clk == period
irq_clk = 0
One-shot generator. Disabled after first generated IRQ.
irq_clk = 0,
run = !oneshot,
stat.gen_cnt++,
stat.miss_cnt = irq_is_pending(irq) == true ?
(stat.miss_cnt + 1) : (stat.miss_cnt),
irq_set_pending(irq)
irq_clk <= period
S_wait
IRQ-generating edge. Generate the IRQ, reset clock and update stat.
run == true && irq_clk == period

Fig. 6. STA model of a periodic IRQ generator

For inner synchronization of an INT selected to be serviced (irq), a broadcast channel (named irq_sig; if ! resp. ? follows the name, a message is sent resp. expected via the channel of the name) is utilized to model the start of ($irq\_sig(irq)$!) resp. return from ($irq\_sig(irq)$?) the corresponding ISR; counters such as $stat.instr\_cnt$ or $stat.irq\_cnt$ are utilized to count the numbers of processed instructions or IRQs.

Next STA (see Fig. 6) is utilized to model the behavior of a periodic IRQ generator; multiple instances of this STA can be created to model INTs that can stimulate a system (for a source of aperiodic INTs, an STA can be created too). The STA starts in S_wait that is the only state in the STA. In the model, the local clock (irq_clk) is utilized to model progress of the generator's local time. On basis of the STA, an IRQ can be produced after a predefined delay ($period$) – either in the one-shot or periodic mode; counters such as $stat.gen\_cnt$ or $stat.miss\_cnt$ are utilized to count the number of generated or missed (i.e., unserviced) IRQs.

The STA from Fig. 7 is utilized to model the execution of an ISR. Multiple instances of this STA can be created, each able to model the behavior of the ISR belonging an IRQ. In the model, the local clock (clk) is utilized to model progress of the ISR's local time. The STA starts in S_wait where it waits ($irq\_sig[irq]$?) until an IRQ (identified by irq) occurs. Then, the ISR is executed for $delay$ units of time and transits back to S_{wait} either clearing the IRQ flag automatically or not (that depends on particular INT-source setup). In the model, the counter $stat.time\_sum$ is utilized to measure the CPU time spent by ISR executions.

In the next, STA models for particular SILs are presented – see Fig. 8a for the polling SIL, Fig. 8b for the strict SIL and Fig. 8c for the bursty SIL models. Because of their evident correspondence with Fig. 1 – on basis of which they were created on –, application of the above-mentioned concepts, similarity to the above-mentioned STAs and limited space in this paper, information to the Fig. 8 is present with no special comments.

Just for an interest, the following counters can be found in the models: In Fig. 8a, $stat.poll\_cnt$ resp. $stat.event\_cnt$ is utilized to count how many times an IRQ-flag has been tested resp. how many times the event signaled by the flag has been serviced, In Fig. 8b/c, $stat.irq\_cnt$ is utilized to count how many times an IRQ has occurred. In Fig. 8c, $irqs\_per\_burst$ resp. $stat.irq\_dis\_cnt$ is utilized to count the number of IRQs within the burst-window resp. how many times INTs have been disabled.

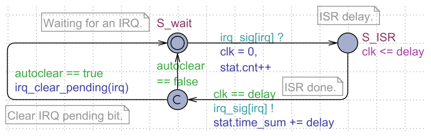

Fig. 7. STA model of an ISR execution

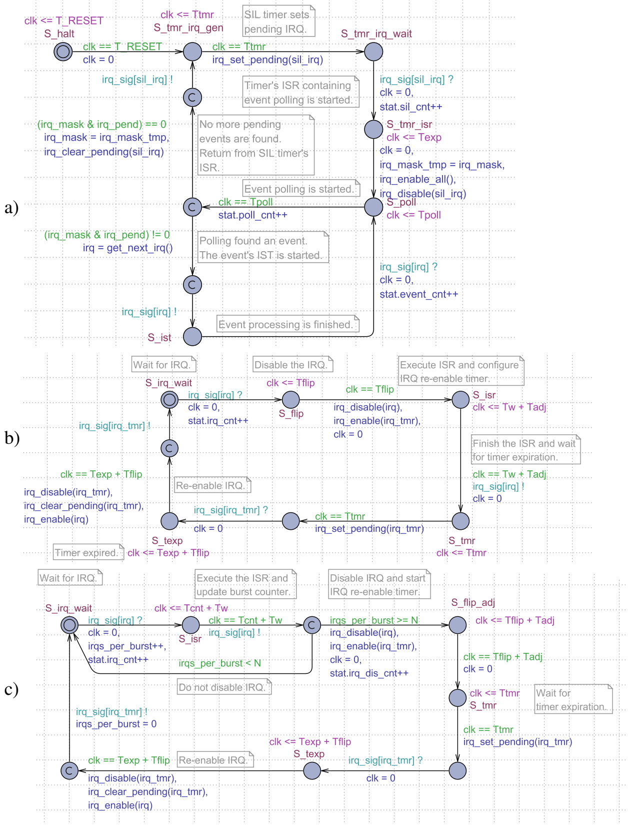

Fig. 8. STA models of a) polling SIL, b) strict SIL and c) bursty SIL

V. Evaluation

To test and demonstrate applicability of our approach, we have decided to perform a set of simulations as well as model checking runs in UPPAAL SMC. For the purpose, we have utilized the following setup for our models:

- each interrupt was expected to arrive at one of the following rates [Hz]: $f_H = \frac{1}{75}$, $f_M = \frac{1}{250}$, $f_L = \frac{1}{700}$ (whereas multiple interrupts could occur at the above-mentioned rates),
- our SIL models were configured by the following default parameters: $t_{flip} = 5$, $t_{adj} = 5$, $t_{exp} = 5$, $t_{cnt} = 6$, $t_w = 5$, $t_{tmr} = 331$.

We have divided the evaluation into several phases, each being focused on particular properties of SILs. A list of the phases, followed by related details and results, can be found in the following text (V-A to V-F). Each of the figures Fig.

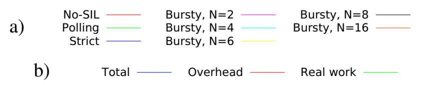

Fig. 9. Legends for the Fig. 10 – 13 (a) resp. for Fig. 14 (b)

10 – 13 (a) resp. for Fig. 14 is visualized for a particular IRQ scenario being expressed in the form x IRQ y where x represents the number of IRQ sources and y identifies their arrival rates, i.e. f_H, f_M or f_L.

A. Phase 1 (Pure IRQ Overhead)

In this phase, the computational overhead (measured as a fraction of the total CPU time) related to IRQs managed by a particular SIL technique has been analyzed (Fig. 10).

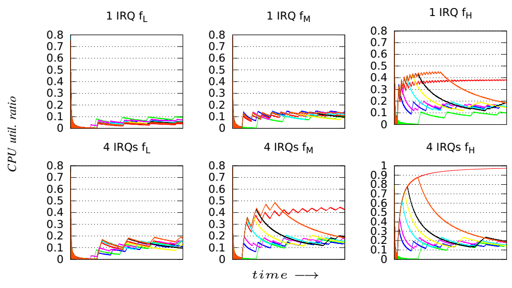

Fig. 10. CPU utilization of IRQ management (excluding SIL overhead).

The overhead includes IRQ arbitration, ISR execution and context manipulation phases; it has been analyzed in first 3000 units of the simulation time, using a query of the `simulate 1 [<= 3000] (t_total - t_instr - t_sil) / (t_total + 1)` form, where `t_total` represents the total CPU time being consumed, out of which `t_instr` resp. `t_sil` is the time for processing instructions from *main* resp. for SIL-related computations. It is evident that without a SIL, CPU can become overloaded because of excessive IRQs.

B. Phase 2 (Ratio of Missed IRQs)

Aim of this phase was to analyze an impact of particular SIL technique to the ratio of missed IRQs (Fig. 11) using the `simulate 1 [<= 3000] irq_missed / (irq_total + 1.0)` query, where `irq_total` represents the total number of IRQs while `irq_missed` the number of missed IRQs. By a missed IRQ we mean one that occurs when the previous one (from the same IRQ source) is still pending, i.e., it has not been serviced yet. It can be seen that more complex SILs (such as bursty with N=16) are able to achieve smaller ratio than simpler SILs (such as polling).

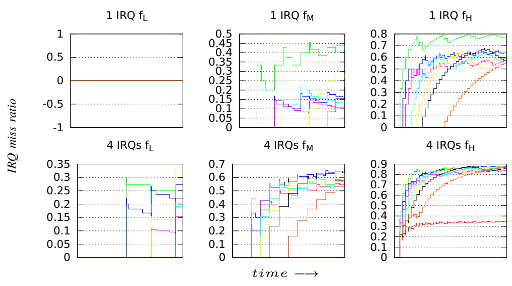

Fig. 11. Ratio of missed IRQs to all IRQs

C. Phase 3 (Pure SIL Overhead)

In the next, the portion of the CPU time used to perform computations related to particular SILs has been analyzed (Fig. 12) using the query `simulate 1 [<= 3000] t_sil / (t_total + 1)`.

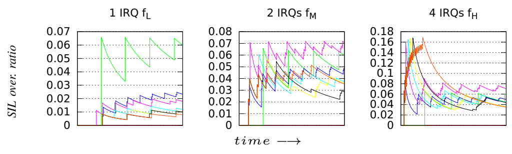

Fig. 12. CPU utilization of particular SIL approaches

While overhead related to polling SIL oscillates about 0.05, overhead of remaining SILs depends on their $f_{arrival}$.

D. Phase 4 (Ratio of Forwarded IRQs)

This phase was dedicated to analyzing the ratio of IRQs outgoing from a SIL (`irq_sil`) to IRQs entering the SIL (`irq_total`), Fig. 13. The query was `simulate 1 [<= 3000] irq_sil / (irq_total + 1.0)`.

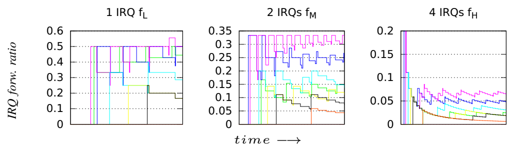

Fig. 13. Ratio of IRQs exiting a SIL to IRQs entering the SIL

E. Phase 5 (CPU utilizations)

Aim of this phase was to analyze a fraction of the CPU time being spent by useful resp. IRQ-related work across various SIL approaches; the query was `simulate 1 [<= 3000] t_total, (t_instr / t_total + 1), (t_total - t_instr) / (t_total + 1))`. In the Fig. 14 (for the legend, see Fig. 9b), it can be seen that while the overhead represents 50 % and more of the useful (real) work for the polling and strict SILs, it decreases with growing N for the bursty SIL.

F. Phase 6 (SMC Queries)

- Probability that the CPU spends at least 75 % of its time by executing instructions unrelated to interrupts could be checked by the `Pr ( [][2940,3000] ((t_instr / (t_total + 1)) >= 0.75))` query, result of which is close to 100 % except of the no-SIL approach with $4f_M$ and $1/2/4f_H$.
- Probability that the CPU can spend more time by executing interrupts than by executing the main loop can be checked by the `Pr ( [][2940,3000] ((t_instr / (t_total + 1)) <= 0.5))` query, result of which is close to 0 except of the no-SIL approach with $2/4f_H$.

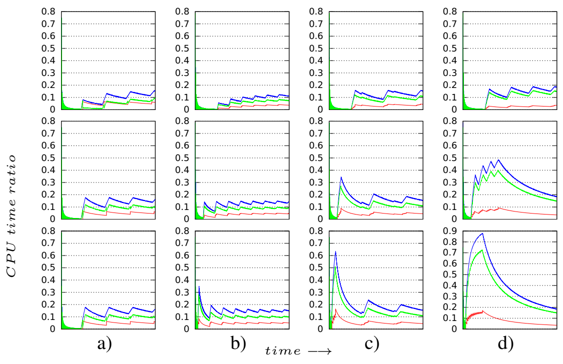

Fig. 14. Comparison of the total CPU time being consumed and its fractions, i.e., IRQ overhead and useful work for the following SILs: a) polling, b) strict, c) bursty, N=4 d) bursty, N=16. Results for the 4 $IRQ\ f_H$ scenario.

- Probability that the CPU can spend more than 75 % of its time by executing interrupts can be checked by the `Pr ( [][2940,3000] ((t_instr / (t_total + 1)) <= 0.15))` query, result of which is close to 0 except of the no-SIL approach with $4f_H$.

VI. Conclusion

In the paper, a novel method of modeling and analysis of SIL-based INT managers for event-driven embedded systems has been presented. The method can be seen as an alternative to classical, ad-hoc analytic approaches utilized in existing works. Our method is built over STAs and SMC instruments and allows one to study the system behavior under particular configuration/scenarios w.r.t. INT subsystem. In near future, we plan to extend our method by modeling HILs and ILs in further layers of the system such as firmware, device drivers or operating system.

Acknowledgment

This work 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. This work was supported by Brno University of Technology under number FIT-S-14-2297 too.

References

[1] A. M. K. Cheng, *Real-Time Systems, Scheduling, Analysis, and Verification*, John Wiley & Sons, Hoboken NJ, United States, 552 p., 2002, isbn: 978-0-471-18406-5.

[2] H. Kopetz, *On the Fault Hypothesis for a Safety-Critical Real-Time System*, Automotive Software – Connected Services in Mobile Networks, Lecture Notes in Computer Science vol. 4147, 2006, no. 1, pp. 31–42, doi: 10.1007/11823063_3.

[3] C. Baier and J.-P. Katoen, *Principles of Model Checking*, Representation and Mind Series. MIT Press, 975 p., 2008, isbn: 9780262026499

[4] G. J. Holzmann, *The Model Checker SPIN*, IEEE Transactions on Software Engineering, vol. 23, 1997, pp. 279–295.

[5] K. L. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*, Ph.D. Dissertation, Pittsburgh, PA, USA, 1992, uMI Order No. GAX92-24209. [Online]. Available from *http://www.kenmcmil.com/pubs/thesis.pdf*

[6] M. Kwiatkowska, G. Norman, and D. Parker, *PRISM: Probabilistic Model Checking for Performance and Reliability Analysis*, ACM SIGMETRICS Performance Evaluation Review, vol. 36, no. 4, Mar. 2009, pp. 40–45, doi: 10.1145/1530873.1530882

[7] A. David, K. G. Larsen, A. Legay, M. Mikucionis, D. B. Poulsen, and S. Sedwards, *Statistical Model Checking for Biological Systems*, International Journal on Software Tools for Technology Transfer, vol. 17, no. 3, Jun. 2015, pp. 351–367, doi: 10.1007/s10009-014-0323-4

[8] R. Calinescu, C. Ghezzi, K. Johnson, M. Pezze, Y. Rafiq, and G. Tamburrelli, *Formal Verification with Confidence Intervals to Establish Quality of Service Properties of Software Systems*, IEEE Transactions on Reliability, vol. PP, no. 99, 2015, pp. 1–19.

[9] Lynx. *Lynx Software Technologies Patented Technology Speeds Handling of Hardware Events*, 2016, [online]. Available: *http://www.lynx.com/whitepaper/lynx-software-technologies-patented-technology-speeds-handling-of-hardware-events/*

[10] F. Cottet, J. Delacroix, C. Kaiser, Z. Mammeri, *Scheduling in Real-Time Systems*, John Wiley & Sons, Hoboken NJ, United States, 2002, isbn: 978-0-470-84766-4.

[11] L. E. L. del Foyo, P. Mejia-Alvarez, *Custom Interrupt Management for Real-time and Embedded System Kernels*, in: Proceedings of the Embedded Real-Time Systems Implementation Workshop at the 25th IEEE International Real-Time Systems Symposium, IEEE Computer Society, Washington DC, United States, p. 8, 2004, doi: 10.1.1.100.7025.

[12] J. Regehr, U. Duongsaa, *Preventing Interrupt Overload*, in: Proceedings of the ACM SIGPLAN/SIGBED Conference On Languages, Compilers, And Tools For Embedded Systems, ACM, New York, United States, pp. 50–58, 2005, doi: 10.1145/1070891.1065918.

[13] J. Regehr, *Safe and Structured use of Interrupts in Real-Time and Embedded Software*, in: Handbook of Real-Time and Embedded Systems, I. Lee, J. Y.-T. Leung, and S. H. Son Eds., 1st Edition, Chapman & Hall/CRC, Boca Raton, FL 33487, United States, pp. 16–1 – 16–12, 2007, doi: 10.1.1.79.4651.

[14] R. Pellizzoni, *Predictable And Monitored Execution For Cots-Based Real-Time Embedded Systems*, Ph.D. thesis, University of Illinois at Urbana-Champaign, Bonn, Germany, 2010.

[15] Automotive Open System Architecture GbR (AUTOSAR). *Specification of Operating System*. Technical report, 2016 [online]. Available: *http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/system-services/standard/AUTOSAR_SWS_OS.pdf*

[16] J. Strnadel, Monitoring-Driven HW/SW Interrupt Overload Prevention for Embedded Real-Time Systems, in: Proceedings of the 15th International IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), IEEE CS, US, pp. 121 – 126, 2012, doi: 10.1109/DDECS.2012.6219037.

[17] J. Strnadel, *Load-Adaptive Monitor-Driven Hardware for Preventing Embedded Real-Time Systems from Overloads Caused by Excessive Interrupt Rates*, in: Architecture of Computing Systems - ARCS 2013, Lecture Notes in Computer Science, Springer, pp. 98 – 109, 2013, doi: 10.1007/978-3-642-36424-2_9.

[18] J. Strnadel, *On Design of Priority-Driven Load-Adaptive Monitoring-Based Hardware for Managing Interrupts in Embedded Event-Triggered Real-Time Systems*, in: Proceedings of the 16th International IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), IEEE CS, pp. 24 – 29, 2013, doi: 10.1109/DDECS.2013.6549783.

[19] J. Strnadel, *Comparison of Generally Applicable Mechanisms for Preventing Embedded Event-Driven Real-Time Systems from Interrupt Overloads*, in: Proceedings of the 4th Eastern European Regional Conference on the Engineering of Computer Based Systems (ECBS-EERC), IEEE CS, pp. 39 – 44, 2015, doi: 10.1109/ECBS-EERC.2015.15.

[20] G. Behrmann, A. David, and K. Larsen, *A Tutorial on UPPAAL*, In Formal Methods for the Design of Real-Time Systems, Lecture Notes in Computer Science, Springer Berlin Heidelberg, vol. 3185, 2004, pp. 200–236, doi: 10.1007/978-3-540-30080-9_7

[21] A. David, K. Larsen, A. Legay, M. Mikuionis, and D. Poulsen, *UPPAAL SMC Tutorial*, International Journal on Software Tools for Technology Transfer, vol. 17, no. 4, 2015, pp. 397–415, doi: 10.1007/s10009-014-0361-y

[22] R. Alur and D. L. Dill, *A theory of timed automata*, Theoretical Computer Sci., vol. 126, no. 2, Apr. 1994, pp. 183–235, doi: 10.1016/0304-3975(94)90010-8