UDK 004.4’42

# An Exact Schedulability Test for Real-time Systems with an Abstract Scheduler

*Garanina N.O. (A.P. Ershov Institute of Informatics Systems SB RAS)*

In this paper, we formally describe real-time systems with an abstract scheduler using Kripke structures. This formalization allows us to refine the abstract scheduler in its terms. We illustrate this approach with a non-preemptive global fixed priority scheduler (NE-GFP). We also formulate a safety property for real time systems using linear temporal logic LTL. We implement our formalization of real-time systems with a NE-GFP scheduler in language Promela used in the SPIN verification tool and make experiments for proving or disproving the safety property to evaluate the effectiveness of our approach.

*Keywords*: real-time systems, exact schedulability test, Kripke structures, model checking, Promela, SPIN

## 1. Introduction

Classical real-time systems proposed in [11] are a set of tasks that occur from time to time with period $P$ or more, have a deadline $D$ and execution time $C$. In the modern world, such systems arise literally at every step – these are embedded systems, and Internet of Things systems, and technological processes, business processes, and automatic control systems in the automotive industry, avionics, space industry, etc. These tasks can use the resources of one or more processors. As a rule, there are significantly fewer processors than tasks, so the question of scheduling their execution naturally arises. There are different ways to specify schedulers depending on the subject domain. For example, in some cases it is possible to allow interruption of low-priority tasks, while in other cases such an approach can lead to a system failure. The main question for real time systems described in terms of execution time, deadline and periodicity is the question of safety: is it true that in a given set of tasks with fixed features and a given scheduler, no task will ever miss its deadline?

This problem has been solved for many years for various real time systems and various schedulers. For systems with only one processor, the problem has been studied quite well [11]. However, for multiprocessor systems, the problem of a very large number of task behavior variants arises, and exact methods for checking the safety property turn out to be poorly applicable in an explicit form. Approaches based on near-optimal scheduling have been proposed [2], but exact schedulers are preferable and exact approaches continue to develop [3, 4, 8, 12, 17].

In addition to the development of specialized methods for precise schedulability verification, there is a small number of works using general formal methods for analyzing programs and systems. In particular, [9, 16] proposes the representation of static-priority global multiprocessor scheduling and non-Preemptive self-suspending real-time tasks using timed automata, which make the verification of the safety of real time systems rather resource-consuming. In [13], a special case of a real time system was modeled in the Promela language of the SPIN verification tool [7], and in [15] authors present a Promela model for real-time system on a single-processor. In [6], graph games is used to make easier the reachability problem in exact shedulability test.

In our work, we formally represent real-time systems with an abstract scheduler as Kripke structures, which are used to verify parallel and distributed systems using model checking. Such formalization allows us to refine the abstract scheduler in terms of the proposed Kripke structure and obtain specific real-time systems. We present an example of refining the abstract scheduler to a non-preemptive GFP scheduler. In addition, we formulate the safety property in terms of liner temporal logic LTL [5]. We model our formalization of real-time systems with a non-preemptive GFP scheduler in the Promela language of the SPIN and conduct a series of experiments to prove or disprove the safety property to clarify the performance of our method.

The rest of the paper has the following structure. In Section 2, we recall base definitions of real-time systems and scheduling and formalise real-time systems as Kripke structures. Section 3 considers features of the Promela and describes the Promela model for a real-time system with the non-preemptive global fixed priority scheduler. The conclusion is given in Section 4.

## 2. A Real-time Kripke Structure

We consider that *a real-time system* is a set of tasks $T = (T_1, ..., T_n)$, where each *task* $T_i = (C_i, D_i, P_i)$ has *an execution time* $C_i$, *a relative deadline* $D_i$, and *a minimum period* $P_i$. Each task $T_i \in T$ can generate a potentially infinite number of jobs, every of which requires $C_i$ units of time. These jobs must be completed before $D_i$ time units after the release time. Release time instants are separated by at least $P_i$ time units. If there are no other restrictions on jobs' releases, these tasks are refereed as *sporadic tasks*. In this paper, we study the base case of real-time systems in which all task parameters are integers. All jobs are executed on $m$ processors. If the number processors is less then number of tasks ($m < n$), we need a *scheduler* to decide which task's job to run next. We assume that scheduling decisions are taken at discrete time instants starting from 0. The *schedulability test problem* is to detect if every jobs of every task in the real-time system is finished before its deadline. For the rest of the paper we fix above real-time system $T$.

We would like to deal with real-time system $T$ as with a finite model to apply model checking techniques. Inspired by the paper [1], we introduce current values of task parameters as follows. For every task $i \in T$, let tuple $s_i = (i, C'_i, D'_i, P'_i, rel_i, bad_i)$ be a state of task $i$, where

- $C'_i \leq C_i$ is time left until a job of this task ends;
- $D'_i \leq D_i$ is time until the deadline of a job of this task;
- $P'_i \leq P_i$ is time until the the next admissible release of a job of this task;
- $rel_i \in \mathbb{B}$ is boolean variable which marks a job release: it becomes $true$ when task $i$ release a job, and it becomes $false$ when the job is completed.
- $bad_i \in \mathbb{B}$ is boolean variable which marks that a job misses the deadline soon: it is $false$ if $C'_i \leq D'_i$, and it becomes $true$ otherwise.

For brevity, we refer to boolean constants $true$ and $false$ as **1** and **0**, respectively. For representing the processors' load, we also introduce variable *busy*: a number of jobs currently executing at some moment ($busy \leq m$).

A scheduler must exactly define conditions under which each task can execute its job and conditions for permitting a job execution. There are many types of schedulers, for example,

- Global fixed priority (GPF). The set of tasks are ordered: $T_1$ has the highest priority, $T_n$ has the lowest priority and a major task job has priority over a minor task job;
- Earliest deadline first (EDF). The task with the closest deadline has the highest priority;
- Non-preemptive. No job can be interrupted by other job (even from a major task);
- Preemptive. A job can be interrupted by other job from a major task.

In our Kripke structure, for modeling an abstract scheduler we use an abstract predicate $go(i, s, t)$ that is $true$ if task $i$ can execute the job at state $t$ which is a successor of state $s$, and $false$ otherwise. Further, we specify $go(i, s, t)$ for the non-preemptive GPF scheduler.

For calculating the change of processor load *busy*, we also need to compute a modification number – a quantity of tasks which jobs are just finished or released and just admitted to execution by the scheduler. For this, we introduce predicate $fin(i, s)$ for just finished jobs that is $true$ if task $i$ finishes its job in state $s$, i.e. $s.C'_i = 0$, and $false$ otherwise. To compute the modification number, we treat $go(i, s, t)$ and $fin(i, s)$ as integer numbers (1 for $true$ and 0 for $false$).

Let $Prop$ be a set of propositions consisting of arithmetic comparisons of current values of task parameters and propositions about a number of running jobs.

Now, we define real-time system $T$ with an abstract scheduler as *a real-time structure* $M^T = (S^T, s_0^T, R^T, L^T)$, where

- the finite set of states $S^T = \prod_{i=1}^{n}(\{i\} \times [0..C_i] \times [0..D_i] \times [0..P_i] \times \mathbb{B} \times \mathbb{B}) \times [0..m]$; for global state $s \in S^T$, $s_i = (i, C_i', D_i', P_i', rel_i, bad_i)$ is *a projection of* $s$ *on task* $i$, and $s.C_i'$, $s.D_i'$, $s.P_i'$, $s.rel_i$, $s.bad_i$ and $s.busy$ are projections of $s$ on its components;
- the initial state $s_0^T = \prod_{i=1}^{n}\{(i, C_i, D_i, P_i, \mathbf{0}, \mathbf{0})\} \times \{0\}$;
- the total transition relation $R^T \in S^T \times S^T$ is defined by composing relations for $i$-task projections of global states $s$ and $t$. $(s,t) \in R^T$ iff $t.busy = s.busy + \sum_{i=1}^{n}(go(i,s,t) - fin(i,s))$ and one of the following points holds:
  1. $s_i = (i, C_i, D_i, P_i, \mathbf{0}, \mathbf{0})$, and
     (a) $t_i = (i, C_i, D_i, P_i, \mathbf{0}, \mathbf{0})$ – task $i$ do nothing;
     (b) $t_i = (i, C_i, D_i - 1, P_i - 1, \mathbf{1}, \mathbf{0})$, and $\neg go(i,s,t)$ – task $i$ releases the job and it is not started;
     (c) $t_i = (i, C_i - 1, D_i - 1, P_i - 1, \mathbf{1}, \mathbf{0})$, and $go(i,s,t)$ – task $i$ releases the job and it is immediately started;
  2. $s_i = (i, C_i', D_i', P_i', \mathbf{1}, \mathbf{0})$, $t_i = (i, C_i', D_i' - 1, P_i' - 1, \mathbf{1}, \mathbf{0})$ with $s.D_i' > 0$, and $\neg go(i,s,t)$ – task $i$ is waiting for permitting its job;
  3. $s_i = (i, C_i', D_i', P_i', \mathbf{1}, \mathbf{0})$, $t_i = (i, C_i' - 1, D_i' - 1, P_i' - 1, \mathbf{1}, \mathbf{0})$ with $0 < s.C_i' < C_i$, $0 < s.D_i' < D_i$, $s.C_i' \leq s.D_i'$, and $go(i,s,t)$ – task $i$ executes a job;
  4. $s_i = (i, 0, D_i', P_i', \mathbf{1}, \mathbf{0})$, or $s_i = (i, C_i, D_i', P_i', \mathbf{0}, \mathbf{0})$, or $s_i = (i, C_i, D_i, P_i', \mathbf{0}, \mathbf{0})$, and
     (a) if $s.P_i' = 0$
        i. $t_i = (i, C_i - 1, D_i - 1, P_i - 1, \mathbf{1}, \mathbf{0})$, and $go(i,s,t)$ – task $i$ finishes its job normally, and it is releasing and starting its job at this moment;
        ii. $t_i = (i, C_i, D_i - 1, P_i - 1, \mathbf{1}, \mathbf{0})$, and $\neg go(i,s,t)$ – task $i$ finishes its job normally, and it is releasing and not starting its job at this moment;
        iii. $t_i = (i, C_i, D_i, P_i, \mathbf{0}, \mathbf{0})$ – task $i$ finishes its job normally, and it is not releasing;
     (b) if $s.D_i' = 0$ and $s.P_i' > 0$ then $t_i = (i, C_i, D_i, P_i' - 1, \mathbf{0}, \mathbf{0})$ – task $i$ finishes its job normally and waiting for the next release;
     (c) if $s.D_i' > 0$ then $t_i = (i, C_i, D_i' - 1, P_i' - 1, \mathbf{0}, \mathbf{0})$ – task $i$ finishes its job normally and waiting for the next release;

5. $s_i = (i, C'_i, C'_i - 1, P'_i, \mathbf{1}, \mathbf{0})$ and $t_i = (i, C'_i, C'_i - 1, P'_i, \mathbf{0}, \mathbf{1})$ – a job of task $i$ will miss the deadline definitely;
6. $s_i = (i, C'_i, C'_i - 1, P'_i, \mathbf{0}, \mathbf{1})$ and $t_i = (i, C'_i, C'_i - 1, P'_i, \mathbf{0}, \mathbf{1})$ – task $i$ is in the bad state forever.

- evaluation function $L : Prop \longrightarrow 2^{S^T}$ is standard: it assigns comparison propositions to those states in which they are true.

To refine the abstract scheduler, we need to specify predicate $go(i, s, t)$. This predicate for all types of schedulers uses information about the load of processors *busy* and the system tasks: their parameters, priorities, time from releases, time until deadlines, etc. All this information is available at the system states $s$ and $t$, hence $go(i, s, t)$ can be formulated in terms of our real-time Kripke structures. For example, in the case of non-preemptive global fixed priority scheduler, predicate $go(i, s, t) = (|Maj_i| + s.busy < m)$, where $Maj_i = \{j \in [1..n] \mid j < i \wedge ((s.rel_j = \mathbf{1} \wedge s.C_j = C_j) \vee (s.rel_j = \mathbf{0} \wedge t.rel_j = \mathbf{1}))\}$ is a set of released jobs with higher priority that have not yet started.

We mark a state of real-time system $T$ with $D'_i = C'_i - 1$ for some task $i$ as *a bad state* because in this case there is no time left to meet the deadline. Bad states in the Kripke structure $M^T$ is a set $Bad\_States = \{s \in S^T \mid \exists i \in [1..n] : \ s.bad_i = \mathbf{1}\}$. Let proposition *bad* be *true* in state $s$ if $\bigvee_{i=0}^{n}(s.bad_i = \mathbf{1})$ is *true*, and be $false$ otherwise. Hence, the exact schedulability test for real-time system $T$ is to check if LTL formula $\Phi_T = \mathbf{G}(\neg bad)$ is satisfiable in $M^T$: the real-time systems never reaches a state in which some task in its bad state.

## 3. A Real-time System with a Non-preemptive Global Fixed Priority Scheduler in Promela

In this section, we describe implementation of real-time structure $M^T$ for real-time system $T$ in Promela – an input language of model checker SPIN. Promela language is used to describe parallel communicating processes based on the CSP formalism [10]. Promela program consists of parallel processes communicating through channels or shared variables. The execution of a set of Promela parallel processes exploits the interleaving semantics. Interleaving can be bounded by `atomic` and `d_step` statements, which permit interruption of specified sequence of process actions. The Promela language includes blocking control-flow statements `if` and `do`. Promela model can be verified by model checker SPIN against LTL requirements, hence it assumes only finite types for model variables. Our real-time structure $M^T$ has a finite number of states and its representation in Promela does not require data abstractions.

We model $M^T$ in Promela to perform exact schedulability test, i.e. to detect if every jobs of every task is completed before its deadline. We specify an abstract scheduler of $M^T$ as a non-preemptive global fixed priority scheduler. For simplicity, we also set the period $P_i$ be equal to $D_i$ for every $i$. Below, we give the implementation details skipping some code for brevity.

The initial Promela process starts the scheduler and `NumPrc` number of tasks with synthetic parameters $C_i$ and $D_i$ naming each task by its number $i$. Specific non-synthetic real-time systems can be modelled by operator `run task(i, C_i, D_i)` for particular $C_i$ and $D_i$.

```
init{
atomic{
  run scheduler();
  for (i : 0 .. NumPrc-1){ run task(i, i+1, 2*(i+2)); }}
}
```

Listing 1: Promela-process for launching the system

Following the definition of $M^T$, we describe tasks that release and execute their jobs as Promela processes that implement transition relation $R^T$ almost directly. For example, the Promela code for Point 2 of the definition is given in Listing 2 (lines 8–20). In green comments of this listing, we give explanations of the task actions. If a task fails its deadline, it sets special boolean variable `BAD` to *true* (line 26).

```
 1  proctype task (byte me; byte C; byte D) {
 2   bool go = false;       // predicate go(i,s,t) computed by the scheduler
 3   bool release = false; // a variable for marking job realeases
 4   byte C_cur = C;        // C' -- time left until a job ends
 5   byte D_cur = D;        // D' -- time until the deadline
 6  do
 7  :: atomic{ C_cur == C && D_cur == D && !release -> // can release a job
 8     if
 9     :: release = true;    // releases a job
10        request ! me       // requests the scheduler for job execution
11        work++;            // action for synchronization step
12        responce[me] ? go // receives the responce from the scheduler
13        if
14        ::  go ->
15            C_cur--; D_cur--; busy++; // executes the job, point 2.c
16        :: else -> D_cur--;            // waits, point 2.b
17        fi
18     :: work++; // do nothing, point 2.a
19     fi
20     responce[me] ? _ // action for synchronization step
21     }
22  :: atomic{ C_cur == C && D_cur > 0 && D_cur < D && C_cur <= D_cur && release
       -> // release and not started
23    ...
24  :: atomic{ C_cur > D_cur && release -> // fails deadline
25     BAD = true; // point 6
26     break;
27     }
28  od
29  }
```

Listing 2: Promela-process for tasks

In our model, the scheduler gives permission for job executions and synchronises Promela processes for tasks to prevent unwanted interleaving. First, the scheduler collects requests for job executions in boolean array `req_prc` (lines 4–6). Second, following non-preemptive and GFP settings, the scheduler knowing the number of free processors (line 10) scans the array of requests from its first element in the order (line 11). If it finds that the next in line task $i$ asks for its job execution and there are free processors (line 13), it sends signal *go* to this task (lines 14) and decreases the number of free processors (lines 15). When the number of free processors becomes zero (line 16), the scheduler sends the rest of requesting tasks the message with the prohibition of its job execution (line 17). After scheduling actions, this Promela process performs synchronization between tasks, resetting the number of `work` tasks executing their step and sending them permission to continue working (line 24). The scheduler's scanning, communication and synchronisation actions are placed in an `atomic` block, the sequence of actions of which is treated by SPIN as a single computational step. This placing provides synchronisation between the tasks and the scheduler and decreases the model checking computational complexity.

```
 1 proctype scheduler(){
 2 ...
 3 do
 4 :: atomic{ work > 0 && !end -> ...
 5    request ? num; req_prc[num] = true; // collects requests
 6    ... }
 7 :: atomic{ work == NumPrc && empty(request) ->  ...
 8    if
 9    :: NumReqs > 0 ->      // there are some requests
10       free = MAX - busy; // number of free processors
11       for (i : 0 .. NumPrc - 1){
12       if
13       :: req_prc[i] && free != 0 ->
14          responce[i] ! true; ...   // go!
15          free--;
16       :: req_prc[i] && free == 0 ->
17          responce[i] ! false; ... // don't go!
18        :: else -> skip;
19        fi
20        } ...
21    :: else -> skip; // no requests
22    fi
23    // actions for synchronization step:
24    work = 0; for (i : 0 .. NumPrc - 1){ responce[i] ! true }
25    }
26 :: BAD -> break;
27 od
28 }
```

Listing 3: Promela-process for the scheduler

To exactly test a real-time system for schedulability, we check LTL formula `[]!BAD` in SPIN tool. If this formula is satisfiable in the real-time system, no its task misses its deadline. We made some experiments with SPIN model checker (version 6.5.1), on a CPU with 4 cores and 8 GB RAM. We definitely prove the safety of our real-time system for 5 tasks and 4 processors. When we increase the number of tasks by one, the SPIN verification time becomes too long (more then an hour) for proving safety. But if a real-time system is unsafe, SPIN discovers it very quickly: for 40 tasks and 20 processors it takes only 0.074 seconds to find the counterexample (the sequence of task releases leading to missing some deadline).

The complete Promela model for real-time systems with the non-preemptive global fixed priority scheduler is located in the repository [18].

## 4. Conclusion

In this paper, we formalise real-time systems with an abstract scheduler as a Kripke structure – the real-time Kripke structure. We show that this abstract scheduler can be refined in terms of a real-time Kripke structure. In particular, we present a non-preemptive global fixed priority scheduler by specifying conditions under which tasks' jobs are started.

In future, we plan to refine the abstract scheduler for other types of schedulers, i.e. preemptive GPF scheduler, preemptive and non-preemptive EDF scheduler, etc. These specifications can used for modeling real-time systems in input languages of model checkers in order to perform exact schedulability tests and for teaching. We also plan to use our formalisation of real time systems to develop new effective algorithms for the exact schedulability test, e.g. backtracking based algorithms.

## References

1. V. Bonifaci and A. Marchetti-Spaccamela, Feasibility analysis of sporadic real-time multiprocessor task systems. // Algorithmica, 2012.
2. B. B. Brandenburg and M. Gul, Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations. // Proceedings of Real-Time Systems Symposium (RTSS). IEEE, 2016.
3. Artem Burmyakov, Enrico Bini, and Eduardo Tovar. 2015. An exact schedulability test for global FP using state space pruning. // In Proceedings of the 23rd International Conference on Real Time and Networks Systems (RTNS '15). Association for Computing Machinery, New York, NY, USA, 225–234. https://doi.org/10.1145/2834848.2834877
4. A. Burmyakov, E. Bini and C. -G. Lee, Towards a Tractable Exact Test for Global Multiprocessor Fixed Priority Scheduling. // in IEEE Transactions on Computers, vol. 71, no.

11, pp. 2955-2967, 1 Nov. 2022, doi: 10.1109/TC.2022.3142540.

5. Clarke E.M., Henzinger T. A., Veith H., and Bloem R. Handbook of model checking. Springer, 2018. 1210 p.
6. G. Geeraerts, J. Goossens and T. -V. -A. Nguyen, A Backward Algorithm for the Multiprocessor Online Feasibility of Sporadic Tasks. // 2017 17th International Conference on Application of Concurrency to System Design (ACSD), Zaragoza, Spain, 2017, pp. 116-125, doi: 10.1109/ACSD.2017.9.
7. Holzmann G. J. The SPIN Model Checker, Primer and Reference Manual. Addison-Wesley: 2003.
8. Pourya Gohari, Jeroen Voeten, and Mitra Nasri. Reachability-Based Response-Time Analysis of Preemptive Tasks Under Global Scheduling. In 36th Euromicro Conference on Real-Time Systems (ECRTS 2024). Leibniz International Proceedings in Informatics (LIPIcs), Volume 298, pp. 3:1-3:24, Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2024) https://doi.org/10.4230/LIPIcs.ECRTS.2024.3
9. Guan, N., Gu, Z., Deng, Q., Gao, S., Yu, G. Exact Schedulability Analysis for Static-Priority Global Multiprocessor Scheduling Using Model-Checking. // Software Technologies for Embedded and Ubiquitous Systems. SEUS 2007. Lecture Notes in Computer Science, vol 4761. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-75664-4_26
10. Hoare C. A. R. Communicating sequential processes. Prentice-Hall: 1985.
11. Jane W. S. Liu, Real-Time Systems. Prentice Hall, 2001. 610 p.
12. Ranjha, S., Gohari, P., Nelissen, G. et al. Partial-order reduction in reachability-based response-time analyses of limited-preemptive DAG tasks. Real-Time Syst 59, 201–255 (2023). https://doi.org/10.1007/s11241-023-09398-x
13. Staroletov, S. A Formal Model of a Partitioned Real-Time Operating System in Promela. // Proceedings of the Institute for System Programming of the RAS (2020): Volume 32, Issue 6, Pages 49–66, DOI: https://doi.org/10.15514//ISPRAS-2020-32(6)-4
14. Sun, Y., Lipari, G. A pre-order relation for exact schedulability test of sporadic tasks on multiprocessor Global Fixed-Priority scheduling. // Real-Time Syst 52, 323–355 (2016). https://doi.org/10.1007/s11241-015-9245-9
15. P. Sukvanicha, A. Thongtak and W. Vatanawood, Formalizing Real-Time Embedded System into Promela // MATEC Web of Conferences 35 03003 (2015) DOI: 10.1051/matecconf/20153503003

16. B. Yalcinkaya, M. Nasri and B. B. Brandenburg, An Exact Schedulability Test for Non-Preemptive Self-Suspending Real-Time Tasks. // 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), Florence, Italy, 2019, pp. 1228-1233, doi: 10.23919/DATE.2019.8715111.
17. Q. Zhou, G. Li, C. Zhou and J. Li, Limited Busy Periods in Response Time Analysis for Tasks Under Global EDF Scheduling. // in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 40, no. 2, pp. 232-245, Feb. 2021, doi: 10.1109/TCAD.2020.2994265
18. URL: https://github.com/GaraninaN/RealTimeSystems/blob/main/np-gfp-rts.pml, 2024.