# Modeling Single Event Upsets in UPPAAL SMC for Real-time DAG Scheduling

Lukas Miedema
l.miedema@uva.nl

Benjamin Rouxel
b.rouxel@uva.nl

Clemens Grelck
c.grelck@uva.nl

## Abstract

Real-time cyber-physical systems have become ubiquitous. As such systems are often safety-critical, designers must include mitigations against various types of hardware faults, including *Single Event Upsets* (SEU). SEUs are transient faults that only momentarily affect a single processor, after which the processor returns to normal operation. The effect of a SEU may manifest itself in the job running on that processor as incorrect output.

We present a new approach for analyzing schedulability using UPPAAL in *Stochastic Model Checking* (SMC) mode while including mitigations for SEUs using job restarts into the scheduler. By restarting a job after experiencing a SEU, the fault-free response of the job is delayed, potentially multiple times. We propose a method that informs the system designer about the distribution of the applications makespan, including the probability that a given deadline will be met in the presence of SEUs.

## Keywords

single upset event, soft error, transient fault tolerance, UPPAAL, timed automaton, DAG scheduling

## 1 Introduction

In this work we focus on online scheduling while dealing with *Single Upset Events* (SEUs) on symmetric multi-core platforms. SEUs, also called *soft errors* or *transient faults*, are a type of fault which only momentarily affects a single processor and does not leave it in a degraded state. Such faults may, for example, be caused by cosmic rays [8]. The impact of a SEU is that the output of the running code may be erroneous, which can be disastrous for safety-critical systems.

In *Directed Acyclic Graph* (DAG) scheduling, the deadline is shared between a set of tasks. Our method does not allocate extra time per task to handle restarts, but instead requires moving the shared deadline to facilitate tasks restarting. It is up to the online scheduler to assign this extra time dynamically based on which tasks require a restart.

UPPAAL [1] is an existing tool for modeling, validating and verifying real-time systems, which has found use in the RTS community [7]. The tool can verify properties or give counterexamples, e.g. prove that a deadline will always be met. When mitigating SEUs, the number of restarts a task can require is not bounded. Instead, each number of restarts carries a probability. Our method does not require an upper bound on the extra time needed to facilitate restarts, unlike for example the $\Delta_f$ limit used by Mosse et al. [5]. Instead, our method provides insight into how likely it is that a given deadline will be missed without such an upper bound.

To estimate probabilities, we use UPPAAL in *Stochastic Model Checking* (SMC) mode. This mode is incompatible with models not designed to be used with SMC, like the UPPAAL RTS template library by Shan et al. [7]. Probability has been used before in real-time scheduling [2], and examples of this approach include using statistical means for determining a sensible upper bound estimate for the *Worst-Case Execution Time* (WCET) [3].

**Contributions**. We present a new method for representing a DAG of tasks and the hardware as an UPPAAL SMC model including SEUs, without putting limits on the number of job restarts. Our model includes an online scheduler implemented in the UPPAAL language, allowing the scheduler to react to the effects of job restarts caused by SEUs. For any given deadline, the UPPAAL SMC model can show the probability that the deadline will be met including SEU mitigations. It is then up to the application designer to choose an acceptable risk level.

## 2 Fault and Task Model

### 2.1 Task model and communication

The DAG of tasks has a single, constrained deadline $D \leq T$, allowing us to consider each execution of the DAG in isolation with exactly one job per task (without fault detection). Furthermore, for each task $\tau_i \in \tau$, a WCET value $C_i$ must be available, which we use in our online scheduling algorithm. Our scheduler implements a global, fixed-priority, task-migrating and non-preemptive scheduling algorithm. The priority $P_i$ of task $\tau_i$ is directly derived from the WCET value $C_i$. The higher the WCET, the higher the priority of that task. Let $n$ be the number of tasks, and $i \in [0, n)$, we define the priority as $P_i = C_i \cdot n + i$, resulting in unique priorities due to the inclusion of $i$.

Communication between tasks is exclusively handled via the exchange of *tokens*, which may contain arbitrary data. Furthermore, we consider communi-

cation overhead negligible. In order to launch tasks multiple times, we assume all tasks to be stateless and free of side-effects.

### 2.2 Fault Detection and Mitigation

We assume that when a SEU occurs, it affects only one processor. While the SEU itself is transient, effects of the event may still be present and continue to affect the job. We assume that the SEU manifests itself as an incorrect result for that job (incorrect output tokens).

Let there be $m$ processors, then each processor $\pi_k$ where $k \in [0, m)$ has an associated SEU fault rate parameter $\lambda_k$. We model the inter-arrival times of faults with an exponential distribution as we are only concerned with SEUs and not with permanent degradation of the hardware. Exponential distributions are the only continuous memory-free distributions, hence knowledge about how long a processor has been fault-free does not impact the future probability of a fault in any way. Since we focus on symmetric multi-core systems, we treat the fault rate for each processor as identical ($\forall k \in [0, m) : \lambda_k = \lambda$).

We validate the output of a task using *Dual Modular Redundancy* (DMR) [6] by spawning two jobs for every task. If the content of the output tokens does not match, one or both jobs experienced a SEU while running. SEUs may also impact jobs in such a way that they never terminate. As such, jobs exceeding their WCET are also considered faulty and are terminated by a watchdog timer.

To recover from a fault, we use *Checkpoint-Restart*. The input tokens are always checkpointed and kept for the duration of the task. If a fault is detected, the produced tokens are discarded and the two jobs are started again using the original input tokens.

Our implementation of DMR runs the two jobs concurrently on separate processors, an approach also known as *Chip-level Redundant Multithreading* (CRT) [6]. As such, the use of DMR itself does not impact the *Worst-Case Response Time* (WCRT) of the task, assuming there are no SEUs requiring restarts.

## 3 Scheduling with Restarts

The number of restarts a particular job needs to endure before it is guaranteed to produce a result without a SEU is not bounded. The lack of such a bound makes the fault-free WCRT effectively infinite, and as such no longer a particular useful metric to work with. As a result, our technique only works with online scheduling.

### 3.1 Definitions

To simplify reasoning about DMR and checkpoint-restart, we introduce the fault detecting task $\overline{\tau}$, running on the fault detecting processor $\overline{\pi}$ as conceptual abstractions. The fault detecting task $\overline{\tau_i}$ is constructed from user-provided task $\tau_i$, but can detect that a SEU has occurred. The $\overline{\tau_i}$ runs the user-provided task $\tau_i$ twice and compares the output tokens of both executions. Similarly, the $\overline{\pi_k}$ is a tuple consisting of two hardware processors. With these constructions in place, a fault detecting task can be scheduled on a fault detecting processor, hence we can reuse existing scheduling algorithms.

### 3.2 Schedulability Testing using Stochastic Model Checking

Our method informs the application designer about the probability of meeting the deadline for each execution of the DAG. To obtain this number, we model the entire application (including the scheduler) as a UPPAAL SMC model [1]. Tasks are treated as black boxes and are assumed to always need their entire WCET estimate. Furthermore, we do not model the individual processors or tasks, but instead we directly model the fault-detecting processors and fault-detecting tasks. We implement our global fixed-priority scheduler in the UPPAAL language, controlling transitions in the task automaton based on its position in the scheduler queue.

### 3.3 Representation as UPPAAL Model

Our method represents the DAG of tasks, the scheduler and the hardware as a composition of four UPPAAL templates:

1. a singleton scheduler;
2. for each fault detecting task $\overline{\tau_k}$, an instance of the `task` template;
3. for each fault detecting processor $\overline{\pi_f}$, an instance of the `processor` template;
4. for each dependency between two tasks, an instance of the `edge` template.

We show the task template in Figure 1. This template is parameterized with the task id $i$ and the WCET estimate $C_i$ of the task that it represents. Tasks start out in the `UnmetDependencies` state until all of their dependencies have been met. Then, they transition to the `Ready` state where they are available to be picked up by the scheduler and assigned to a processor to start running in the `Running` state. After some time, the task either finishes or transitions to a faulty state during execution (`RunningWFault` state), moving it back to the `Unscheduled` state for another try. This procedure sheds light on how faults are modeled: a SEU is triggered by the `processor` model (omitted here for brevity) after which the task running on that processor (if any) moves to the `RunningWFault` state by synchronizing on a channel. Synchronization forces other models in the system to transition as well when in a state with an outgoing edge waiting for the same channel. In this case, the `processor` model signals the `task_transient_fault[`$i$`]` channel, prompting the task $\overline{\tau_i}$ to transition to the `RunningWFault` state.

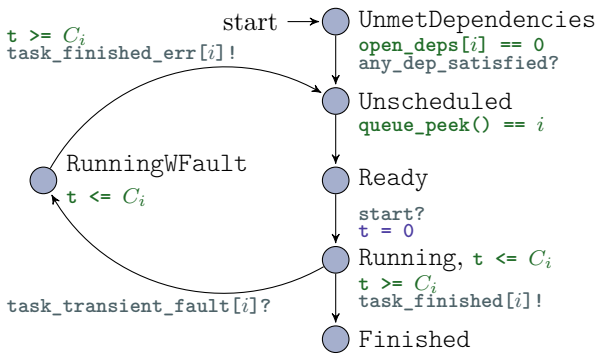



Figure 1: Summary of the UPPAAL model of a task execution. The task continues to run in an erroneous state ("RunningWFault") after experiencing a transient fault.

Running in this state does not change the task from the perspective of the scheduler. It is only when the task has completed that it either signals success ("`task_finished[`$i$`]!`") or error ("`task_finished_err[`$i$`]!`"), to which the scheduler can respond.

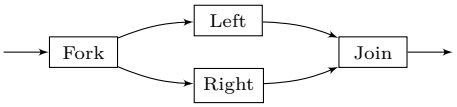
Task execution is represented as a *timed automaton*, where the state is composed of not just discrete variables but also of continuous (clock) variables. In the case of the task model this is the clock variable `t` for time. The clock `t` is set to 0 when transitioning to the `Running` state. The transitions out of the `Running` or `RunningWFault` state are guarded: $t \geq C_i$, enabling the transition only when the task has been running for its WCET estimate. A state invariant on the two running states of $t \leq C_i$ prevents lingering in the running states past the tasks WCET estimate.

Tasks may not always need their full WCET estimate to produce a result. However, modeling the early return of tasks would require knowledge about the frequency and distribution of such cases happening. Despite this, the WCET estimate provides a useful upper bound that lets us provide a lower bound probability that the deadline will be met.

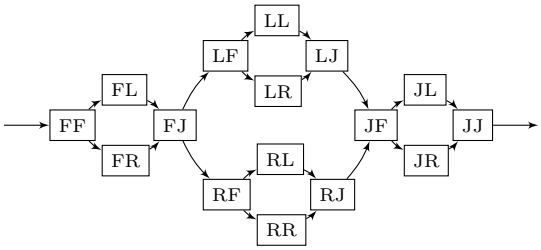
# 4 Evaluation

## 4.1 Experimental Setup

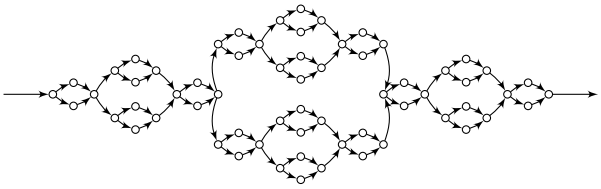
To test our technique we have created three synthetic test cases. As the potential gain by our technique scales with the number of tasks in the graph, we have tested three DAGs with varying critical path lengths. Each test case is a variation of the same fork-join graph with a different level of nesting, as shown in Figure 2. To focus just on the impact of restarting, we set the WCET estimate of each task to the same value of 1000 seconds. We do not give our DAGs a deadline or period, but are instead interested in the distribution of whole-DAG response times. Instead of obtaining a probability of meeting the deadline, we gain insight into what a sensible deadline might be together with the probability of meeting that deadline.




(a) Test case A: fork-join with four tasks




(b) Test case B: nested fork-join graph. Each of the tasks in the previous graph is replaced with the entire graph.



(c) Test case C: double nested fork-join graph. Each of the tasks in test case B is replaced with the entire graph of test case A.

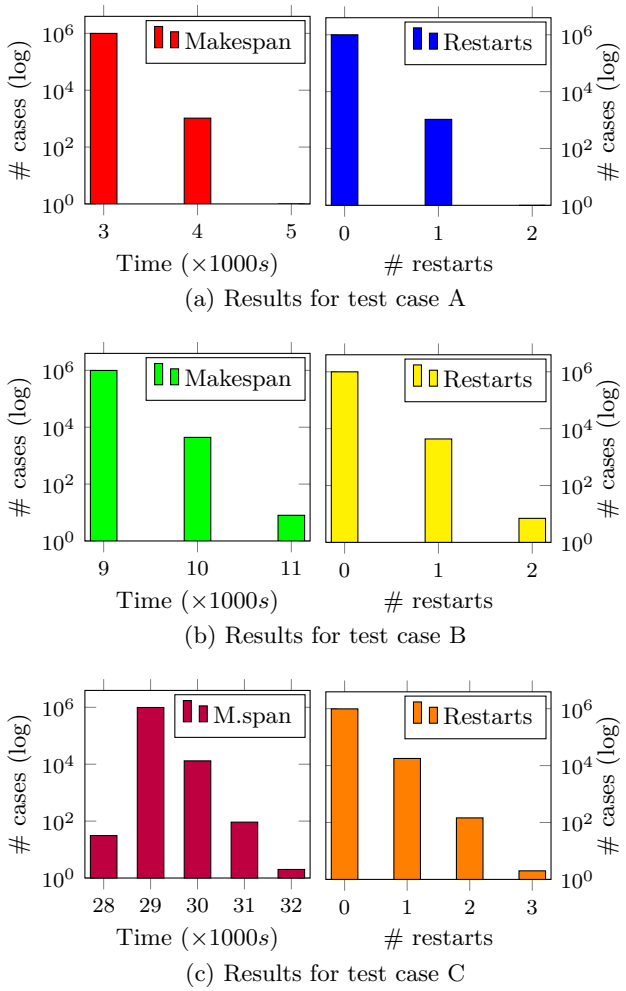
Figure 2: The various test cases, showing $4^1$, $4^2$ and $4^3$ tasks

To execute the tests, we simulate a platform with 4 fault detecting processors (constructed from 8 hardware processors). Furthermore, we set the rate of transient faults for each processor to $\lambda = 10^{-3}/hour$, which is the highest permissible fault rate per RTCA/DO-178C for faults that have only minor criticality level [4].

## 4.2 Results

Figure 3 shows the results of the UPPAAL SMC simulations. The distribution of the whole-DAG makespan (left) informs the application designer about the probability of meeting a particular deadline, or helps determining a suitable deadline. For example, let the deadline of test case C be $D_C = 3.0 \cdot 10^4 s$, the simulations can show us that the probability of missing this deadline is $1.45 \cdot 10^{-4}$. Or, if the deadline has not been determined yet, $D_B > 1.1 \cdot 10^4 s$ could be determined as a suitable deadline for case B. However, the exact bounds depend on the amount of risk acceptable in the domain of the application.

Our UPPAAL representation does not express any variance in the execution time of a task, which means that the whole-DAG execution time is always an integer multiple of a 1000 seconds. For all cases, the number of restarts has a very predictable impact on the makespan. Test case C (Figure 3c), however, also shows signs of something else: a number of runs manage to finish early. We explain this as a *timing anomaly* caused by some non-determinism in the




(a) Results for test case A

(b) Results for test case B

(c) Results for test case C

Figure 3: UPPAAL results for $10^6$ SMC simulations. For each test case, the distribution of whole-DAG makespan is shown on the left. On the right, the number of restarts is shown.

completion order of tasks. When multiple tasks complete at the same time, the scheduler randomly handles one first and reassigns the processor due to a lack of clairvoyance. Even though the scheduler assigned the highest-priority task to the idle processor, it can only consider tasks that have all of their dependencies satisfied. Furthermore, our scheduler does not preempt, as such the release of a higher-priority task may must wait for a lower priority task to finish. Our fixed-priority online scheduling algorithm is not an optimal algorithm (without preemption), which allows this source of non-deterministic behavior to impact the whole-DAG execution time.

As the number of tasks in the DAG increases, so does the number of restarts. This makes sense: the longer the DAG runs, the higher the chance of a transient fault to occur.

## 5 Conclusion and Future Work

Transient faults in processors lend themselves well to be modeled using UPPAAL in SMC mode. We have shown in this paper how an existing SMC tool can be leveraged to get timing information of an application in the presence of SEUs.

Our approach models how the tasks are scheduled, including how an online scheduler may absorb the time lost due to a restart. However, for relatively small test cases (up to 64 tasks), Figure 3 shows a strong correlation between the number of restarts and the makespan. In future work, we will look at larger DAGs of tasks and investigate varying the amount of dependencies between the tasks. We suspect that, for more loosely connected graphs, SEUs can more often be solved by making use of idle processors. As such, for a given size of the DAG, we expect the number of restarts to no longer be a good predictor of the makespan.

Finally, our results are obscured by timing anomalies. In future work, we hope to mitigate these anomalies by implementing clairvoyance in our scheduler.

## Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No. 871259 (ADMORPH project).

## References

[1] P. Bulychev et al. "UPPAAL-SMC: Statistical Model Checking for Priced Timed Automata". In: *arXiv e-prints* (2012).

[2] R.I. Davis and L. Cucu-Grosjean. "A survey of probabilistic schedulability analysis techniques for real-time systems". In: *LITES: Leibniz Transactions on Embedded Systems* (2019).

[3] S. Edgar and A. Burns. "Statistical analysis of WCET for scheduling". In: *22nd IEEE RTSS*. 2001.

[4] A. Löfwenmark and S. Nadjm-Tehrani. "Fault and timing analysis in critical multi-core systems: A survey with an avionics perspective". In: *Journal of Systems Architecture* 87 (2018).

[5] D. Mosse et al. "A nonpreemptive real-time scheduler with recovery from transient faults and its implementation". In: *IEEE Transactions on Software Engineering* (2003).

[6] Isil Oz and Sanem Arslan. "A Survey on Multithreading Alternatives for Soft Error Fault Tolerance". In: *ACM Comput. Surv.* (2019).

[7] L Shan et al. "RTLib: A Library of Timed Automata for Modeling Real-Time Systems". PhD thesis. Grenoble 1 UGA-Université Grenoble Alpe; INRIA Grenoble-Rhone-Alpes, 2016.

[8] F. Wang et al. "Single Event Upset: An Embedded Tutorial". In: *21st International Conference on VLSI Design*. 2008.