D3.3 Third report on analysis techniques for adaptive systems

Project acronym: ADMORPH
Project full title: Towards Adaptively Morphing Embedded Systems
Grant agreement no.: 871259

Due Date:	February 28th, 2023
Delivery:	Month 38
Lead Partner:	Lund University
Editor:	Martina Maggio
Dissemination Level:	Public
Status:	In progress
Approved:	
Version:	1.2

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

DOCUMENT INFO – Revision History

Date and version number	Author	Comments
16/02/2023, v1.0	Martina Maggio	First complete draft
20/02/2023, v1.1	Marcus Völp	Comments
24/02/2023, v1.2	Martina Maggio	Revision

List of Contributors

Date and version number	Author	Comments
24/02/2023, v1.2	Martina Maggio	Sections 1, 5, 7
24/02/2023, v1.2	Dolly Sapra	Section 2
24/02/2023, v1.2	Florian Haas	Section 3
24/02/2023, v1.2	Stefanos Skalistis	Section 4
24/02/2023, v1.2	Holger Blasum	Section 6

Contents

Outline

This deliverable is a report on the consortium's work in Work Package 3, discussing all the active tasks in the work package: Task 3.2 *Design-Space Exploration of Dynamically Evolving Embedded Systems*, Task 3.3 *Adaptivity-aware real-time scheduling policies*, Task 3.5 *Timing Analysis for Certification in Heterogeneous Processing Platforms*, and Task 3.6 Providing Formal Guarantees on the Behavior of the Adaptation Layer.

1 Introduction

In this report we provide a progress report and analyse the results obtained within the tasks connected to the analysis of self-adaptive system. In particular, this work package focuses on how to evaluate the behaviour provided by these systems at runtime. Specifically:

(i) Section 2 discusses design exploration. The ADMORPH project included the design and implementation of a simulation tool that allows us to compare different scenarios and their expected performance characteristics when systems include adaptation policies. The simulation is conducted at system-level, and includes the system - hardware and software - and its evolution. Genetic algorithms are used to find hardware configurations that belong to the Pareto frontier and have the characteristics that it is not possible to optimise an objective (for example: lowering power consumption, or increasing fault resilience, or minimising the execution time of a program) without worsening the performance of the system for some other objectives. Monte Carlo simulations are used to obtain mean characteristics like the mean time to failure for specific system configurations and the mean performance obtained for example with respect to power consumption.

(ii) At runtime, the morphing system can change hardware configuration depending on the current objective and desires, and especially when faults or security attacks are identified, to achieve resilience and fault tolerance. While the characteristics of the configurations are analysed with the simulator, there is a need for the runtime system to determine when to change operational mode and how to adapt. Section 3 describes adaptivity-aware real-time scheduling policies, that are meant to change the scheduling of tasks in the system, depending on the current configuration and operation mode. The ADMORPH scheduling policies guarantee the execution of an application, modelled as a directed acyclic graph, within a given deadline under the presence of faults, up to a predefined number of potentially occurring faults.

(iii) In order to obtain said adaptive scheduling policies, it is necessary to estimate the changes in the execution time of the code for a software application under different hardware configurations. The ADMORPH project enables the estimation of the worst case execution time for a system that is dynamically adapting to faults. Section 4 describes our effort in this direction, providing a timing analysis that connects the scheduling policies and our simulator results.

(iv) Section 5 describes our efforts in the analysis of faults. In particular, we consider that under faults or attacks, the worst case execution time of a software application may change, and hence tasks may experience deadline misses, and fail to provide the necessary information and computational results within the prescribed amount of time. However, to have any hope of recovering, we rely on the adaptation mechanism to provide resilience. We built a tool to compare and analyse the resilience provided according to different task failure models. The tool allows us to compare *weakly-hard constraints*, i.e., different types of constraints on the amount and distribution of said deadline misses.

(v) Finally, in Section 6 we discuss an analysis method to detect interference between applications that are configured to run in parallel on the same hardware on a separation kernel, by analyzing configuration files of the separation kernel.

2 Design-space exploration of dynamically-evolving systems

As already outlined in Deliverable D3.2, we use a Genetic Algorithm (GA) for efficient Design-Space Exploration (DSE). The GA is tasked with determining a suitable system configuration, that comprises of an hardware configuration as well as of the appropriate adaptation strategy for application running on said hardware. The final part on Task 3.3 focuses on efficient fitness evaluations for the GA, so that the algorithm can converge quickly and produce a Pareto-optimal set of design points. We explain the fitness evaluation aspect of the GA and report final results in this deliverable.

2.1 Fitness evaluation

The term *fitness evaluation* for a GA refers to the measurement of the quality of the individuals within the population of the current generation. This evaluation step allows each design point to be measured in order to determine the offspring (via the selection process) and apply other GA operators (mutation and crossover). The evaluation step uses the simulator (developed within the project, and presented in Deliverable D3.2), to get the objective values for an individual.

A run with the simulator provides a Time To Failure (TTF) and an estimate of the power consumption of a specific design configuration, which does not correctly reflect the non-deterministic nature of hardware failures. The simulator would provide more representative evaluations, that can better describe a design point, by offering the Mean Time To Failure (MTTF) and mean power usage. This requires the simulator to run multiple times and obtain a closer approximation of the actual mean values for these two characteristics.

The GA maintains a set of all Pareto-optimal points through the generations (as explained in Deliverable D3.2, Section 3.6.1). These points correspond to the configurations in which none of the objectives of a design point can be further improved without worsening the results for (at least) one other objective. In this section, we explain different methodologies we used for efficient fitness evaluation through multiple simulations for each design configuration.

2.1.1 Monte Carlo Simulation

The most common and straightforward method of evaluating the MTTF is to simply run the simulator multiple times and average the received outputs. Such an approach falls within the Monte Carlo Simulation (MCS) methods (i.e. random sampling in order to obtain numerical results). Algorithm 1 shows the Monte Carlo simulation to evaluate a batch of individuals.

Within this algorithm, each design point is evaluated the same number of times. The simulation budget is spread evenly among the design points. While this approach is useful to evaluate individual design points, it becomes compute intensive for the DSE. This is because of a large population and multiple iterations of the algorithm, leading to an extremely large number of simulations to be performed. To reduce the computation costs involved per design point, we use two techniques from Multi-Arm Bandit (MAB) algorithms, as explained in the next sub-section.

Algorithm 1 Monte Carlo simulation

```
Require: A list of $k$ design points $\mathcal{D}$
Require: The simulation budget $n$
1: function MCS($\mathcal{D}, n$)
2:     Initialize $\bar{X}$                    ▷ the objectives vector, of size $k$, for each design point
3:     for each design point $d_i \in \mathcal{D}$ do
4:         simulate $d_i$ by running the simulator for $n/k$ times
5:         update $x_i \in \bar{X}$            ▷ update MTTF and mean power for $i^{th}$ design point
6:     end for
7:     return $\bar{X}$
8: end function
```

2.1.2 Multi-Arm Bandits (MAB)

MAB algorithms are a subset of Reinforcement Learning algorithms, which is a computational approach to learn from interaction with an environment. Reinforcement learning attempts to maximise a reward signal instead of trying to optimise an objective. In this work, a MAB algorithm is repeatedly faced with k different candidates, and the evaluation (via simulation) of each candidate, that yields a numerical reward from a stationary probability distribution. It is not known beforehand how the reward probability for different candidates is distributed. The goal is to maximise the total reward over a fixed number of simulations. Since there is no prior knowledge about each of the k different options, the MAB algorithm has to find a trade-off between exploring unknown candidates and sampling the current best candidates in order to maximise this total reward.

Scalarised Successive Accepts and Rejects (sSAR)

The Scalarised Successive Accepts and Rejects (sSAR) is a MAB algorithm that works in phases where the end of each phase will rule-out a single design point. This happens when a design point is either the best or the worst from the current set of active designs. After each consecutive phase, the algorithm will focus more towards the so-called “grey area”, i.e., the design points that are not obviously performing good or bad on the design objectives. As we approach later phases, the number of active design point keeps decreasing. Since every phase has a fixed computational budget, the design points in later phases get an increasing number of simulations. The main idea is that within earlier phases it is easier to find the extreme design points (thus requiring fewer simulations) while later phases will only have active design points that are more closely positioned together, thus requiring more simulations to distinguish their objective values.

Notably, this algorithm requires a scalarisation function, which combines all objectives into one objective value (for comparing and ruling-out design points). Another important characteristic of the sSAR is that the number of simulations it performs is dynamic, meaning that running the algorithm twice with the same input will most-likely result in a different number of simulations for the whole population.

Algorithm 2 Scalarised successive accepts and rejects algorithm [6]

```
Require: A list of $k$ design points $\mathcal{D}$
Require: How many individuals to selected $p$
Require: List of scalarisation functions $S$
Require: The simulations budget $n$
 1: function sSAR($\mathcal{D}$, p, S, n)
 2:     $\mathcal{A}_j \leftarrow \mathcal{D}$, $\mathcal{P}_j \leftarrow p$
 3:     Initialize $\bar{X}$
 4:     $\overline{\log}(K) = \frac{1}{2} + \sum_{j=2}^{K} \frac{1}{j}$
 5:     $n_0 \leftarrow 0$, $n_k \leftarrow \left\lceil \frac{1}{\overline{\log}(K)} \cdot \frac{n-K}{K+1-k} \right\rceil$
 6:     for rounds $k = 1, 2, ..., K-1$ do
 7:         for all design points $i$ for which $\exists j : \mathcal{D}_i \in \mathcal{A}_j$ do
 8:             Simulate $\mathcal{D}_i$ for $n_k - n_{k-1}$ times
 9:             update $\overline{x}_i \in \overline{\mathbf{X}}$
10:         end for
11:         for $f_j \in S$ do
12:             Sort designs $\mathcal{A}_j$ according to $f_j$ scalarised emperical means
13:             $i^* \leftarrow$ arm with $\mathcal{P}_j$-th best emperical mean
14:             $i_* \leftarrow$ arm with $(\mathcal{P}_j + 1)$-th best emperical mean
15:             Gap $\Delta_i = \begin{cases} \overline{x}_i - \overline{x}_{i_*} & \text{if } i \text{ is among top } \mathcal{P}_j \text{ designs} \\ \overline{x}_{i^*} - \overline{x}_i & \text{otherwise} \end{cases}$
16:             $i \leftarrow$ the design point with the highest gap in $\mathcal{A}_j$
17:             Remove $i$ from $\mathcal{A}_j$
18:             if $i$ is the best design point in $\mathcal{A}_j$ then
19:                 $\mathcal{P}_j \leftarrow \mathcal{P}_j - 1$                ▷ Design point $i$ is accepted
20:             end if
21:         end for
22:     end for
23:     return $\overline{\mathbf{X}}$
24: end function
```

The original algorithm [6] returns the non-dominated design points of the union of all design points selected by the scalarisation functions, i.e.,

$$\bigcup_{j=1}^{|S|}\bigcup_{i=1}^{p} J_i^j,$$

where J^j is the set of p selected individuals of scalarisation function S_j. Algorithm 2, as used in this work, is slightly altered and does not return the non-dominated design points of this union set because a GA requires for exactly n individuals to be selected (for creating offspring), which is not guaranteed when taking the non-dominated individuals. In our approach, we let sSAR perform simulations in order for it to better distinguish the top n individuals. However, selecting those individuals is done via another analysis method, called NSGA-II [4].

Pareto Upper Confidence Bound

The Pareto Upper Confidence Bound (PUCB) algorithm [5] is another MAB algorithm that does not require any form of scalarisation, and works directly with multiple objectives. The scalarisation may result in a loss of information (by losing the Pareto relationships), so the PUCB algorithm is more suited to be used in multi-objective settings.

This algorithm works in the same phase based manner as sSAR algorithm, however, in each phase sub-optimal design points are removed. After each phase, the design points are ranked based on the NSGA-II algorithm and the lowest ranked individuals are removed from the active design points. In this way, the design points that are performing and potentially in the Pareto set get more simulation runs. This algorithm thus spends most of the simulation budget on the better individuals of the population.

The main disadvantage of the Algorithm 3 is that it needs to find the Pareto set twice per simulation budget spent. Finding the Pareto set is dependant on a non-dominated sorting algorithm, which is of complexity $\mathcal{O}(MN^2)$ [4], where M is the number of objectives and N is the population size. While it scales quadratically on the population size, it comes with a significant overhead when having a high simulation budget, i.e. when the number of simulations per generation is large. For example, when running a GA with a population of 100 over 50 generations with a budget of 10.000 simulations per generation, the Pareto front has to be calculated $2 \cdot 50 \cdot 10.000 = 1.000.000$ times. However, earlier research [5] has shown that PUCB is more fair and more robust when compared to scalarised algorithms.

Figure 1 visualizes the differences between three methodologies discussed in this section. We perform experiments and present our results with all of these approaches in the next section. The design points are ranked from the best (left) to the worst (right) on the horizontal scale. Higher (vertical) bar per design point refers to more evaluations on that design point.

2.2 Results

We study how the GA performs based on the presented simulator and GA operators. As a fitness evaluation method, we use the MCS algorithm for this purpose. These results are obtained by running 100 GAs with a population of 100 over 50 generations. The crossover and mutation probability are respectively 1.0 and 0.3 and selection is done using NSGA-II.

Algorithm 3 Pareto UCB

```
Require: A list of $k$ design points $\mathcal{D}$
Require: The simulation budget $n$
 1: function PARETO_UCB1($\mathcal{D}$, n)
 2:     $N \leftarrow \{1\}_k$                          ▷ $N_i$ is the number of times $d_i$ is simulated
 3:     Initialize $\bar{X}$
 4:     for $d_i \in \mathcal{D}$ do                    ▷ simulate each individual once
 5:         simulate $d_i$
 6:         update empirical reward vector $\overline{x}_i \in \overline{\mathbf{X}}$
 7:     end for
 8:     repeat
 9:         $\mathcal{A}^* \leftarrow$ Pareto set of $\overline{\mathbf{X}}$
10:         $\mathcal{A}' \leftarrow$ Pareto set when adding $\sqrt{\frac{2\ln(n\sqrt[4]{D|\mathcal{A}^*|})}{N_i}}$ to all $\overline{x} \in \overline{\mathbf{X}}$
11:         simulate random individual $i \in \mathcal{A}'$
12:         update $\overline{x}_i \in \overline{\mathbf{X}}$
13:     until $\sum_i^{|N|} N_i \geq n$                ▷ When simulation budget is reached
14:     return $\overline{\mathbf{X}}, N$
15: end function
```

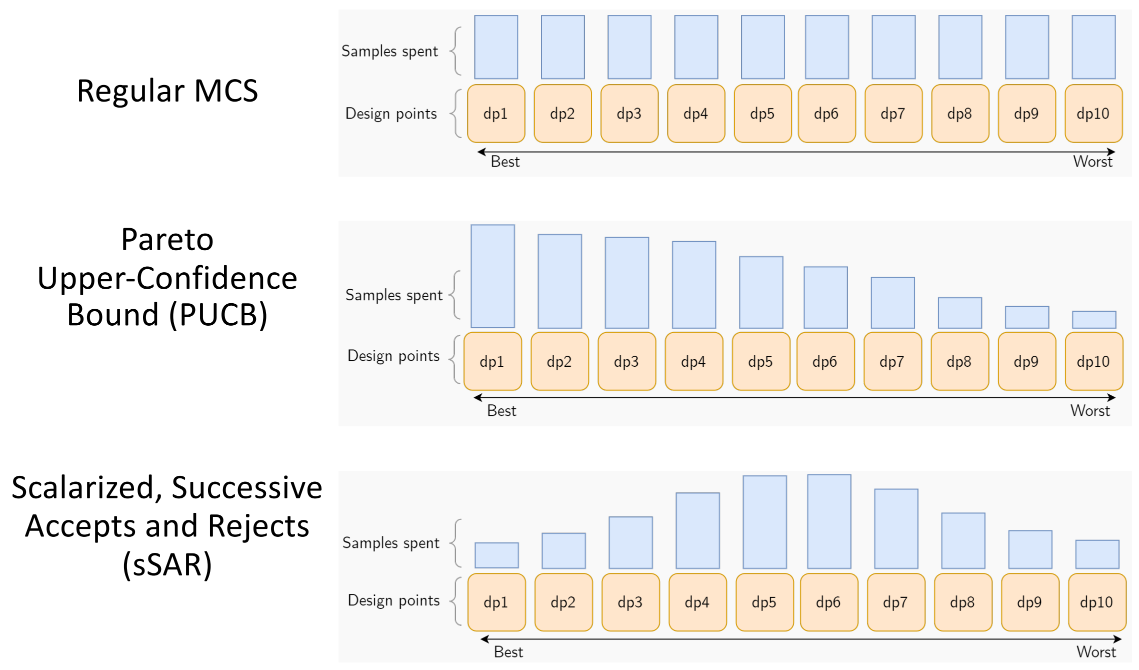
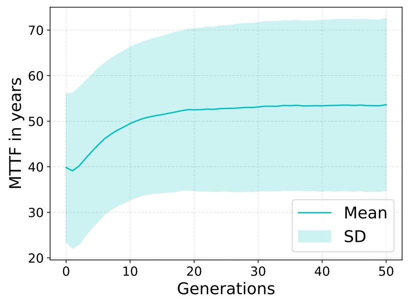
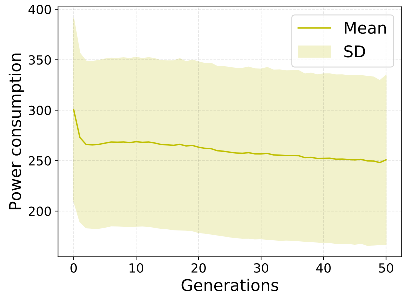

Regular MCS
Pareto Upper-Confidence Bound (PUCB)
Scalarized, Successive Accepts and Rejects (sSAR)
Samples spent
Design points
dp1
dp2
dp3
dp4
dp5
dp6
dp7
dp8
dp9
dp10
Best
Worst

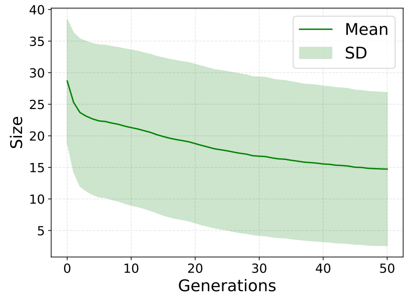
Figure 1: Number of evaluations performed per design point (by the simulator) in a population based on MCS, sSAR and PUCB algorithms.



(a) Average MTTF in years

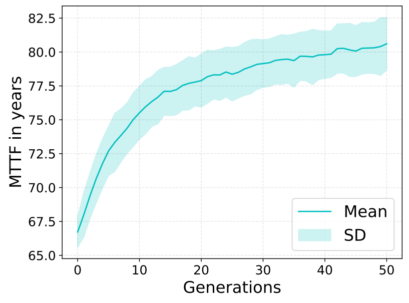


(b) Average power usage

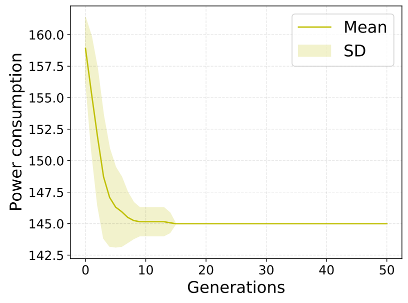


(c) Average grid size

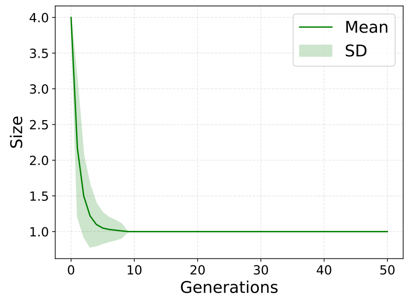
Figure 2: Average objective values over the generations of 100 GAs with a population of 100 using MCS with 100 simulations per design point.



(a) Maximum MTTF in years



(b) Minimum power usage



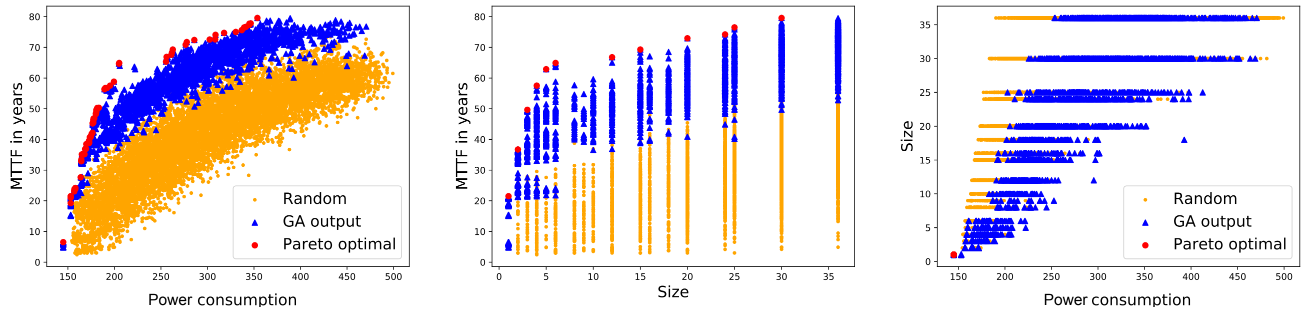
(c) Minumum grid size

Figure 3: Average of the best candidates regarding the three objective values over 50 generations of 100 GAs with a population of 100 using MCS with 100 simulations per design point.

Figure 2 illustrates the evolutions of objectives over the generations based on the average of the population. From these graphs, we can observe that the MTTF increases over the generations, while the other two objectives decrease. The standard deviation seems to be remarkably high, but this is an inherent property of the NSGA-II selection algorithm, which aims to uphold a diverse population by utilising the crowding distance.

Figure 3 shows the average of the best designs found at each generation. The best designs are those with the highest MTTF, the lowest power consumption, or the smallest size. The first graph in Figure 3 demonstrates that after 50 generations, better designs are still being detected. This is not the case for the power consumption objective, where the best design point is found after 15 generations in all GAs. The best designs in terms of the size objective is even earlier detected, at 10 generations. Since the size is a discrete objective, finding the most optimal design point in this objective (i.e., one CPU) can quickly be found. The optimal power consumption is also detectable after a certain number of generations since our design space allows for all applications to be run by a single CPU, which will result in the lowest possible power consumption.

Figure 4 shows the objective space of all the non-dominated design points at generation 50 from



(a) MTTF and power consumption objective space

(b) MTTF and hardware size objective space

(c) Size and power consumption objective space

Figure 4: Non-dominated design points of the final population obtained from running 100 GAs with a population of 100, running for 50 generations using MCS with 500 simulations per design point compared with 10,000 random design points with 1,000 simulations each. Note that the objectives of the GA are three-dimensional, but are projected towards a lower dimension.

the 100 GAs. It also plots 10,000 random design points to give a perspective on where the GAs are converging. When looking at the possible combinations of two out of the three objectives, we can see that the results of the GA are going in the correct direction. Note that there is only a single Pareto optimal point when comparing the size with power consumption (Right-most graph in Figure 4) since they both try to be minimised while being dependant on each other, resulting in one optimal point.

2.2.1 GA with MAB algorithms

In this section, we present the result of GA execution together with sSAR and PCUB algorithms. To compare the different evaluation methods, we have been running 100 GAs with a population of 100 over 50 generations for each of the evaluation methods with a set of three different sample budgets. Since the number of samples that sSAR spends is dynamic, the plots of the experiments contain the average number of samples that were spent per design point.

When spending fewer samples (see the top row in Figure 5), the MAB based algorithms seems to perform better than the MCS on the MTTF objective, but performing worse on the other two objectives. sSAR seem to perform better on 50 samples on the MTTF objective, especially around the 30th generation. At 100 samples, MCS and PUCB seems to both perform equally well, and both better than sSAR for the MTTF objective. It is noteworthy that this is for the average objective values for the whole population (for each generation). However, we are interested more in the best individuals found by the algorithm (and the Pareto set).

When looking at the best design points on the three objectives in Figure 6 and Table 1, we can observe more notable differences. For all the sample budgets, PUCB seems to be able to find a design that has a higher MTTF when compared to the other two approaches. sSAR seems to perform the worst at finding the most optimal candidates in regards to the MTTF objective. Finding the most optimal candidates in the other two objectives does not seem to be influenced

		Average		Best	
Method	simulations	MTTF	Power usage	MTTF	Power usage
MCS	10	50.147 ± 3.6807	0.0294 ± 0.0011	76.6266 ± 3.0939	0.0167 ± 0.0003
	50	50.380 ± 3.7923	0.0296 ± 0.0011	77.0716 ± 3.1870	0.0167 ± 0.0003
	100	50.988 ± 3.9172	0.0297 ± 0.0010	77.3854 ± 3.3186	0.0166 ± 0.0003
sSAR	10	50.380 ± 3.7923	0.0295 ± 0.0010	75.9460 ± 2.8626	0.0167 ± 0.0003
	50	51.131 ± 4.0339	0.0298 ± 0.0010	76.9853 ± 3.2202	0.0167 ± 0.0003
	100	50.725 ± 3.8127	0.0296 ± 0.0011	76.9718 ± 3.1952	0.0167 ± 0.0003
PUCB	10	50.453 ± 3.806	0.0296 ± 0.0011	76.6485 ± 3.0642	0.0166 ± 0.0003
	50	50.580 ± 3.8057	0.0295 ± 0.0011	77.1815 ± 3.3270	0.0167 ± 0.0003
	100	50.772 ± 3.8608	0.0297 ± 0.0010	77.5057 ± 3.4650	0.0167 ± 0.0003

Table 1: Average objectives with their standard deviation, based on the evaluation methods and the average number of simulations per design point

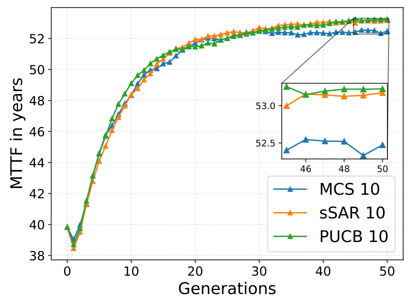
much by the choice of evaluation operator.

While PUCB does seem to perform better, its actual usage in the context of a GA is limited. When comparing the overhead of using the algorithms, PUCB adds a lot of evaluation time ($\approx 50x$) to yield a slightly better result. There might be scenarios where this overhead is not problematic and the slight benefit of better design points might be beneficial. But since DSE in itself is known to be a very time-consuming process, an extra overhead is not preferred. Nonetheless, it does illustrate that addition of MAB approaches to DSE can outperform MCS, when they are being utilised as an evaluation operator.

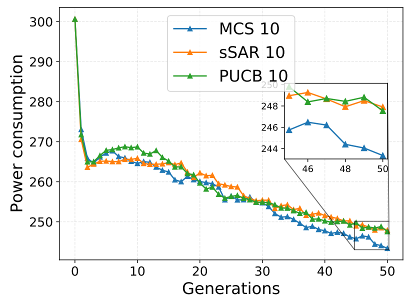
2.3 Conclusion

Modelling and exploring adaptive embedded systems is a time-consuming and a complex task. This work is continuation of the earlier presented adaptive system simulator, which creates and evaluates individual hardware design solutions based on adaptive scheduling policies. For this deliverable, we introduced a few methods to efficiently evaluating the design points through the simulator, within a GA and provided experiments to gain insights on their performance and behaviour.

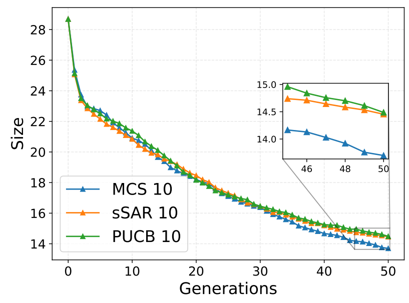
The results from our experiments also demonstrate the scope for further improvement in evaluation time. We expected PUCB and sSAR to improve the simulation time per design point, however we found the opposite to be true. Partial evaluations (to reward a design point for further simulations) caused unforeseen overheads that eventually increased the overall time DSE needs to converge. As an extension to this work, we are designing an approach where statistics or some objective values from whole simulation of one generation define the simulation budgets for next generation. It is still similar to MAB approach, but not really based on arms (partial evaluations) in the traditional sense.



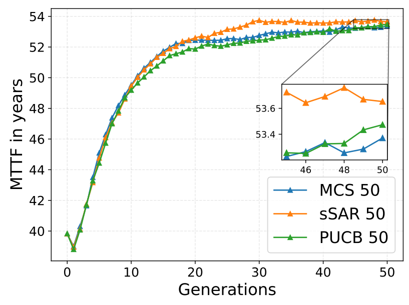
(a) Average MTTF (10 samples)



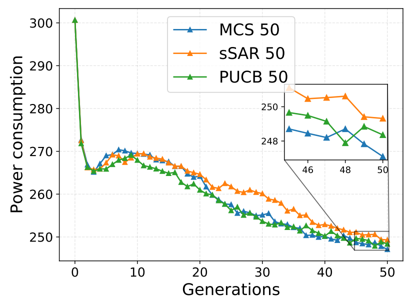
(b) Average power usage (10 samples)



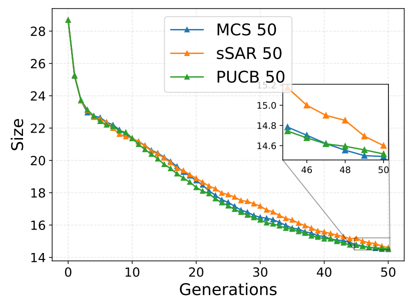
(c) Average grid size (10 samples)



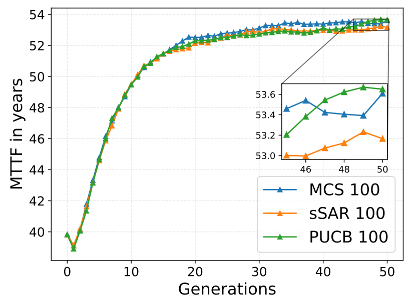
(d) Average MTTF (50 samples)



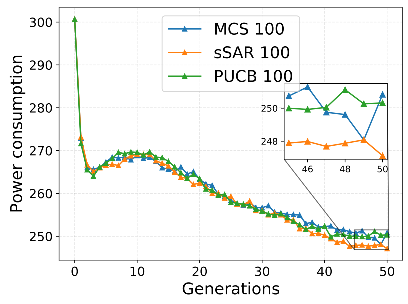
(e) Average power usage (50 samples)



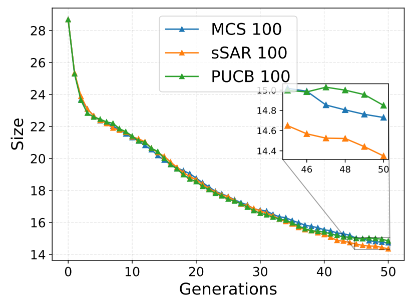
(f) Average grid size (50 samples)



(g) Average MTTF (100 samples)

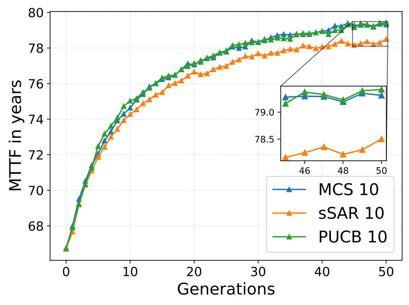


(h) Average power usage (100 samples)

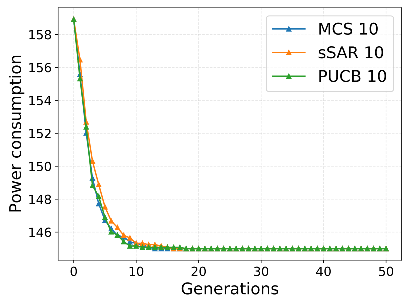


(i) Average grid size (100 samples)

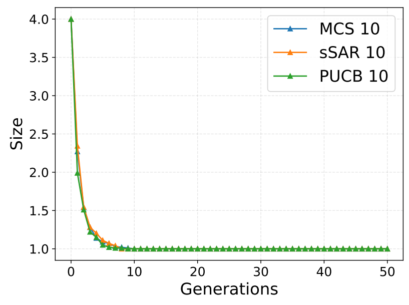
Figure 5: Comparison of the average objective values over generations between MCS, sSAR and PUCB



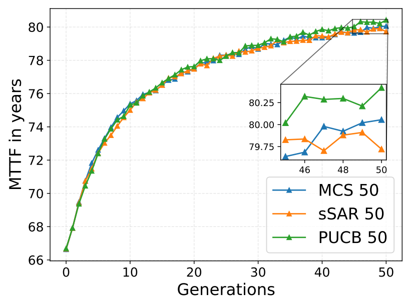
(a) Maximum MTTF (10 samples)



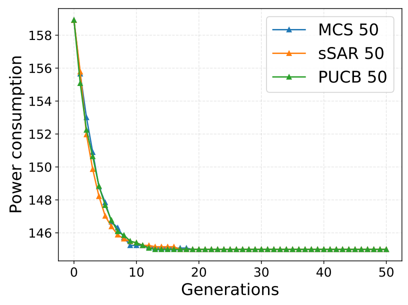
(b) Minimum power usage (10 samples)



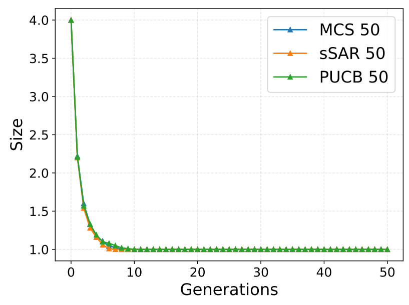
(c) Minimum grid size (10 samples)



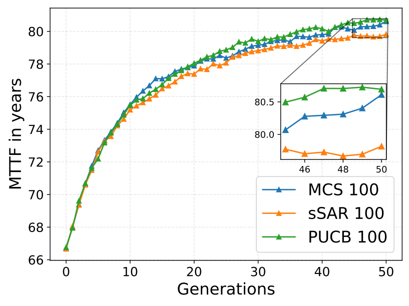
(d) Maximum MTTF (50 samples)



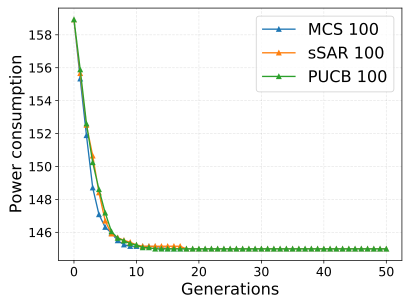
(e) Minimum power usage (50 samples)



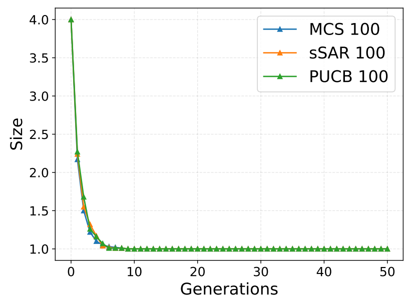
(f) Minimum grid size (50 samples)



(g) Maximum MTTF (100 samples)



(h) Minimum power usage (100 samples)



(i) Minimum grid size (100 samples)

Figure 6: Comparison on the average best objective values over the GA generations between MCS, SAAR and PUBC.

In the ADMORPH design methodology, the identified best configuration will be subjected to an in-depth simulation to find possible failures and obtain the best possible system reaction to each possible fault.

3 Adaptivity-aware real-time scheduling policies

Real-time applications are described by the domain-specific language (DSL), provided by Task 1.1, and represent a directed acyclic graph (DAG). In Task 3.3, scheduling policies are developed to guarantee the execution of the DAG within the specified deadline. To connect the tools of the different work packages, the *ADMORPH Exchange Format* (AXF) was proposed to standardise the representation of DAGs and their additional non-functional properties like deadlines. This extensible text-based format follows the *DOT* format specification to describe graphs consisting of nodes and edges in-between.

The main objective of this task is to enable a WCET estimation of a system that is dynamically adapting to faults. Our scheduling policies will guarantee the execution of the DAG within a given deadline under the presence of faults, up to a predefined number of potentially occurring faults.

Different countermeasures are required to keep the system in an operable state, depending on the type of the fault. If a permanent fault occurs, a *processing element* (PE) fails ultimately and has to be removed from the set of active PEs. As a consequence, an alternative task schedule has to be selected, which is feasible with a reduced number of PEs, while still guaranteeing the complete graph execution within its deadline. Alternatively, transient faults demand only the re-execution of the affected task. Depending on the tightness of the schedule, the overall execution time, even with re-execution of a single task, can be still below the WCET.

Based on the fault model with the expected rates of transient and permanent faults, the DAG deadline, and the individual task execution times, different schedules for the DAG execution can be calculated. By specifying the number and type of faults that must be tolerable during the execution, the required number of PEs can be determined. This allows for a trade-off between error coverage and cost of additional (spare) hardware.

The ADMORPH scheduling run-time environment (RTE) reads the DAG from the AXF file to calculate possible schedules, and executes the graph. Different hardware architectures are supported, since the tasks are provided as C source code, and are compiled for the target architecture. The RTE allows to measure the task execution of a DAG to record the observed execution times, and stores these execution times in the AXF. With this, the scheduling computations can be performed on a server system, independent of the target architecture. This is especially beneficial in combination with the design space exploration (DSE) of Task 3.2, where task schedules for different system configurations have to be calculated repeatedly.

Building on the work described in Deliverable 3.2, the RTE was enhanced to support spare PEs and a remapping of tasks to PEs in case of a permanent error. Further, a fault-injection mechanism was developed to trigger the error detection mechanism of the RTE, which allows the simulation of both permanent and transient faults.

Next, scheduling policies were developed that consider a given fault model to calculate possible schedules for a DAG with a specified minimum number of tolerable faults. This includes the analysis of the reconfiguration phase and the consideration of the impact of the reconfiguration on the overall graph execution.

However, remapping PEs on a different core in case of permanent failure, as well as switching between schedules with different task mappings, leads to additional interferences. Given the situation that a task was already executed once on a processing element, the local caches already hold the instructions for subsequent executions, as well as constant data input that does not result from another actor execution. If this core fails and the task requires to be remapped, additional traffic hits the bus or interconnect network, since the instructions and data have to be fetched again. This interferes with the concurrent execution of other tasks in the system, and potentially delays the execution of other actors, thus impacting the worst case behaviour. As a consequence, these interferences must be considered in the schedulability analysis.

4 Timing analysis for certification in heterogeneous processing platforms

Safety-critical applications typically have tasks with real-time requirements that must be met in order for the system to be considered safe. Adapting the execution of such tasks, if not done properly, poses a threat to safety as timeliness and predictability become hard to verify. In addition, contemporary heterogeneous processing platforms, in particular COTS platforms, have complex processing architectures, an increased level of shared resources, and complex interconnect infrastructures. This structure makes it very difficult to calculate accurate bounds for the Worst-Case Execution Time due to shared-resource contention that causes timing interference. On top of timing interference, adapting the execution order, i.e. the task schedule poses an additional challenge as the amount of timing interference changes.

In Task 3.5, we consider the certification challenges that stem from adapting a system and its task schedule, which is in line with ADMORPH, and not how to generate a single schedule for heterogenous platforms. In fact, this is an on-going research topic with several EU projects dedicated to that, e.g. ARGO, MASTECS, etc. In order to provide a system that can be certified the standards of the domain (e.g. in avionics DO-178C, ARINC 653, etc.) define, either explicitly or implicitly, several artifacts that should be presented to the certification authority. When it comes to hardware configuration and scheduling, the certification authorities (CAs), at their highest certification level, require that H/W configuration and schedules are determined at design-time and are presented as artifacts to the CA. In lower certification levels, such requirements are gradually relaxed and approved scheduling algorithms can be used at runtime, alleviating the need for the a-priori generation of schedules.

In order to cover such a varying spectrum of timing analysis requirements for adaptive systems, we employ the standardized Architecture Analysis and Design Language (AADL) and the CHEDDAR timing analysis tools. AADL is an architecture description language, extensible by annexes, and is used to model the software and hardware architecture of an embedded, real-time system. Having an AADL model of the system allows to perform several types of analysis, depending on the annexes uses, including Fault-Tree Analysis, Failure mode and effects analysis as well as scedulability analysis. CHEDDAR is a schedulability analysis tool for AADL that includes known scheduling algorithms, e.g, EDF, RM, Hierarchical Scheduling policies (ARINC 653), etc. Having a common model and industry-used tools that enable schedulability analysis along with fault-analysis is well aligned with the goals of ADMORPH.

Nevertheless, these models and tools have not been designed for adaptive systems and it is generally considered quite tedious to write such models. To obtain AADL models that encompass adaptivitly and to enable the aforementioned analysis, we utilize the Multi-model Model of Computation (MoC) defined in Task 3.4, "Models of computation and derived architectures to allow seamless reconfiguration".

4.1 Models of computation and derived architectures to allow seamless reconfiguration

4.1.1 Running Example

Consider an embedded computer of a search-and-rescue civil aircraft that contains the functionality necessary for radar and radio communication and that runs two other tasks: one high-criticality task and another low-criticality one. An example schedule for when the aircraft is cruising is illustrated in Figure 7a. It meets the timing requirements (deadlines) of all tasks on a dual-core architecture. While cruising, the radar is looking for large objects in its immediate flight path, whereas while searching for survivors the radar is looking for smaller objects in all directions. As a result radar changes its worst case execution time requirement for the two different phases of the mission and possibly also its invocation frequency (or period). In our working example, this would correspond to a deadline change for the radar from time instance $t = 7$ to time instance $t = 4.5$, as illustrated in Figure 7b. Again, a valid schedule can be found. In addition, when a core overheats, the operating frequency of that core needs to be dropped to avoid permanent damage. This results in potentially longer executions. As a result, radar and radio can no longer be co-located on the same core and the low-criticality task must be dropped, both in cruise mode (Figure 7c) and in search mode (Figure-7d), since no feasible schedule can be found with all tasks executing.

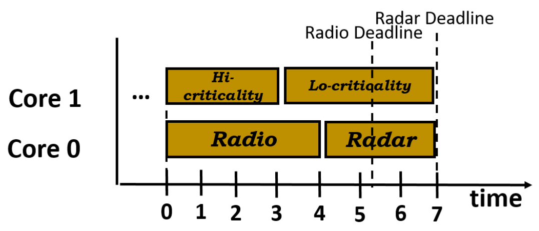
4.1.2 Task model

In task-models, a system is modeled by a set of tasks T that are executed periodically. Each task τ releases a *job* j_τ every p_τ time units, called the period. That job j must complete within $d_\tau \leq p_\tau$ times units, called *relative deadline*. The Worst-case execution time (WCET) for any job of task τ is denoted as C_τ.

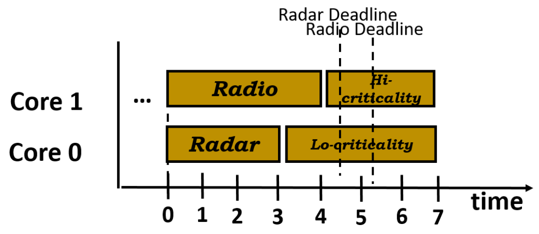
Definition 1 (Task-model)**.** *A task-model is the tuple* $TM = (T, C, P, D)$*:*

(i) T *is the set of tasks.*

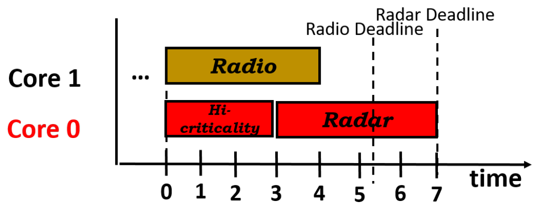
(ii) C *is the set of WCET of each task*



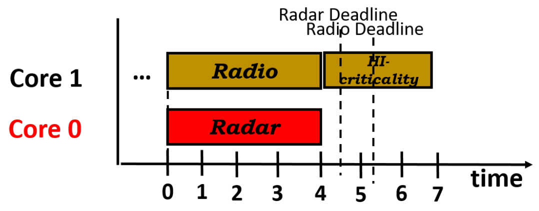
(a) Normal execution while cruising



(b) Normal execution while searching



(c) Abnormal execution while cruising



(d) Abnormal execution while searching

Figure 7: (Motivational example) Execution of the cruise/search modes in normal/abnormal timing conditions

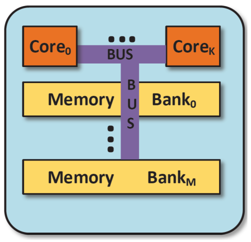
(iii) P is the set of periods of each task

(iv) D the set of deadlines of each task

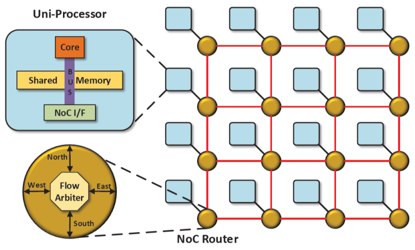
4.1.3 Computing architecture model

At the time this report was written, several research and commercial computing platforms with multiple Processing Elements (PE) were available. These broadly can be categorised according to i) the type of PEs, ii) the on-chip memory organisation and iii) the network, in case of multiple embedded devices as illustrated in Figure 8a.

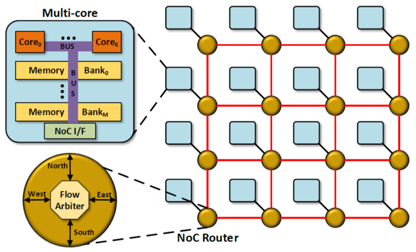
Homogeneous (as opposed to heterogeneous) are platforms where all PEs are of the same type, e.g. CPU, GPU, DSP, Routers, etc., and have the same overall speed when executing a task (i.e. all PEs have the same clock speed, cache size, I/O interfaces and any other mechanism that can affect the PEs' timing behavior). From the perspective of how shared memory is organised, there are three main categories: centralised, distributed and mixed. In a centralised memory organisation, the time for any PE to access any memory location (typically via a bus) is uniform, when there is no interference, that is, access times do not depend on the targeted memory location (address). On the other hand, in the distributed memory organisation, PEs use a different mechanism, such as Direct Memory Access (DMA) engines or on-chip-networks (NoCs), to access remote memory locations, thus having a Non-Uniform Memory Access (NUMA) in terms of timing. The mixed memory organisation is a combination of the centralised and distributed memory organisations, where a subset of PEs access their shared memory uniformly, but accessing the memory shared by another set of PEs is non-uniform.



(a) Centralised architecture



(b) Distributed architecture



(c) Mixed architecture

Figure 8: Possible architectures

In order to capture the different architectures with a single model that will be used to model adaptive systems, the generic architecture model is introduced. A single model is important as it enhances applicability of the proposed methods, which in the context of hard real-time systems is desirable, as safety properties have to be proven only once.

Definition 2 (Generic architecture model)**.** *A generic architecture model is a tuple $\mathcal{GA} = (\mathcal{C}, \mathcal{L}, \mathcal{M}, \mathcal{N})$ where:*

(i) *$\mathcal{C}$ is a set of sets of PEs; each set $c \in \mathcal{C}$ is called a computing cluster, with each computing cluster c containing one or more cores k, i.e. $k \in c$*

(ii) *$\mathcal{L} \subseteq \mathcal{C} \times \mathcal{C}$ is a set of links among computing clusters that form the network*

(iii) *$\mathcal{M}$ is a set of memory banks/locations*

(iv) *$\mathcal{N}$ is the set of network channels of a network interface*

When the generic architecture model is instantiated to a concrete architecture model matching with one of the architecture types described earlier, parameters are chosen accordingly:

(i) the centralised architectures $MA = (\{\{k_1, \ldots, k_N\}\}, \emptyset, \mathcal{M}, \emptyset)$ are instantiated without any network (Figure 8a),

(ii) the distributed architectures $DA = (\{\{k_1\}, \ldots, \{k_N\}\}, \mathcal{L}, \mathcal{M}, \mathcal{N})$ are instantiated with one core k per cluster (Figure 8b), and

(iii) mixed architectures are instantiated as $MX = (\mathcal{C}, \mathcal{L}, \mathcal{M}, \mathcal{N})$ (Figure 8c).

For the architectures that have a network, it is assumed that each cluster has one network interface, with multiple channels.

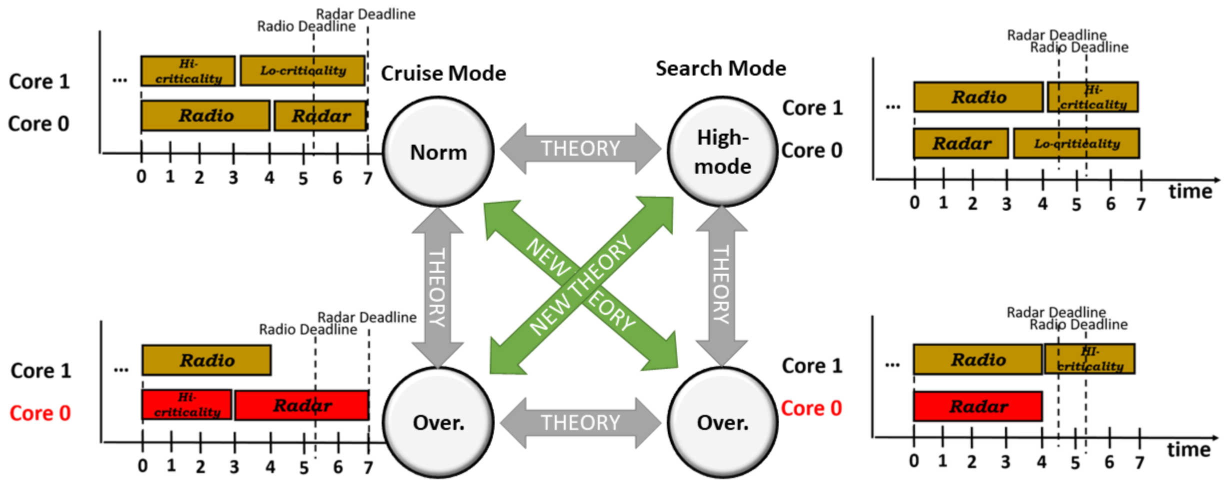


Figure 9: State transition of the running example

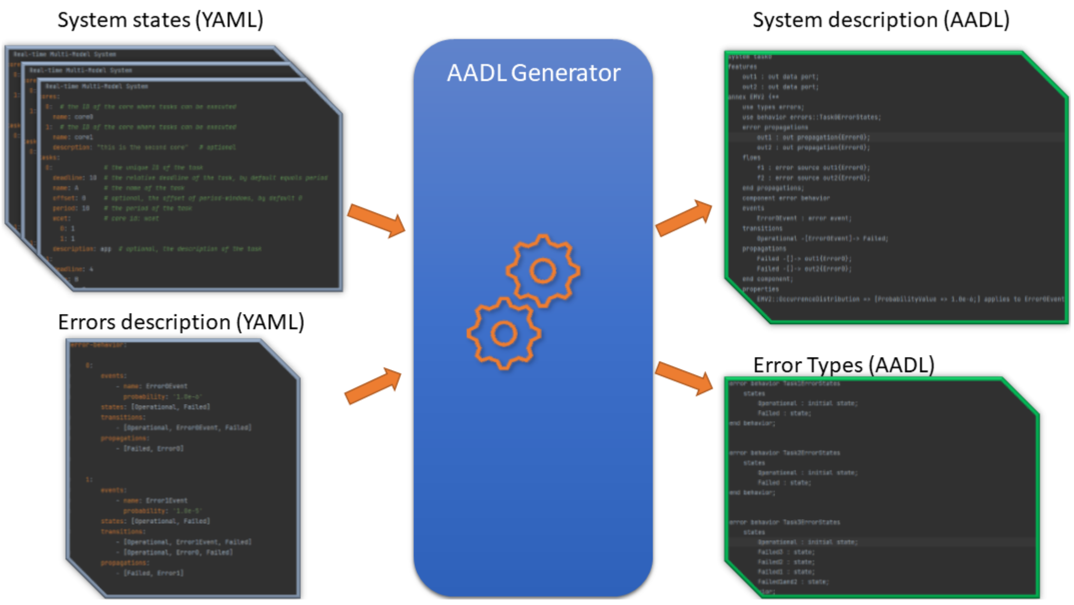
4.1.4 Model Reconfiguration

A task model effectively describes the system requirements from a functional and timing perspective, whereas the architecture model describes the physical capabilities of the underlying hardware. Any change of any of the parameters of the task or architecture mode constitutes an adaptation; revisiting our previous example of the search-and-rescue aircraft, the change in timing requirements in the various phases of the missions, is reflected by a change in the corresponding task model, that is transitioning from a task model TM, for the cruise mission, to a task model TM', for the search mission. Such a transition can be performed safely, if it has been either pre-computed at design time or if a schedulability test is passed at runtime. In a similar manner, a change in hardware capabilities (e.g. the failure of a core) can be reflected as a transition from an architecture model $\mathcal{GA}$ to a new architecture model $\mathcal{GA}$' without this core, as illustrated in Figure 9.

Utilizing task and architecture model pairs to describe the state of a system, including the desired state into which a system should evolve, we can describe all required adaptations effectively, including updates of task graphs with reconfiguration tasks, on-demand redundancies, communication re-mapping, failing cores, etc. As such, these models can be used to build algorithms that solve the associated optimisation problems, either at design-time or at runtime.

4.2 Multi-model to AADL generation for timing analysis

In order to be able to perform timing analyses and fault analyses, we developed an automatic generator of AADL models from a Multi-Model MoC. The structure of the workflow is illustrated in Figure 10. As shown in the Figure, the Multi-Model is a collection of textual files (YAML) where each file describes the state of software and hardware in one configuration. In addition, for the fault-analysis, several other files are created, which describe the types of errors, replication, etc. With those inputs, the generator creates a fundamental AADL model that can be used for schedulability and fault-analysis.



```
Real-time Multi-Model System
cores:
  0:  # the ID of the core where tasks can be executed
    name: core0
  1:  # the ID of the core where tasks can be executed
    name: core1
    description: "this is the second core"   # optional
tasks:
  0:              # the unique ID of the task
    deadline: 10  # the relative deadline of the task, by default equals period
    name: A       # the name of the task
    offset: 0     # optional, the offset of period-windows, by default 0
    period: 10    # the period of the task
    wcet:         # core id: wcet
      0: 1
      1: 1
    description: app  # optional, the description of the task
  1:
    deadline: 4
```

```
error-behavior:

    0:
        events:
            - name: Error0Event
              probability: '1.0e-6'
        states: [Operational, Failed]
        transitions:
            - [Operational, Error0Event, Failed]
        propagations:
            - [Failed, Error0]


    1:
        events:
            - name: Error1Event
              probability: '1.0e-5'
        states: [Operational, Failed]
        transitions:
            - [Operational, Error1Event, Failed]
            - [Operational, Error0, Failed]
        propagations:
            - [Failed, Error1]
```

```
system task0
features
    out1 : out data port;
    out2 : out data port;
annex EMV2 {**
    use types errors;
    use behavior errors::Task0ErrorStates;
    error propagations
        out1 : out propagation(Error0);
        out2 : out propagation(Error0);
    flows
        f1 : error source out1(Error0);
        f2 : error source out2(Error0);
    end propagations;
    component error behavior
    events
        Error0Event : error event;
    transitions
        Operational -[Error0Event]-> Failed;
    propagations
        Failed -[]-> out1(Error0);
        Failed -[]-> out2(Error0);
    end component;
    properties
        EMV2::OccurrenceDistribution => [ProbabilityValue => 1.0e-6;] applies to Error0Event
```

```
error behavior Task1ErrorStates
    states
        Operational : initial state;
        Failed : state;
end behavior;


error behavior Task2ErrorStates
    states
        Operational : initial state;
        Failed : state;
end behavior;


error behavior Task3ErrorStates
    states
        Operational : initial state;
        Failed3 : state;
        Failed2 : state;
        Failed1 : state;
        Failed1and2 : state;
```

Figure 10: AADL generation tool

Despite the fact that the generated AADL model is able to perform the aforementioned analyses, certain elements may not be incuded as the type of certificatition heaviliy guides the extend of the AADL model and this is a proof-of concept approach. Hence, in order to reach certification standards the engineer may have to augment the AADL model with the required attributes, but the skeleton and the majority of the model is generated automatically. For the sake of coherence, we shall report more detailed examples of the used workflow and tools in Deliverable D5.3, alongside the use-cases, where it is more relevant.

5 Providing formal guarantees on the adaptation layer

In Deliverable D3.2, we introduced the theoretical underpinning of the study of weakly-hard systems, in the most possible general way, and the importance of the weakly-hard model for adaptive and morphing embedded systems. Using weakly-hard systems, we can model problems and faults and properly determine when the embedded system needs to react to external event – being these events cyber-attacks, faults, or due to interference of any sort.

Weakly-hard tasks behave according to patterns of hit and missed deadlines that are (mainly) window-based. The model constrains one or more of the following items: (i) the minimum number of deadlines that are hit, (ii) the minimum number of consecutive deadlines that are hit, (iii) the maximum number of deadlines that may be missed, or (iv) the maximum number of consecutive deadlines that may be missed. The aim of our research here is to provide a formal analysis tool for systems that satisfy one or more of these weakly-hard constraints. The satisfied constraints form the set Λ, properly defined in Deliverable 3.2, where we discussed our theoretical investigation.

The last part of our work on Task 3.6 was devoted to a more practical endeavour: the creation of a software library[1] for the analysis of weakly-hard tasks to:

(i) compare two arbitrary weakly-hard constraints or two sets of weakly-hard constraints, obtaining answers about their dominance (are the two constraints equivalent, does one dominate the other, or is there no dominance relation between the two),

(ii) translate a weakly-hard constraint or a set of weakly-hard constraints into a corresponding directed labeled graph, that represents (and is able to generate) all the sequences that belong to the satisfaction set of the set of constraints,

(iii) produce all the sequences of arbitrary length that satisfy a set of weakly-hard constraints.

Our software library, `WeaklyHard.jl` is built having scalability as a first-class citizen, to allow the analysis of relevant large case studies like the ones defined in Work Package 5.

5.1 `WeaklyHard.jl`

We distribute `WeaklyHard.jl` as an open-source package, written in the Julia [2] programming language. Julia is a scripting language with Just-In-Time compilation. The language design is centered upon two concepts: type-stability and function specialisation through multiple-dispatch. The type-stable compilation provides an implementation that is close to the hardware, resulting in an efficient code execution. Multiple-dispatching allows us to write a user-friendly code library (or package). Additionally, Julia's builtin package manager simplifies the distribution of non-proprietary packages.

Any weakly-hard constraint can be represented using an automaton. Automata have been used in the analysis of networked systems [10, 19], schedulability [20, 7, 8], and control systems [12, 13, 14, 18]. Generally, for weakly-hard constraints, each automaton vertex represents the task's state, i.e., the relevant suffix of the sequence of job outcomes. Edges, on the contrary, represent an outcome, that force a transition from a current state to a different one, based on missing or hitting the current deadline. Weakly-hard systems are however inherently complicated to analyse, due to their combinatorial nature. While the representation is simple enough, the complexity is translated to the state space and comes into play when the window length increases and the number of vertices rapidly grows.

In the following, we present a novel and scalable approach to the generation of automata that represent sets of constraints.

5.1.1 Weakly-hard constraints as directed labeled graphs

Suppose that $\tau \vdash \Lambda$. We use $\mathcal{G}_\Lambda = (V_\Lambda, E_\Lambda)$ to indicate the directed labeled graph $\mathcal{G}_\Lambda$ corresponding to the automaton representation of τ. Here, V_Λ represents the set of *vertices* in the graph and E_Λ represents the directed *edges* between vertices (also denoted *transitions*). Each vertex $v_i \in V_\Lambda$

[1]The library is distributed as open source software at `https://github.com/NilsVreman/WeaklyHard.jl`

represents a word $w_i \in \mathcal{S}(\Lambda)$. With a slight notational abuse, vertices v_i will occasionally (when it is evident from context) be treated as their word representations w_i. The transition $e_{i,j} \in E_\Lambda$ corresponds to a tuple $e_{i,j} = (v_i, v_j, c_{i,j})$ where the vertex pair $v_i, v_j \in V_\Lambda$ denotes respectively the tail and head of the transition, and the character $c_{i,j} \in \Sigma$ corresponds to the transition's label. A transition $e_{i,j}$ is feasible if and only if the concatenation of the character $c_{i,j}$ to the word w_i satisfies all the constraints in Λ. Formally:

$$e_{i,j} \in E_\Lambda \Leftrightarrow \{w_i\,(2, |w_i|)\,,\, c_{i,j}\} = w_j \vdash \Lambda.$$

Finally, for two vertices $v_i, v_j \in V_\Lambda$ we say that v_j is a direct successor of v_i if there exists a transition $e_{i,j} \in E_\Lambda$. Without loss of generality, we will assume that each vertex $v_i \in V_\Lambda$ can have at most two direct successors with distinct transition outcomes, i.e., one successor v_{j_1} through e_{i,j_1} with label $c_{i,j_1} = 1$ and (if permissible) one successor v_{j_2} through e_{i,j_2} with label $c_{i,j_2} = 0$.

5.1.2 Addressing scalability

It is always possible to construct the automaton $\mathcal{G}_\Lambda$ in a naïve way, including $|\mathcal{S}_N(\Lambda)|$ vertices, where N is the maximum window length of the constraints in Λ. In order to improve performance and scalability, we include the following optimisations:

(i) representing words as bit strings,

(ii) minimising the automata size by combining equivalent vertices during the automata generation, and

(iii) representing large sets of constraints with their dominant subset.

Support for bit string operations (like shifting) is essential for efficient sequence management. Logical and bitwise operations are directly supported by all processors, thus they are highly optimised and require a minimal amount of instruction cycles. We use the following notation: & is the *bitwise and*, | is the *bitwise or*, and $\ll$ is the *logical left-shift*.

Each word $w \in \mathcal{S}(\Lambda)$ is a sequence of outcomes and can therefore be interpreted as a string of bits. This follows from a miss or hit outcome being represented by respectively a 0 or 1 in the alphabet Σ. The rightmost character in w is the outcome of the last job, hence $w = 001$ implies that the last deadline has been a hit, but the two previous ones were missed. Assuming that the task τ experienced the outcomes w and the next outcome is $c \in \Sigma$, then the new sequence of outcomes becomes $w' = (w \ll 1) \mid c$.

The size of the naïve automaton can be reduced substantially by combining vertices that would otherwise result in language-equivalent states [9]. Two vertices $v_{i_1}, v_{i_2} \in V_\Lambda$ are considered equivalent if they share the same direct successors with the same transition outcomes, i.e., $e_{i_1,j} = e_{i_2,j}$, $\forall v_j \in V_\Lambda$.

As an example, consider the `AnyHit` constraint $\lambda = \binom{x}{k} = \binom{1}{2}$. Trivially there are only three feasible vertices in the naïve automaton, since there are $2^k = 4$ words in Σ^k and $w = 00$ is infeasible. The words $w_1 = 11$ and $w_2 = 01$ share the same direct successors with the same transition outcomes, i.e., they are equivalent since $(w_1 \ll 1) = (w_2 \ll 1)$ and $(w_1 \ll 1) \mid 1 = (w_2 \ll 1) \mid 1$, considering

Algorithm 4 Generation of the minimal automaton representation $\mathcal{G}_\Lambda$, of a set of weakly-hard constraints Λ.

```
1:  procedure BuildAutomaton($\Lambda$)
2:      $V_\Lambda \leftarrow \{v_1 = (1 \ll n) - 1\}$
3:      $E_\Lambda \leftarrow \emptyset$, $finished \leftarrow false$
4:      while not $finished$ do
5:          $finished \leftarrow true$
6:          for $v_i \in V_\Lambda$ do
7:              if $\nexists e_{i,j} \in E_\Lambda$ then
8:                  $finished \leftarrow false$
9:                  $v_{j_0} \leftarrow compact(\Lambda, (v_i \ll 1))$
10:                 $v_{j_1} \leftarrow compact(\Lambda, (v_i \ll 1) \mid 1)$
11:                 if $v_{j_0} \vdash \Lambda$ then
12:                     $V_\Lambda \leftarrow V_\Lambda \cup \{v_{j_0}\}$
13:                     $E_\Lambda \leftarrow E_\Lambda \cup \{e_{i,j_0} = (v_i, v_{j_0}, 0)\}$
14:                 end if
15:                 $V_\Lambda \leftarrow V_\Lambda \cup \{v_{j_1}\}$
16:                 $E_\Lambda \leftarrow E_\Lambda \cup \{e_{i,j_1} = (v_i, v_{j_1}, 1)\}$
17:             end if
18:         end for
19:     end while
          return $\mathcal{G}_\Lambda = (V_\Lambda, E_\Lambda)$
20: end procedure
```

the window length $k = 2$. Intuitively, the fact that it is possible to combine vertices comes from the realisation that the past becomes irrelevant. Once a new outcome is obtained, only the latest outcomes are relevant. Combining the equivalent vertices results in a new vertex representing the word $w = w_1 \,\&\, w_2$. When a vertex v_i is added to the graph, its direct successors v_{j_0} and v_{j_1} are generated and added to the set of vertices V_Λ if and only if $w_{j_0}, w_{j_1} \vdash \Lambda$ respectively. Thus, all infeasible and equivalent vertices (and corresponding paths in the naïve graph) are removed at runtime.

Finally, we construct the graph $\mathcal{G}_{\Lambda^*}$ for the smallest possible set of constraints $\Lambda^* \subseteq \Lambda$ that is equivalent to Λ, i.e., such that $\mathcal{S}(\Lambda^*) = \mathcal{S}(\Lambda)$. This equivalence also translates to the automata $\mathcal{G}_{\Lambda^*} \equiv \mathcal{G}_\Lambda$ since they represent all the feasible sequences $w \in \mathcal{S}(\Lambda) = \mathcal{S}(\Lambda^*)$.

Algorithm 4 explains how we generate the minimal automaton $\mathcal{G}_\Lambda$. The automaton is initialised with a single vertex corresponding to the word $w_1 = 1^n$, $v_1 = (1 \ll n) - 1$. Here, n is the largest number of hits required in a window from any of the constraints $\lambda_i \in \Lambda$, e.g., $n = 3$ for the set $\Lambda = \left\{\binom{3}{5}, \left\langle{2 \atop 5}\right\rangle\right\}$. As long as there exists vertices $v_i \in V_\Lambda$ that have no direct successors, the successors v_{j_0}, v_{j_1} are created and added to the vertex set if the resulting words satisfy all the constraints in Λ. Since the weakly-hard constraints are vulnerable to additional deadline misses (but not hits), the constraints need only to be verified in the case when a deadline miss is added to

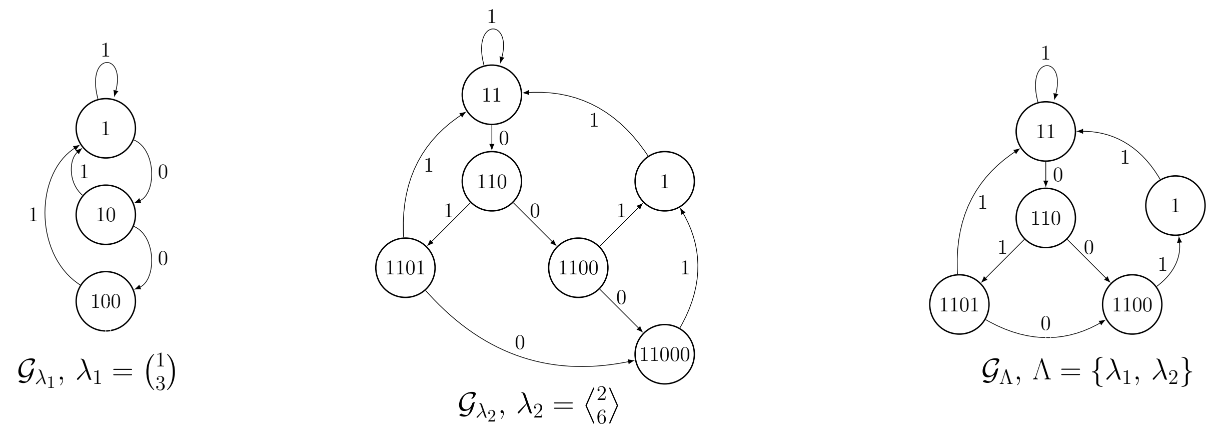

1
10
100
0
1
$\mathcal{G}_{\lambda_1}$, $\lambda_1 = \binom{1}{3}$
11
110
1101
1100
11000
1
0
$\mathcal{G}_{\lambda_2}$, $\lambda_2 = \left\langle {2 \atop 6} \right\rangle$
11
110
1101
1100
1
0
$\mathcal{G}_{\Lambda}$, $\Lambda = \{\lambda_1, \lambda_2\}$

Figure 11: Automata $\mathcal{G}_{\lambda_1}$, $\mathcal{G}_{\lambda_2}$, and $\mathcal{G}_{\Lambda}$ representing respectively λ_1, λ_2, and $\Lambda = \{\lambda_1, \lambda_2\}$ from Example 1.

the sequence, i.e., for v_{j_0}.

The new words are passed through a function in order to *compact* them. This function reduces the new word to the minimal, equivalent word that would still be able to represent the constraints in Λ. In particular, if either $v_{l_0} = (v_i \ll 1)$ or $v_{l_1} = (v_i \ll 1) \mid 1$ would result in an existing, equivalent vertex v_{i_0} or v_{i_1}, they are reduced to the corresponding existing one.

We now provide a brief example to illustrate how the automata differ when different constraint types are taken into account. In particular, we focus on `AnyHit` and `RowHit` constraints, since they have been relatively neglected in the scientific literature. We consider a set $\Lambda = \{\lambda_1, \lambda_2\}$ where $\lambda_1 = \binom{1}{3}$ and $\lambda_2 = \left\langle {2 \atop 6} \right\rangle$.

Example 1 (Automaton for a Set of Constraints)**.** *Given the two weakly-hard constraints* $\lambda_1 = \binom{1}{3}$ *and* $\lambda_2 = \left\langle {2 \atop 6} \right\rangle$*, we apply the theorems presented in Deliverable D3.2 and confirm that there is no partial ordering between the constraints, i.e.* $\lambda_1 \not\preceq \lambda_2$ *and* $\lambda_2 \not\preceq \lambda_1$*. Following the steps in Algorithm 4, we generate the minimal automaton representations of the two constraints, as well as the automaton that represents the constraint set* $\Lambda = \{\lambda_1, \lambda_2\}$*, i.e.,* $\mathcal{G}_{\lambda_1}$*,* $\mathcal{G}_{\lambda_2}$*, and* $\mathcal{G}_{\Lambda}$*. The results are shown in Figure 11, where the leftmost, middle, and rightmost directed labeled graphs correspond respectively to* $\mathcal{G}_{\lambda_1}$*,* $\mathcal{G}_{\lambda_2}$*, and* $\mathcal{G}_{\Lambda}$*.*

The most important novelty presented in this work is the possibility to analyse weakly-hard constraint *sets* containing *all* the weakly-hard constraints types. Prior work proposed alternative solutions to the automaton generation problem, handling either a specific type of constraint [18], or a separate solution for each individual constraint type [12]. Being able to analyse sets of constraints in a scalable way brings us one step closer to the analysis of traces coming from real systems, in which the window lengths could be quite large and in which it is often easier to constrain the minimum number of hits (e.g., via execution in a protected environment without interference) rather than the maximum number of misses.

Table 2: Functions offered by `WeaklyHard.jl`.

Function	**Description**
`AnyHitConstraint(x, k)`	Defines a constraint $\lambda = \binom{x}{k}$
`AnyMissConstraint(x, k)`	Defines a constraint $\lambda = \overline{\binom{x}{k}}$
`RowHitConstraint(x, k)`	Defines a constraint $\lambda = \left\langle {x \atop k} \right\rangle$
`RowMissConstraint(x)`	Defines a constraint $\lambda = \overline{\langle x \rangle}$
`is_satisfied(Lambda, w)`	Returns **true** if $w \vdash \Lambda$, i.e., if the word w satisfies all the constraints in Λ, and **false** otherwise (note: can be invoked also passing a single constraint λ as parameter)
`is_dominant(lambda1, lambda2)`	Returns **true** if $\lambda_1 \preceq \lambda_2$ and **false** otherwise
`is_equivalent(lambda1, lambda2)`	Returns **true** if $\lambda_1 \equiv \lambda_2$ and **false** otherwise
`dominant_set(Lambda)`	Returns $\Lambda^* \subseteq \Lambda$
`build_automaton(Lambda)`	Returns the automaton $\mathcal{G}_\Lambda$ (note: can be invoked also passing a single constraint λ as parameter)
`random_sequence(G, N)`	Returns a word w, $\lvert w \rvert = N$ obtained through an N-step random walk in $\mathcal{G}_\Lambda$
`all_sequences(G, N)`	Returns the satisfaction set $\mathcal{S}_N(\Lambda)$ corresponding to $\mathcal{G}_\Lambda$

5.1.3 `WeaklyHard.jl` functionality

The most relevant user-exported functions provided by `WeaklyHard.jl` are summarised in Table 2.[2] In addition to the automata generation, the toolbox provides functions to compare constraints and obtain answers about their dominance and equivalence, to reduce a set of constraints to their dominant subset, and to generate sequences of arbitrary length satisfying sets of weakly-hard constraints. Furthermore, the tool includes additional functions as syntactic sugar to simplify the user experience, that are excluded from the table as they do not add significant functionality.

[2]The code includes a README file that guides the user through the setup of the package and provides simple usage examples. The only prerequisite is the Julia interpreter and compiler, available at `https://julialang.org`. The code can be found at `https://github.com/NilsVreman/WeaklyHard.jl`.

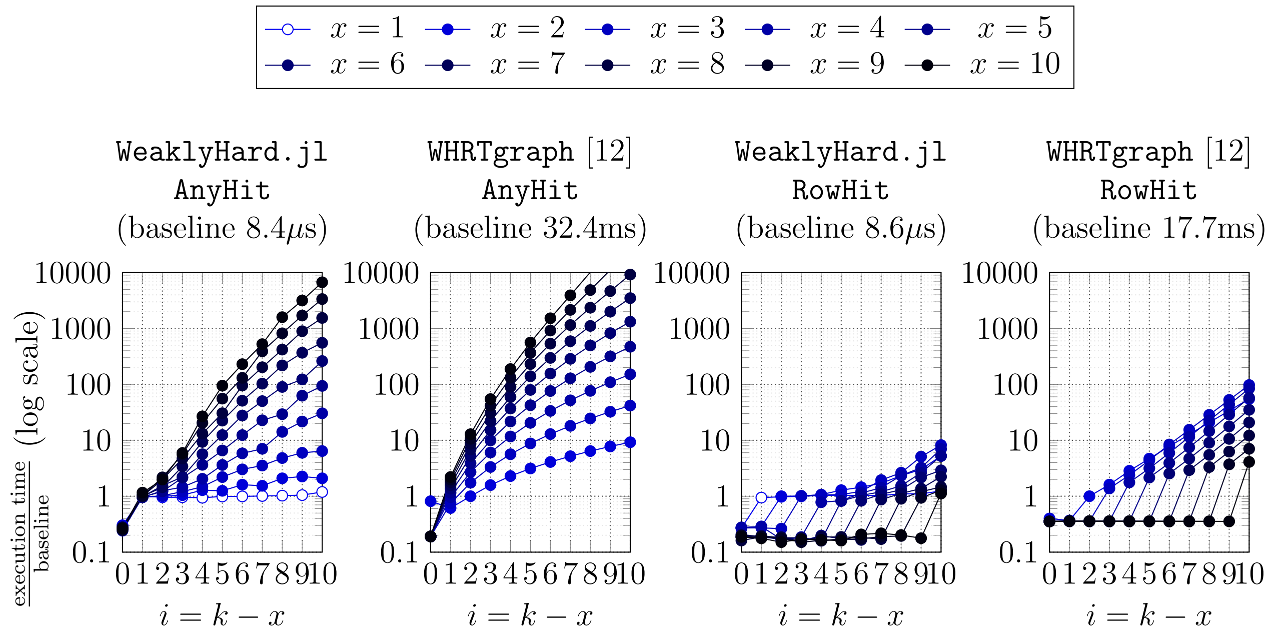


Figure 12: Execution time comparison for `AnyHit` and `RowHit` constraints with `WeaklyHard.jl` and `WHRTgraph` [12] increasing the difference between window size and number of hits constrained. Baseline values are reported on top of the corresponding plots.

5.2 Experimental results

We now present some brief extract of our experimental results.[3] First, we assess the scalability of the automaton generation code, comparing `WeaklyHard.jl` with the state-of-the-art, `WHRTgraph` [12, 13]. Then, we conduct a sensitivity analysis of `WeaklyHard.jl` to determine which parameters affect the execution time for the automata generation in cases that cannot be handled with other tools, i.e., sets of weakly-hard constraints. We provide results on how the type of constraints, maximum window length, and constraint set cardinality affect the scalability of the automaton generation. Finally, we investigate the average cardinality of the dominant constraint set Λ^* as a function of the cardinality of the original set, $|\Lambda|$.[4]

5.2.1 Comparing `WeaklyHard.jl` and `WHRTgraph`

The literature contribution that is closest to our research is `WHRTgraph` [12, 13]. `WHRTgraph`'s analysis of weakly-hard tasks is also based on the construction of automata. While `WHRTgraph` handles only one weakly-hard constraint at a time, it can construct the automaton that corresponds to `AnyHit` and `RowHit` constraints, making it the reference in terms of analysis capabilities. `WHRTgraph` is implemented in MATLAB, while `WeaklyHard.jl` is implemented in Julia. Hence, comparing the

[3]All the reported experiments ran on an Intel Xeon E5-2620 v3 @ 2.40GHz CPU with 126GB RAM memory.

[4]Note that the dominant set of constraints has been introduced in Deliverable 3.2 and the method to determine which constraints belonging to the original set Λ are included in Λ^* was presented as part of our theoretical analysis.

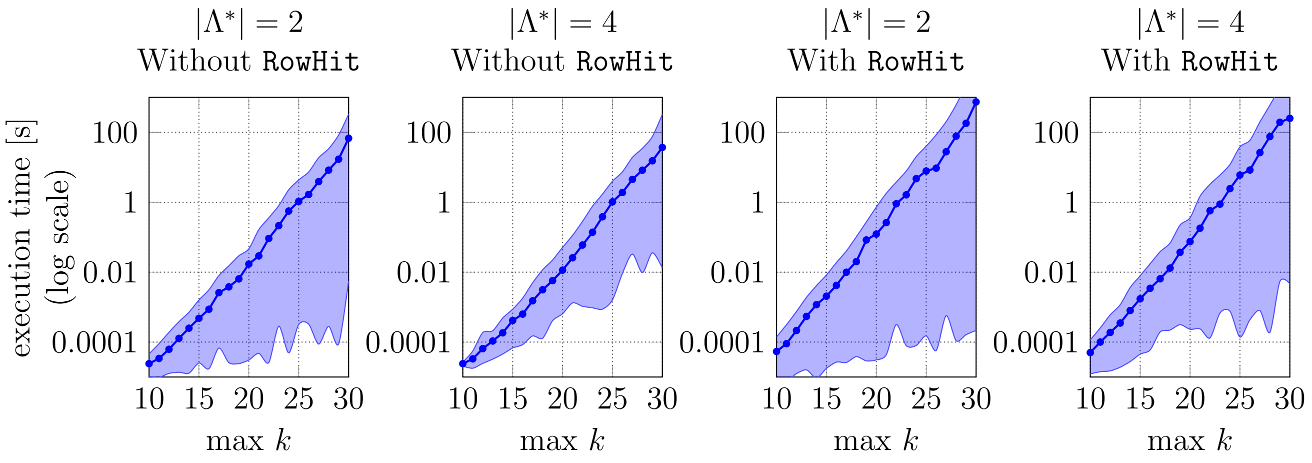


Figure 13: Execution time comparison for the generation of the automaton for sets of constraints with increasing maximum window sizes max k. Average values are reported alongside the areas between minimum and maximum execution times.

execution times of the two (on their own) is pointless. Furthermore, we are more interested in assessing the scalability to an increase in the constraint window size than the absolute numbers for the execution times. We therefore define a baseline case, for a fair comparison, i.e., the reported results are fractions and multiples of the baseline, which is different for each tool and constraint type.

To test the scalability of the automaton generation, we ask both `WeaklyHard.jl` and `WHRTgraph` to generate the automata that correspond to the `AnyHit` $\binom{x}{k}$ and `RowHit` $\left\langle {x \atop k} \right\rangle$ constraints for $x \in \{1, 2, \ldots, 10\}$, $k = x + i$ and $i \in \{0, 1, \ldots, 10\}$. We divide the obtained results by the baseline value, i.e., the execution time needed for the corresponding tool to generate the automaton for the given constraint type, $x = 2$ and $k = 4$.[5]

Figure 12 shows the mean value of the execution time for the automaton generation, divided by the corresponding baseline value, using a logarithmic y-axis. The baseline computation times for `AnyHit` constraint are 8.4μs for `WeaklyHard.jl` and 32.4ms for `WHRTgraph`. On the contrary, for a `RowHit` constraint, the baseline computation time is 8.6μs for `WeaklyHard.jl` and 17.7ms for `WHRTgraph`. Due to the extensive computational time necessary to build the automata using `WHRTgraph`, each automaton was built 30 times (i.e., each point in the figure is the mean of 30 execution times). `WeaklyHard.jl` is significantly faster, thus, each automata was built 100 000 times to reduce the variance in the resulting execution time.

`WHRTgraph` represents a weakly-hard constraint with an automaton that is slightly different, yet equivalent to the one we generate in `WeaklyHard.jl`. In particular, the automaton generated by `WHRTgraph` has fewer vertices and each transition represents the number of consecutive deadline

[5]The choice of the baseline case reflects the simplest constraint that is correctly handled by both `WeaklyHard.jl` and `WHRTgraph`. Comparing the methods, we unveiled that `WHRTgraph` is unable to find an automaton for constraints in which $x = 1$. The two plots for `WHRTgraph` in Figure 12 do not contain results for $x = 1$ (white filled markers) precisely due to this problem.

misses allowed between the vertices. Thus, a transition between two vertices in `WHRTgraph` is not equivalent to one outcome (as for `WeaklyHard.jl`), reducing flexibility; multiple successive outcomes for each transition make it difficult to handle sets of weakly-hard constraints. At a first glance, an automaton representation with fewer nodes sounds more efficient. However, we show that `WeaklyHard.jl` scales better than `WHRTgraph` by more than an order of magnitude (and, while this is less relevant, the baseline numbers also show that `WeaklyHard.jl` is significantly faster).

Comparing the scalability of the different tools applied to the `AnyHit` constraints (leftmost plots), the computational time complexities follow similar trajectories. However, we observe that `WeaklyHard.jl` is more than an order of magnitude faster than `WHRTgraph`. The same is true for the `RowHit` constraint graphs (rightmost plots). The speedup is due to the efficiency-improving features included in `WeaklyHard.jl`. Additionally, despite the substantial amount of tests conducted with `WeaklyHard.jl`, the execution time of `WHRTgraph` seem to follow a smoother trajectory. This is due to the baseline values for `WeaklyHard.jl` being in the microsecond-range, i.e., if the mean computation time for an automaton varies between 1ms and 2ms (e.g., due to extensive garbage collection, or other types of noise like network interference) the value reported in Figure 12 would jump from 100 to 200.

Overall, we conclude that even for a single constraint, treating scalability as a first-class citizen allows us to reduce the execution time of the automaton generation by an order of magnitude compared to the state of the art. In the following subsection, we focus on what `WeaklyHard.jl` offers that is novel compared to the state of the art, i.e., the analysis of sets of heterogeneous weakly-hard constraints.

5.2.2 Analysing sets of weakly-hard constraints

`WeaklyHard.jl` is the first tool that provides the ability to analyse sets of weakly-hard constraints. In the following we conduct a sensitivity analysis to assess the scalability of the automaton generation for a set of weakly hard constraints. In particular, we are interested in finding how the window size affects the execution time of the tool, and how the composition of the set influences the execution time.

We therefore conduct an experimental campaign, varying the input parameters and measuring the execution time for the automaton generation. We randomise minimal dominant sets of constraints, imposing that at least one of the constraints has a window size of $k \in \{10, 11, \ldots, 30\}$ and denoting this value with $\max k$. We generate sets with either $|\Lambda^*| = 2$ or $|\Lambda^*| = 4$. We allow these sets to include one `RowHit` constraint or none, to test our conjecture that the inclusion of `RowHit` constraints leads to an increase in the execution time for the automaton generation.

The results of our study are shown in Figure 13. For each of the values of $\max k$ in the figure, we generate 50 dominant sets of constraints Λ^*. The figure shows the average execution time in seconds (as a line) and the area representing the span between minimum and maximum execution time values.

The first conclusion that we can draw is that the average execution times follow straight lines in a logarithmic scale, thus clearly pointing to the exponential time complexity that follows an increase of the maximum window size of the constraints. As mentioned in previous work [16], the complexity is inevitable when working with such expressive task models.

When the cardinality of the set $|\Lambda^*|$ increases (i.e., comparing the two leftmost plots and the two rightmost plots with one another) we do not experience a significant change in the maximum execution time. In our experiments, we increase the cardinality of the set Λ^* without introducing dominated constraints. The additional constraints will very often just prune the resulting automata. In fact, states that would have been reachable with fewer constraint become unreachable due to the additional constraints. However, we experience a slight reduction of the execution times variance. The reduction in the variance is explained by the nature of the dominant set. We compare two dominant sets, Λ_1^* and Λ_2^*, with the same $\max k$. When $|\Lambda_1^*| = 2$ and $|\Lambda_2^*| = 4$, the set Λ_2^* includes constraints that are less restrictive (otherwise they would dominate the other constraints in the set). Hence, the set Λ_2^* is less likely to be trival to analyse, with the automaton generation taking longer time in general. Incidentally, this is also the reason why we believe analysing sets that include more than 4 weakly-hard constraints would not give additional insights on `WeaklyHard.jl`'s scalability.

Finally, when we include a `RowHit` constraint in the set Λ^*, the execution time increases by more or less an order of magnitude. This is unsurprising and follows from `RowHit` constraints being more difficult to analyse and enforce. In fact, combining the `RowHit` constraint with the other weakly-hard constraints reduces the number of language-equivalent vertices in the automata and hence increases the execution time of the algorithm. This is further reinforced by the fact that when a dominant set Λ^* includes a `RowHit` constraint, this implies that the other constraints in the set have to be very conservative in order to neither dominate nor be dominated by it.

Overall, we remark that `WeaklyHard.jl` is able to generate an automaton for a set Λ^* of 4 constraints with $\max k = 30$, including a `RowHit` constraint, in less than 200 seconds. This is achieved thanks to scalability being treated as a first-class citizen and enables the analysis of constraints with large windows.

5.2.3 Dominant constraint set

In Section 5.2.2 we investigated dominant sets Λ^* with cardinality $|\Lambda^*| \in \{2, 4\}$. Here we justify why this is a relevant benchmark despite the low cardinality.

We select a maximum window size $\max k = 100$. The window size is large enough that we can find an expressive variety of constraints without partial ordering. We randomly generate sets Λ containing $|\Lambda| \in \{1, \ldots, 100\}$ constraints. For each value of $|\Lambda|$ we generate 1000 different sets, excluding all the trivial constraints that would reduce to $\overline{\lambda}$ and $\underline{\lambda}$. We then compute the minimal dominant set Λ^* corresponding to each set. Figure 14 shows the average cardinality of Λ^* (solid line) and the experienced range (area).

As can be seen, most constraint sets reduce to minimal dominant sets with cardinality less than 4, therefore motivating the relevance of our investigation of the automaton generation execution time. Generally, it is also interesting to see that the generation of additional constraints tends to reduce the cardinality of Λ^* after a peak is reached. This is not surprising, as adding a new random constraint increases the chances of the added constraint being dominant over some of the constraints in the set.

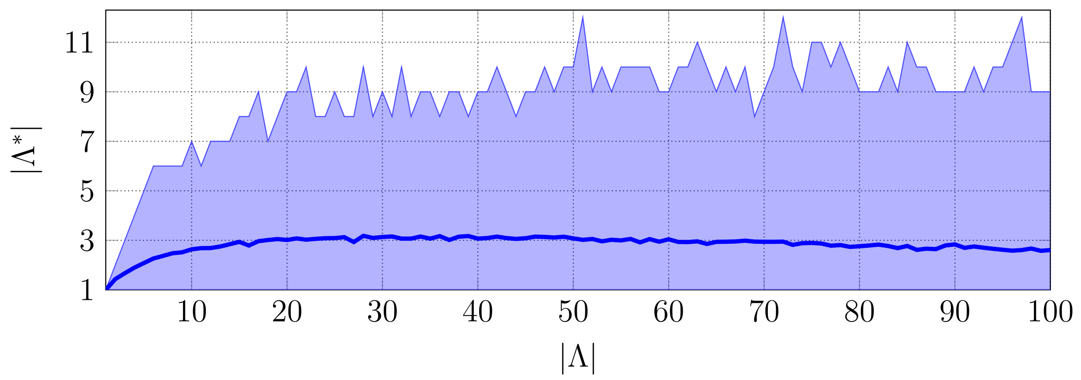


Figure 14: Average cardinality of the dominant set Λ^* as a function of $|\Lambda|$ with $\max k = 100$ for 1000 randomly generated constraint sets Λ.

5.3 Conclusion

The research behind this work package is motivated by the attention the weakly-hard model is receiving both in academic and in industrial research. We have developed `WeaklyHard.jl`, an open-source tool for the analysis of weakly-hard tasks, including both functions to relate different weakly-hard constraints to one another and functions to generate automata-based models for the outcome of tasks that may miss some deadlines.

We envision `WeaklyHard.jl` to be used for understanding the relation between different weakly-hard constraints and multiple constraint types. As an example, to validate our conjectures on the relation between `AnyHit` and `RowHit` constraints, we used `WeaklyHard.jl` to generate all the sequences satisfying a given constraint. We then inquired about the satisfaction of another constraint via brute force testing. This gave us confidence in the formulation and proof of Theorems presented in Deliverable D3.2, that complete the relation graph that links weakly-hard constraints of different types.

This deliverable included an experimental evaluation focusing on the scalability of the most critical function in `WeaklyHard.jl`, the generation of the automaton representing a set of weakly-hard constraints. Furthermore, we analysed the dominance between different constraints and built minimal dominant sets of constraints. To the best of our knowledge, `WeaklyHard.jl` is the first tool that enables the analysis of tasks that satisfy sets of weakly-hard constraints.

6 Automatic validation of safety and security cases for adaptive systems

Servers and consumer electronics (such as PCs or smartphones) have since long been designed as generic platforms, which host many of different applications over their lifetime. With growing flexibility demands for embedded systems, this trend of providing a platform has sprung over to embedded systems. Embedded systems such as cars, trains, airplanes or e.g. satellites however have much more stringent security and safety requirements than consumer electronics. Thus, mixed criticality systems are systems that allow the coexistence of software of different criticality levels on the same hardware, while ensuring non-interference.[1, 17]

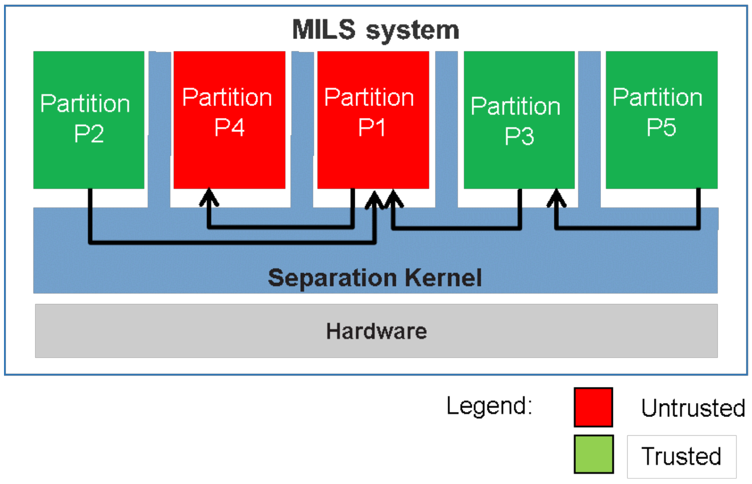


Figure 15: Example MILS system with partitions of different criticalities (red = untrusted, green =trusted)

Mixed critical systems allow to run many applications in parallel. The integrity of safety and security-critical systems is characterized by the need for non-interference. Non-interference is key to avoid failure propagation.

Multiple Independent Levels of Security and Safety (MILS) is a high-assurance security architectural approach for such mixed-critical systems based on the concepts of separation and controlled information flow, providing strongly separated execution environments (15). These execution environments ("partitions") can host e.g. standalone applications or entire guest operating systems (such as e.g. Linux). MILS has origins in security since the 1980s [15]. However, also safety is frequently included in the acronym's expansion, given that the provision of deterministic execution environments also covers non-interference e.g. protection from safety faults such as accidental denial-of-service ("babbling idiots"). The MILS design can enable effective and efficient compositional certification.

To be viable as a COTS product, a separation kernel is used in many types of different embedded systems. That is, for a specific embedded system (e.g. in a car, in an electricity counter, etc.), a system integrator buys a separation kernel and does an individual configuration suited to the needs of the system[3]. In particular, the configuration also allows to connect workloads in partitions together and to share resources (e.g. joint use of a network card or use of shared memory for sharing data between partitions). But conversely this ability means that a system integrator that does not understand or does not follow the guidance properly (by unintended configurations of sharing of resources) may introduce interference.

In principle the assignment of communication channels and resources to a separation kernel can be seen as a graph, as shown in Figure 16.

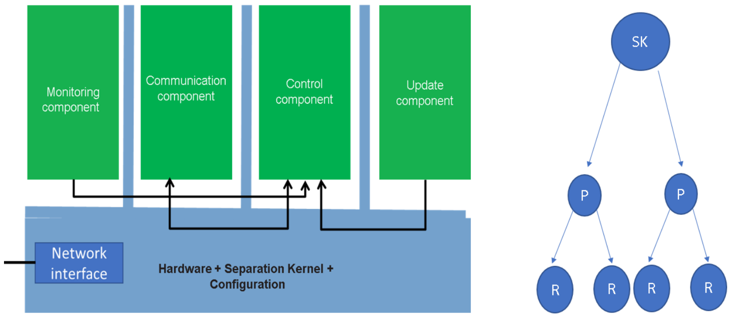
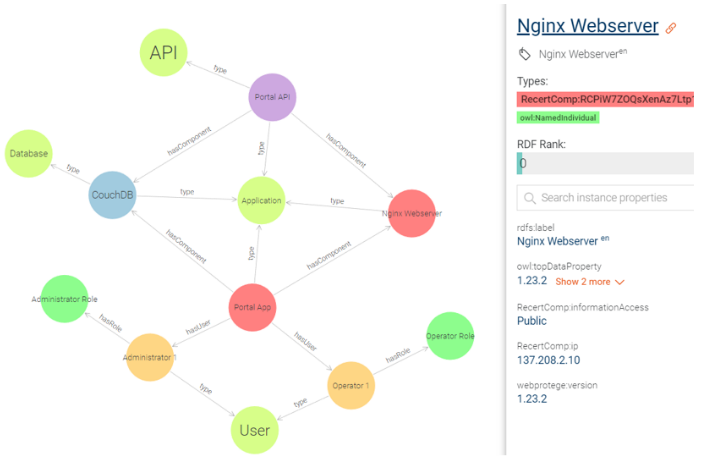


Figure 16: Resources; graph representation of resource allocation

In addition to the the resource configuration at the separation kernel, applications can have their own assignment of resources, as shown exemplarily for a webserver application in Figure 17.



Nginx Webserver
Nginx Webserver en
Types:
RecertComp:RCPiW7ZOQsXenAz7Ltp
owl:NamedIndividual
RDF Rank:
0
Search instance properties
rdfs:label
Nginx Webserver en
owl:topDataProperty
1.23.2 Show 2 more
RecertComp:informationAccess
Public
RecertComp:ip
137.208.2.10
webprotege:version
1.23.2

Figure 17: Graph of a webserver

Of course, these representations also can be combined, as shown in Figure 18.

The German IDEA project had previously experimentally encoded graphs in the neo4j graph database, which allowed us to visualise conflicts created by shared resources. Figure 19 shows the detection of a shared resource (in this case, a console used for debugging purposes). In this case having this shared console in a testing (debugging-enabled) system was intentional, but the system integrator should pay attention to not having this configuration in the production system, if a possible undesired interference channel is to be avoided.

In the ADMORPH project, we have automated the encoding of graphs: Our workflow now is to parse the XML configuration file of the separation kernel as XML DOM, push that data into graph analysis software (igraph in this case) and find shared resources by graph analysis. By application of the this approach we have created queries to igraph on selected resources that check for shared

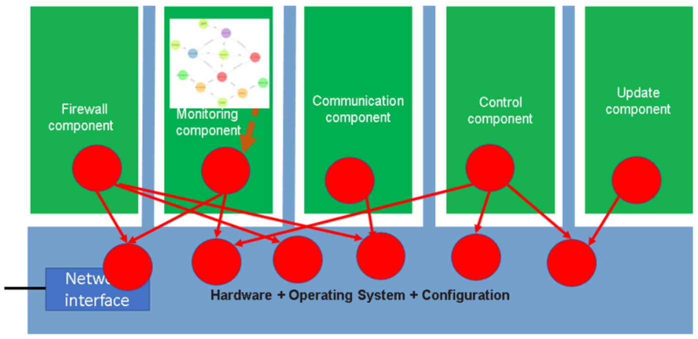


Figure 18: Graph of a webserver combined with graph of separation kernel

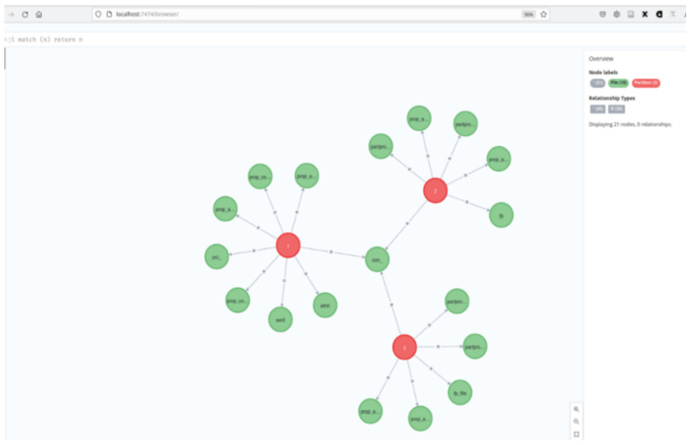


Figure 19: Graph analysis detects shared resource usage

resources. A system integrator can use this, to detect shared resources, and then (manually) has to judge whether the sharing was intentional or not.

When parsing the XML configuration, a lesson learned is that some of the configuration data refers to the local context, and other to a global resource context. This has to be taken into account. That is, when parsing configuration data (XML) globally shared resources must be detected and appropriately handled. We therefore added resource attributes to the XML notation to identify global resources, i.e. to distinguish global from local resources.

7 Conclusion

The research conducted in this work package has been presented in the following peer reviewed publications:

(i) Sobhan Niknam, Anuj Pathania, Andy Pimentel; T-TSP: Transient-Temperature Based Safe Power Budgeting in Multi-/Many-Core Processors; IEEE International Conference on Computer Design (ICCD 2021),

(ii) Nils Vreman, Richard Pates, and Martina Maggio; `WeaklyHard.jl`: Scalable Analysis of Weakly-Hard Constraints; IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2022).

The simulator, whose latest advance is described in Section 2, is available as open source software at `https://github.com/sea-art/Simuflage/`. `WeaklyHard.jl`, described in Section 5, is available as open source software at `https://github.com/NilsVreman/WeaklyHard.jl`, and included in the General Julia registry (`https://github.com/JuliaRegistries/General`).

When dealing with control systems, the joint spectral radius [11] is a very important tool. Combining the automaton generation presented in Task 3.6 with the joint spectral radius (as discussed in Work Package 1, Task 1.3) allows us to provide guarantees for control systems. This is included in the following publications:[6]

(i) Jie Wang, Martina Maggio and Victor Magron; SparseJSR: A Fast Algorithm to Compute Joint Spectral Radius via SparseSOS Decompositions; American Control Conference (ACC 2021).

(ii) Nils Vreman, Paolo Pazzaglia, Jie Wang, Victor Magron and Martina Maggio; Stability of Control Systems under Extended Weakly-Hard Constraints; IEEE Control Systems Letters and International Conference on Decision and Control (CDC 2022).

8 References

[1] Damir Bartakovic, Axel Söding-Freiherr von Blomberg, Oliver Kühlert, Tibor Jukić, Guillaume Fumaroli, Hans-Jürgen Herpel, Patricia Lopez Cueva, Juan-Maria Carranza, and Maxime Guy. Use of Multiple Independent Levels of Security and Safety (MILS) architecture for space on memory protection unit (MPU)-based systems. 2022.

[2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and V.B. Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[3] Bundesamt für Sicherheit in der Informationstechnik (BSI). BSI-DSZ-CC-1146-2022: PikeOS Separation Kernel, Version 5.1.3.

[4] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and Tanaka Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. In *International conference on parallel problem solving from nature*, pages 849–858. Springer, 2000.

[6]Note that the following two publications do not belong to WP1, T1.3 but rather to WP3, T3.6, even though the work is clearly connected.

[5] Madalina M Drugan and Ann Nowe. Designing multi-objective multi-armed bandits algorithms: A study. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2013.

[6] Madalina M Drugan and Ann Nowé. Scalarization based pareto optimal set of arms identification algorithms. In *2014 International Joint Conference on Neural Networks (IJCNN)*, pages 2690–2697. IEEE, 2014.

[7] E. Fersman, P. Pettersson, and W. Yi. Timed automata with asynchronous processes: Schedulability and decidability. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 67–82, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[8] Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computing*, 205(8):1149–1172, 2007.

[9] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[10] C. Huang, K. Wardega, W. Li, and Q. Zhu. Exploring weakly-hard paradigm for networked systems. In *Proceedings of the Workshop on Design Automation for CPS and IoT*, DESTION '19, page 51–59, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Raphaël Jungers. *The joint spectral radius: theory and applications*, volume 385. Springer Science & Business Media, 2009.

[12] Steffen Linsenmayer and Frank Allgöwer. Stabilization of networked control systems with weakly hard real-time dropout description. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 4765–4770, 2017.

[13] Steffen Linsenmayer, Michael Hertneck, and Frank Allgöwer. Linear weakly hard real-time control systems: Time- and event-triggered stabilization. *IEEE Transactions on Automatic Control*, 66(4):1932–1939, 2021.

[14] Paolo Pazzaglia, Luigi Pannocchi, Alessandro Biondi, and Marco Di Natale. Beyond the Weakly Hard Model: Measuring the Performance Cost of Deadline Misses. In Sebastian Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:22, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[15] John Rushby. Design and verification of secure systems. In *Eighth ACM symposium on operating system principles*, pages 12–21, 1981. tex.lookup: Rushby1981design-and-verification-of-secure-systems.

[16] M. Stigge and W. Yi. Graph-based models for real-time workload: a survey. *Real-Time Systems*, 51:602–636, 2015.

[17] Sergey Tverdyshev, Holger Blasum, Bruno Langenstein, Jonas Maebe, Bjorn De Sutter, Bertrand Leconte, Benoît Triquet, Kevin Müller, Michael Paulitsch, Axel Söding-Freiherr von Blomberg, and Axel Tillequin. *MILS architecture*. EURO-MILS, 2013.

[18] E.P. van Horssen, A.R.B. Behrouzian, D. Goswami, D. Antunes, T. Basten, and W.P.M.H. Heemels. Performance analysis and controller improvement for linear systems with (m, k)-firm data losses. In *2016 European Control Conference (ECC)*, pages 2571–2577, 2016.

[19] M. van Osch and S.A. Smolka. Finite-state analysis of the can bus protocol. In *Proceedings Sixth IEEE International Symposium on High Assurance Systems Engineering. Special Topic: Impact of Networking*, pages 42–52, 2001.

[20] Haibo Zeng and Marco Di Natale. Schedulability analysis of periodic tasks implementing synchronous finite state machines. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 353–362, 2012.