D3.2 Second report on analysis techniques for adaptive systems

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

Due Date:	June 30th, 2021
Delivery:	Month 18
Lead Partner:	Lund University
Editor:	Martina Maggio
Dissemination Level:	Public
Status:	In progress
Approved:	
Version:	1.0

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
17/06/2021, v1.0	Martina Maggio	First complete draft

List of Contributors

Date and version number	Author	Comments
08/03/2021, v0.1	Martina Maggio	Initialization
10/06/2021, v0.2	Christoph Kühbacher	Section 4
12/06/2021, v0.3	Don Kuzhiyelil	Section 8
14/06/2021, v0.4	Nils Vreman	Section 7
15/06/2021, v0.5	Sobhan Niknam	Sections 2 and 3
16/06/2021, v0.6	Stefanos Skalistis	Sections 5 and 6
17/06/2021, v0.7	Martina Maggio	Introduction and Conclusion
30/06/2021, v1.0	Martina Maggio	Internal review comments

Contents

Executive summary

This deliverable is a report on the consortium's work in Work Package 3, discussing all the tasks in the work package: Task 3.1 *System-level simulation of dynamically evolving embedded systems*, Task 3.2 *Design-Space Exploration of Dynamically Evolving Embedded Systems*, Task 3.3 *Adaptivity-aware real-time scheduling policies*, Task 3.4 *Models of Computation and derived architectures to allow seamless reconfiguration*, 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. The report presents the conclusion of Task 3.1 and the status near to conclusion of task 3.4, a midpoint evaluation for Task 3.3, and the start of 3.2, 3.5 and 3.6.

1 Introduction

In this report we provide an overview of the tasks connected to the analysis of self-adative system. In particular, this work package focuses on how to evaluate the behaviour provided by these systems at runtime.

In particular, this report focuses on preliminary work conducted to:

(i) Design a simulation tool that allows us to compare different scenarios and their expected performance characteristics. This simulation should be conducted at the system-level and should include the system - hardware and software - and its evolution. From the theoretical perspective, we use Monte Carlo (MC) simulation as the main tool to perform many different simulations in variable environments and with varying characteristics.

(ii) In parallel with the modeling and simulation of the system level characteristics of our embedded systems, we need to derive models of computation. We started to work on the derivation of models that include tasks with precedences (for example: we need to aquire a new image from a camera before being able to understand if something has happened with respect to previous frames). This is per se not new, but the novelty with respect to existing work lays in this precedence list to be dynamically updated at runtime. For example, due to an attack, we may determine that we need to aquire frames from different cameras in order to be certain that nobody has tampered with the image acquisition process. At runtime our morphing system will be able to recognise this, so we need to be able to handle the computation in different scenarios.

(iii) Once we understood the varying characteristics of our computation model, we need to be able to handle these characteristics in terms of resource distribution. For example, two cores may be needed to perform the same analysis on two different images simultaneously, such that our answer can still be produced in real-time. Maybe, we would find that two cores are not enough, and we need to use dedicated hardware like FPGAs or a graphic card for the image processing in order to meet the original application requirements.

(iv) Finally, we want to be able to formally guarantee the behaviour of reconfigurable and morphing systems.

The next sections will enter into details on what has been done for each of the work package objectives. In particular, Section 2 discusses the system level simulator developed in Task 3.1, Section 3 enters into details on the design-space exploration problem that allows us to determine and execute systems in the most appropriate system configurations from Task 3.2. Section 4 provides results on the study conducted in Task 3.3 of adaptive scheduling policies. Section 5 discusses our results on the model of computation that we intend to use from Task 3.4. Section 6 briefly recaps our intention for timing analysis in Task 3.5. Section 7 discusses our initial results on providing formal guarantees for the adaptation layer, obtained in Task 3.6. Section 8 discusses our initial investigation from Task 3.7 on how to automatically provide safety guarantees for certification and validation.

2 System-level simulation of dynamically-evolving systems

In this section, we first briefly recap the main characteristics of the system-level simulator that we developed to analyze and evaluate adaptively morphing embedded systems. The simulator has been extensively discussed in Deliverable D3.1 and (for conciseness) we assume the reader is familiar with the content of the previous deliverable. Then, we present the results of our extensive experimental campaign to validate the functionality of the simulator.

2.1 Simulator overview

In the previous D3.1 report, we talked about the need for the run-time self-adaptivity of an embedded system to ensure reliability and/or prolong the system lifetime. We further explained that several factors, e.g., the number and types of processing cores in the underlying platform and adaptivity policy, need to be decided during the early stage of the designing process of such an adaptive embedded system, causing a potential explosion in terms of size of the design space. For exploring such a design space and possibly finding near-optimal design instances, we further introduced our novel fast system-level simulator, called *Simuflag*. The main structure of this simulator is shown in Figure 1. Here, we briefly remind the reader how this simulator works. The full details were given in D3.1.

The simulator takes two types of inputs: 1) system-level modeling of an adaptive system, and 2) parameters related to the environment and the context that the system will be evaluated in. More specifically, the model of the adaptive system consists of a platform, application workload(s) together with initial application(s) mapping on the platform, and an adaptivity policy to handle processor failures. On the other hand, the contextual parameters represent the environment in which the simulator operates and can be altered within the simulator concerning the user's needs. For instance, some contextual parameters are the environmental temperature, fault model(s), fault distribution, neighbor thermal influences, static power consumption for the processors on the platform. Then, the simulator evaluates the time to failure (TTF) and power/energy consumption of the system concerning the contextual parameters. These parameters are essential for assessing the goodness of different system-level designs of mission- and safety-critical systems, targetting in the Admorph project, that require a long operational time.

The simulator works in timesteps, i.e., the time between each two consecutive processor failures, where during each timestep several calculations are performed to advance the given system to the next timestep. A system that is being simulated is always in one of the three phases: functioning correctly, handling a failure, or has failed. The system is always initialized in the first phase by default. Once a failure has occurred, concerning the given ageing model, the system will move to the second phase. In the second phase, there are two possibilities. The applications on the failed processor can either be successfully remapped to some of the alive processors concerning the policy, which results in the system returning to the first phase, or one or more applications can not be mapped on any alive processor. When the latter occurs, the system has permanently failed and enters the third phase when the simulator will report the desired outputs.

The essence of the simulator is the combination of three individual models that are interacting

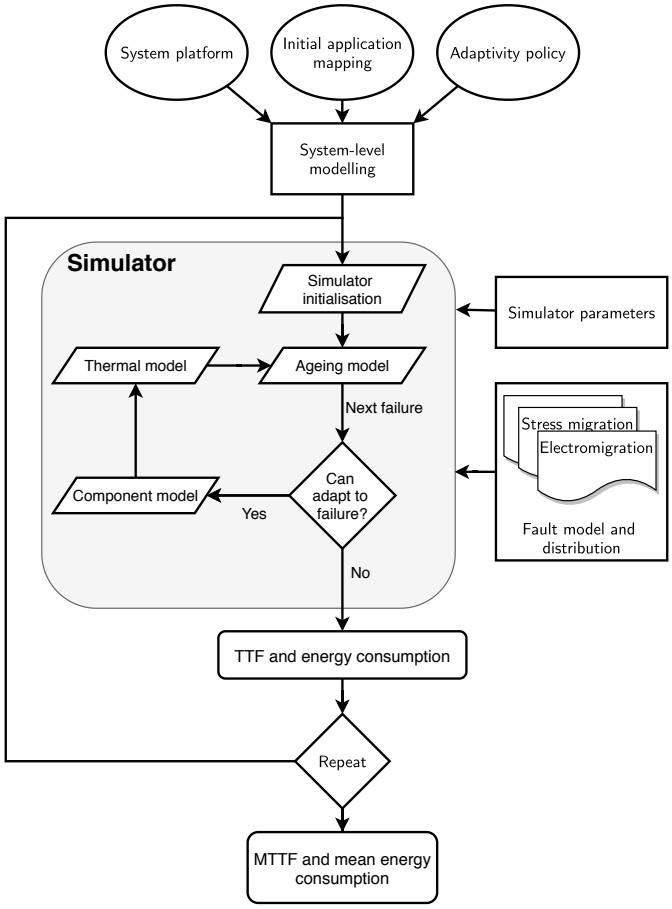


Figure 1: Schematic overview of our proposed simulation framework

with each other in order to simulate the behavior of an adaptive system over the course of time. The three models are, respectively, the platform, thermal, and ageing models. The algorithm of the simulator will work in timesteps where in each timestep the TTF will be accumulated and the models will be updated based on the system state (e.g., any updated thermals will affect the ageing of the system).

2.2 Experimental setup

2.2.1 Selected parameters

As described in subsection 2.1, our simulator framework provides a certain degree of freedom in terms of parameters and models that can be utilized. In all our experiments, the component faults are introduced via the electromigration (EM) fault model (Section 2.1 in the D3.1 report) which are distributed according to the Weibull distribution using the shape parameter of $k = 5$. We use an environmental temperature of 20 °C without any thermal fluctuation (Eq. (18) in D3.1). The neighbour convolution kernel (Eq. (15) in D3.1) has been set to 0.1 for all adjacent neighbours.

Table 1: Hardware specifications of the computing platforms used to obtain the results for the experiments.

	Rhino	**Uvalight**	**DAS5 cluster**[1]	**Personal desktop**
CPU	Intel Xeon E5-2650 v4	Intel Xeon Silver 4216	Intel Xeon E5-2630 v3 (dual)	AMD Ryzen 5 3600
Cores	12	16	16	6
Threads	24	32	32	12
Frequency[2]	2.2 - 2.9 GHz	2.1 - 3.2 GHz	2.5 - 3.2 GHz	3.6 - 4.2 GHz
Memory	64GB	4GB	64GB	32GB
Operating system	Ubuntu 18.04	Debian 10 (buster)	CentOS 7.4	Ubuntu 18.04

The parameters for the modeled platforms and applications (i.e., design points) are as follows:

(i) three types of processor (CPU) components,

(ii) four distinct applications that have to be executed,

(iii) selection of between 1 up to 20 CPUs,

(iv) placement of the CPUs on at most a 6×6 grid,

(v) selection out of three adaptivity policies: most-slack first, least-slack first and the random policy (which will all be discussed later on), and

(vi) selection of a heuristic algorithm for an initial application mapping among best-fit, worst-fit, first-fit or next-fit algorithms (also explained later on).

2.2.2 Hardware

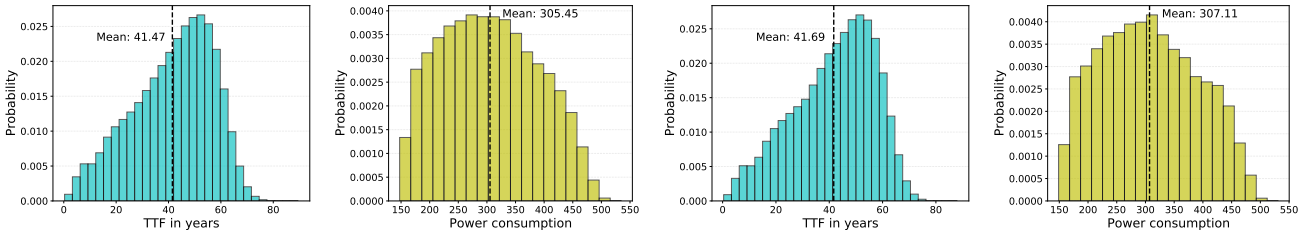
Running the simulator to evaluate multiple design points are independent tasks. Thus, to speed up the simulation time, multiple simulations can be performed in parallel on different computing resources. Table 1 lists the four computing resources that we use with their most important hardware specifications.

[1] `https://www.cs.vu.nl/das5`

[2] Listed as base frequency - max turbo frequency

Table 2: Information of the two datasets utilised for the experiments in subsection 2.3.

	Number of design points	**Samples per design point**	**Total number of samples**	**95% confidence interval**	
				MTTF	Power
Dataset 1	70,000	1,000	70,000,000	± 0.19414 years	± 0.47634
Dataset 2	2,800	10,000	28,000,000	± 0.06082 years	± 0.15073



(a) TTF Dataset 1 (b) Power Cons. Dataset 1 (c) TTF Dataset 2 (d) Power Cons. Dataset 2

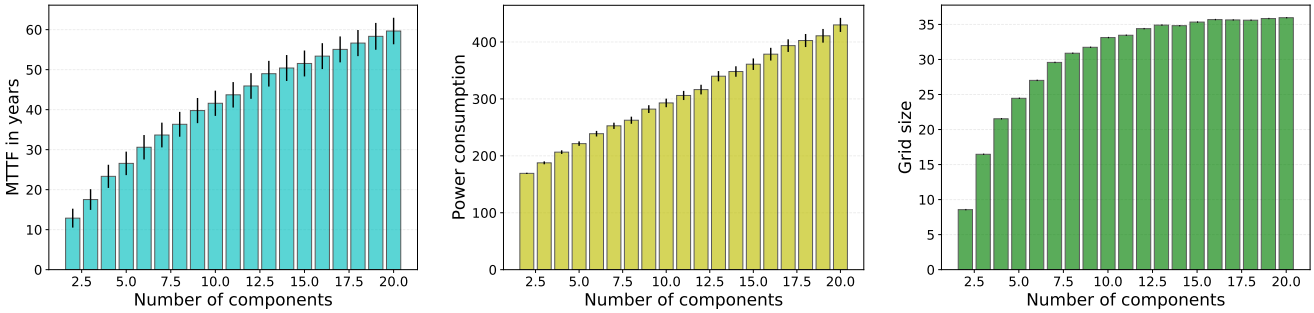
Figure 2: The distribution of the computed TTF and power consumption for all samples of dataset 1 and 2.

2.2.3 Datasets

As described earlier, utilizing the simulator frequently for many design points is a time-consuming process. To prevent evaluating each design point repeatedly[3], we have created two distinct datasets. These datasets include many random design points, which will all be sampled frequently and stored in a list. We will use the design points from these datasets in our experiments and utilize the dataset as a lookup table. In this way, if k samples are required, we can take k random elements from the list containing the samples of the design point.

The first dataset contains 70,000 random design points, where each design point has been sampled 1,000 times, resulting in a total of 70,000,000 samples. This dataset is mainly focused on receiving a high number of design points with a reasonable number of samples to calculate the means relatively accurately. The second dataset contains 2,800 random design points with 10,000 samples per individual, resulting in a total of 28,000,000 samples. The main function of the second dataset is to have more samples per design point, allowing us to more accurately calculate the means by taking more samples. Figure 2 provides an overview of how the TTF and power consumption of all samples are distributed in both datasets. The computing platforms listed in Table 1 are utilized for the collecting of the data for these datasets.

[3]Each evaluation is referred to as a *sample*.



(a) MTTF (b) Power consumption (c) Grid size

Figure 3: Performance of design points on the three objectives as the number of computational resources increases. The graphs show the mean objective values of the design points from dataset 1, grouped by the number of components.

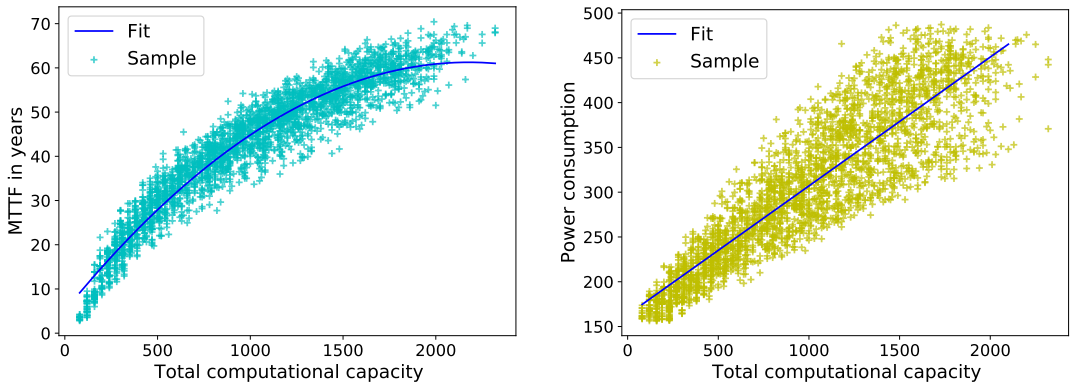
2.3 Simulator validation

In this section, we present the results of experiments conducted for validating the functionality of our simulator. The simulator validation is based on comparing the outputs of the simulator, using both dataset 1 and 2 (Table 2), with logical expectations. The main purpose of the simulator validation is to check if our simulator provides a feasible exploration space, wherein choices in the solution space have logical influences on the objective space, i.e., system performance, power consumption, mean time to failure (MTTF), etc. We also provide a simple validation experiment by comparing our simulator to a state-of-the-art simulator.

2.3.1 Selection and placement of computational resources

The selection and placement of computational resources have a significant influence on the MTTF and power consumption objectives. Since all of the design points from our datasets are running the same number of applications, selecting more than four components will result in one or more processing cores functioning as spare components. These components can be reserved until when one or more working components are worn out and permanently failed. Then, the applications are remapped from the failed components to spare ones in order to adapt the system and resume its functionality. The power consumption is also expected to increase. Our presented power consumption model (Eq. (12) in the D3.1 report) is dependant on the number of idle components using the specified idle power consumption.

Figure 3 presents how the objectives are affected as the number of components is increasing. For the MTTF, we can see in Figure 3(a) that it increases as the number of components increases. However, it does seem to curve off, which also makes sense from our ageing model (Section 2.2.2.4 in D3.1). Idle components are still slowly wearing-out, due to environmental and idle temperatures. For this reason, components that remain idle for many years are already significantly worn out. In our model, the effectiveness of having spare components will decrease after a long enough



(a) MTTF (b) Power consumption

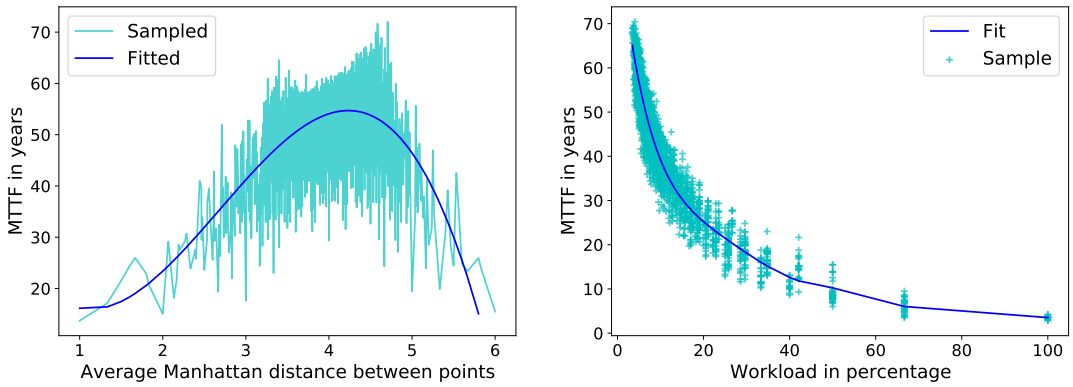
Figure 4: Average MTTF and power consumption as the total sum of computational capacity of design points increases. Taken from 2,800 random design points with 10,000 samples each.

period. The power consumption in Figure 3(b) seems to increase linearly based on the number of components. The main fact to notice is that the standard deviation (indicated as a line on top of the bars) seems to increase as the number of components increases. This can be explained by the fact that having more components increases the number of component failure scenarios, which greatly influences the power consumption and its standard deviation. The grid size (Figure 3(c)) is mainly dependant on the random design point creation algorithm rather than the simulator itself. Since design points are randomly placed on a 6×6 grid (according to the aforementioned design space for the modelled platforms), the chance for inefficient random placements is increased by increasing the average grid size.

The number of components is only one influential factor on the objectives. The kind of computational components that have been selected is also influential since certain types of processing cores are more powerful (i.e., have a higher computational capability). Figure 4 combines these two factors and shows the MTTF and power consumption objectives based on the total number of computational capacity. Both the graphs show a similar shape and curvature as shown in Figure 3. The variance as the computational capacity increases is also more easily visible in Figure 4(b), which has a direct correlation with the number of components selected.

Figure 5(a) shows how the average distance between components affects the average MTTF of design points. Since our model captures the effect of neighbouring thermal influences, having components closely placed together should decrease the MTTF. From this figure, the MTTF seems to first start increasing, which after a certain average distance between processing cores decreases again. The average Manhattan distance between points is constructed by taking the sum of the Manhattan distance between all unique pair of locations on the board, i.e.

$$\sum_{i=0}^{m-1} \sum_{j=i+1}^{m} \mathrm{d}(i,j),$$



(a) MTTF per average distance

(b) MTTF per workload

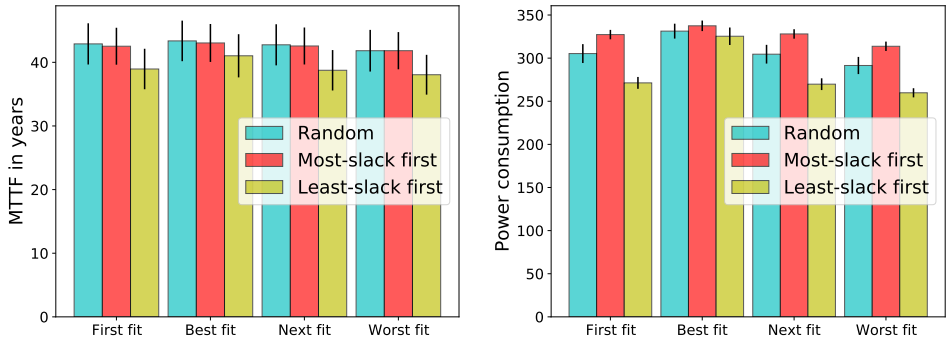
Figure 5: This figure shows how the MTTF is affected by both the average distance between processing cores and the workload in percentage, using dataset 1 (Table 2). The distance is plotted between the interval 1 to 6. The workload is calculated by dividing the sum of applications by the total computational capacity.

where m is the number of components within the design point and d(i, j) is the Manhattan distance between component i and j, of which the sum is divided by the total number of pairs. The very first and last points of this graph show that a low and high average distance results in a low MTTF. This is mainly because this only occurs with a low number of selected components thus having only a few spare components.

Figure 5(b) shows how the workload affects the MTTF in years. Note that all of our faults are temperature-dependent and the temperature is mainly determined by the workload. Therefore, there is a very direct correlation between the workload and the probability of faults occurrence. In this graph, the workload seems to get more spread out on the right side, which is inherent to our provided applications and components in the search space.

2.3.2 Adaptivity policy and initial application mapping selection

As mentioned in the D3.1 report, an adaptivity policy is a function that will migrate applications running on a failed component to other working ones at run-time. In our experiments, we use three specific policies namely *most-slack-first*, *least-slack first*, and *random* that will just remap the applications randomly. The most-slack first policy (Algorithm 1) will sort the still functional components based on their amount of slack (Eq. (10a) in D3.1 report) and will map the applications to the component with the highest slack. Once an application is mapped, the sorting has to be updated. The core concept of most-slack-first is to uniformly distribute the applications as much as possible among functional components. The least-slack first policy (Algorithm 2) does the exact opposite as the most-slack-first policy, where the applications will be remapped to the components that have the least slack. This will result in keeping the components with the highest slack as spare components while trying to map most of the applications to as few components as possible.



(a) MTTF (b) Power consumption

Figure 6: Average MTTF and power consumption objectives of the design points of dataset 1 according to their adaptivity policy and initial application mapping function.

Similarly, as mentioned in the D3.1 report, initial application mapping indicates which applications should be executed initially on which processor. However, the task mapping is analogous to a bin-packing problem which is known to be an NP-hard problem. To this end, in this section, we use a set of well-known heuristic algorithms for solving the bin packing problem, namely *next-fit*, *first-fit*, *best-fit*, and *worst-fit* algorithms.

Algorithm 1 Most-slack first policy

```
1: procedure MOST_SLACK_FIRST(C, a_i)
2:     C_sorted ← sort C based on most slack
3:     for c_i ∈ C_sorted do
4:         if slack of c_i > a_i^req then
5:             add (a, c_i) to f_ac
6:             break
7:         else
8:             system failure
9:         end if
10:    end for
11: end procedure
```

Algorithm 2 Least-slack first policy

```
1: procedure LEAST_SLACK_FIRST(C, a_i)
2:     C_sorted ← sort C based on least slack
3:     for c_i ∈ C_sorted do
4:         if slack of c_i > a_i^req then
5:             add (a, c_i) to f_ac
6:             break
7:         end if
8:     end for
9:     if a could not be mapped then
10:        system failure
11:    end if
12: end procedure
```

Now, after defining the policy and initial task mapping algorithms, our goal in this section is to look at the influences of all possible combinations of these algorithms on the MTTF and power consumption objectives. This matter is shown in Figure 6. When looking solely at the performance of the policies, we can observe that the least-slack first policy is the best choice when the low power consumption is prioritized by sacrificing some lifetime of the system. The concept of least-slack first is that the policy tries to utilize the least number of processing cores as possible and prefers to let fewer processing cores work with a higher workload. In our current model, this concept will

Table 3: Comparison of the lifetime evaluation results, in years, for our simulator and CALIPER [6] for a 2×2 homogeneous grid based on 10,000 samples.

	CALIPER [6]		**Our simulator**	
Workload	MTTF	SD	MTTF	SD
0.1	80.88	20.69	82.30	7.93
0.2	51.88	13.32	53.98	5.93
0.3	32.31	9.55	35.53	4.69
0.4	23.22	6.94	25.27	3.39
0.5	16.95	5.14	17.99	2.58
0.6	10.87	3.99	11.45	2.62
0.7	8.27	3.05	8.49	1.98
0.8	6.34	2.35	6.37	1.41
0.9	4.91	1.83	4.81	1.08

result in lower power consumption due to the low workload of most of the components but will wear through the available processing cores faster decreasing the longevity.

The most-slack first policy seems to be better at the MTTF objective, at the cost of having a higher power consumption. The concept of the most-slack first policy is the opposite to the least-slack first, where it tries to spread out the workload as much as possible to keep the workload per processing core at its minimum. Since processing cores are operating at much lower temperatures it increases the longevity, but having multiple non-idle processing cores active will result in higher power consumption in our current model.

The random policy seems to perform even better than the most-slack algorithm on the MTTF objective while also operating more efficiently regarding the power consumption. Out of this experiment, it seems that the random policy would provide the best trade-off between these two objectives. This might be because the other two policies are the 'extremes' and that somewhere in between most-slack first and least-slack first will provide the best trade-off.

Another observation in Figure 6 is that the initial application mapping using the worst fit heuristic algorithm results in the lowest power consumption and MTTF compared to other bin-packing heuristic algorithms. This observation can be interpreted with respect to the processing heterogeneity available on the platform. The processors with higher capacity are more performance efficient and thus consuming more power compared to power-efficient ones. Thus, more processors with different types are possibly exploited under the worst fit heuristic, which tends to distribute the workload uniformely over all processors on the platform, compared to other heuristics. As a result, the worst fit heuristic results in the lowest power consumption. On the other hand, more processors are simultaneously getting aged under the worst fit heuristic while leaving fewer spare processors, to be used later when one or more processors are permanently failed, compared to other heuristics. As a result, the worst fit heuristic results in the lowest MTTF.

While the results of these policies might not accurately correspond to reality, they can be

explained from our presented ageing and power consumption models. It also shows that the choice of policy has a distinct and significant effect on these two objectives, making the policy an important factor for optimizing design points during the DSE. The initial application mappings show quite similar results towards the MTTF objective, but do show a significant influence on the power consumption objective.

2.3.3 CALIPER simulator comparison

CALIPER [6] is a state-of-the-art simulator that is available as open-source[4] and allows us to use it for the purpose of comparison. CALIPER solely allows for homogeneous grid systems to be simulated, which our simulator is also capable of using the same capacity for all components and placing them adjacent to each other. We compare the lifetime output of our presented simulator to theirs for a 2×2 homogeneous system with a range of different running application workloads and failures that are introduced via EM distributed according to the Weibull distribution. The results are shown in Table 3, where SD stands for standard deviation.

The goal of this validation experiment is not to provide a thorough comparison but to illustrate that similar output for an arbitrary homogeneous design point can be obtained by using our simulator. The biggest deviation in lifetime output is at a lower workload because the adaptive behaviors of these two simulators are different. Overall, from a simple verification standpoint, our simulator shows similar MTTF output compared to CALIPER.

3 Design-space exploration of dynamically-evolving systems

Using our system-level simulator, introduced and evaluated in section 2, we aim as the next step to develop efficient design-space exploration (DSE) techniques that allow for determining appropriate system configurations, comprising both the hardware configuration and the adaptivity strategy. An efficient design-space exploration incorporates both efficient searching and evaluating of the design points in the design space. The design space exploration of an adaptively morphing embedded system is a multi-objective optimization process with a vast number of possible design candidates that can not be explored by exhaustive search. For this reason, meta-heuristics are used, where Genetic Algorithms (GA) have demonstrated to show promising results for this type of DSE [24, 27, 28]. This section is therefore devoted to introducing the concepts, ideas, and operators that are considered within the GA-based exploratory algorithm of this research. Next, as the future work in Task 3.2, we plan to evaluate system configurations using Monte-Carlo simulations and reinforcement learning techniques (like multi-armed bandit algorithms) for efficient sampling of the simulation model.

[4] `https://github.com/D4De/caliper`

3.1 Genetic Algorithms

GAs are heuristic search algorithms that apply to a wide range of problems due to their flexibility. The basis of this type of algorithm is the population, which represents a list of possible solutions to the problem that it tries to solve (in our case, a multi-objective DSE). The population is iteratively refined in so-called *generations*. Proceeding from one generation to the next involves four specific types of operators that are based on natural evolution. Each generation should converge the algorithm more towards optimal solutions. The following three important elements are required for a GA to be used for a specific type of problem:

(i) **Each individual of the population** is encoded as a string-like representation, often referred to as the *chromosome*. The representation itself plays an important role in the DSE process, as the representation determines the choice of the genetic operators [20].

(ii) **A fitness function** is required to evaluate chromosomes. This evaluation function will determine how each individual of the population performs regarding the objectives. It allows the determination of which design points are good and which are bad. In this project, we use our *Simuflag* simulator (see section 2) to evaluate the individuals.

(iii) **The GA operators** allow the algorithm to both utilize knowledge of good design points (known as exploitation) and to take exploratory steps. The crossover operator will combine the genetic material of two parental individuals (in the form of chromosome parts) to create offspring. The mutation operator will change the offspring slightly to include exploration. A selection operator will determine, based on the fitness, which pair of individuals are selected for the next generation.

Algorithm 3 Basic genetic algorithm

```
1: initialise population
2: repeat                          ▷ Each iteration corresponds to a new generation
3:     crossover
4:     mutation
5:     evaluation
6:     selection
7: until termination condition
```

Algorithm 3 shows the main loop of the algorithm. A GA will start with an initial population, which can be any arbitrary group of design points (chromosomes), but preferably a diverse set of individuals selected from the whole design space. Out of the current population, a new generation will be formed through a crossover function and a mutation operator. The crossover implements a method that mixes chromosomes of the parents into children, which are part of the offspring of this generation. Some of the offspring will then be mutated, which is commonly done by a certain random probability. To allow convergence towards a more optimal solution, the best chromosomes

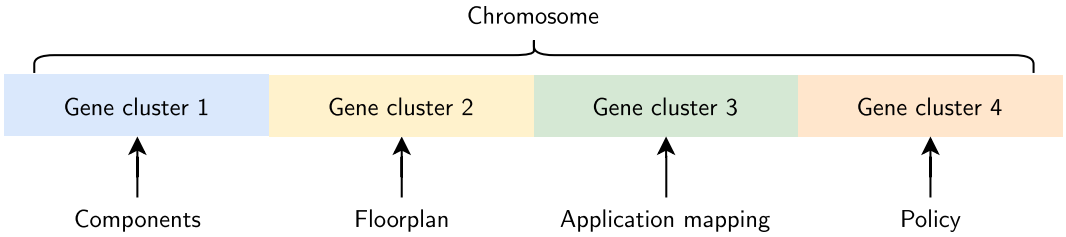


Figure 7: Chromosome cluster representation of a single design point containing the components, their locations, the application mapping and an adaptive policy function.

of the parental population and the offspring are selected and will be used to determine the next generation. This process of crossover, mutation, and selection is repeated a finite number of times until the GA meets certain conditions.

The success of a GA is highly dependant on the choice of parameters and operators. There are the exogenous parameters, which are the global parameters of the GA defining global elements such as the population size and the selection procedure. Endogenous parameters define properties on the level of chromosomes, such as mutation probability. It is beforehand not known how the different parameters will affect the final result of a GA. Thus they have to be evaluated and compared to observe what works best that itself is an optimization problem.

In this work, we implement a GA using the Distributed Evolutionary Algorithms in Python (DEAP) framework [11]. The source code of the GA implemented in this section is available within the earlier mentioned GitHub project.[5]

3.2 Chromosome representation

A single design point has to be able to be represented via a chromosome (or genotype). This chromosome represents a proposed solution to the multi-objective DSE problem. It must be possible for a chromosome to be directly translated to a design point (as described in Section 2.2.1 in the D3.1 report). As a recap, a design point consists of (i) an $m \times n$ grid of computing components, (ii) an application mapping and (iii) an adaptive policy. The information of a single computing component can be separated into the computational capability and its position on the grid. The chromosome representation is separating these two elements, allowing for a distinction between a component's computational capability and its location.

A lot of distinct heterogeneous information has to be captured by a single chromosome. For this reason, the chromosome is split up into four different clusters as shown in Figure 7. Each cluster has a direct relation to the elements that make up a single design point. Dividing the chromosome into these four clusters allows an easy translation (often called a genotype-phenotype mapping [20]) to the actual design point and for more logical crossover and mutation operators. A k-point crossover operator itself does not make much sense in the aspect of heterogeneous genes since a variety in

[5] `https://github.com/sea-art/Simuflage/tree/master/src/DSE`

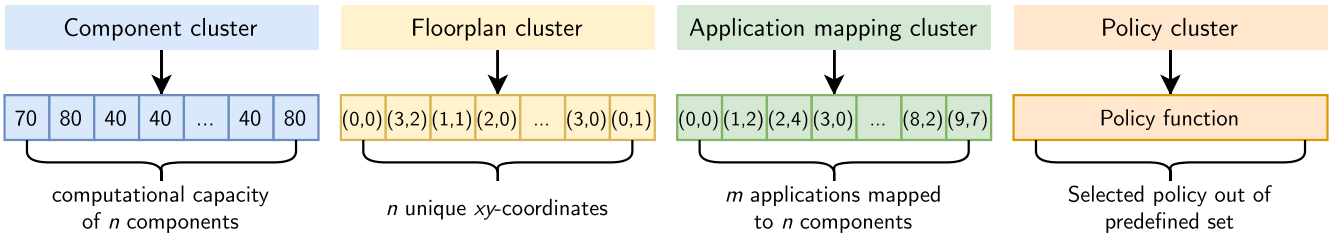


Figure 8: Initial chromosome cluster specifications, illustrating more concretely what a chromosome consist of. The application mapping cluster will in a later subsection be adjusted.

length and information of each gene of the chromosome would not provide correct offspring. The chromosome also has to cope with various constraints, as will be discussed later on in this section.

Several challenges arise by using this cluster chromosome notation. Most of the clusters are in at least one way dependant on each other (with exception of the policy). The floorplan, which is the encoding of the locality of computation resources (gene cluster 2), is directly linked with the components themselves (gene cluster 1). This implies that these clusters are required to be of the same length, but each gene in the floorplan cluster must also be unique. Similar requirements hold for the application mapping since the mapping is dependant on the chosen number of components and their computational capacity. The mapping must be done concerning the capabilities of the components (gene cluster 1). The mutation and crossovers operators have to be able to cope with these dependencies and constraints.

Figure 8 makes the chromosome representation more concrete. The component cluster (previously referred to as gene cluster 1) consists of a set of n numbers where the i-th number will represent the computational capability of the i-th component. Each gene in the floorplan cluster is a unique xy-location represented as a tuple of two values. The i-th gene of the application mapping represents where application i will be mapped to. The genes in the application mapping actual cluster will therefore contain a tuple (a, c) where a is the index of the application that is mapped to the c-th component. While the representation might be shortened to just a single string of values (since the i-th value of the string will correspond with the i-th application), it is less directly corresponding with the actual application mapping of a design point (i.e. this tuple gene cluster notation allows for a more direct genotype-phenotype mapping). However, this application cluster shows to be inherent to several design problems for the crossover and mutate operators. For this reason and as we will explain in subsubsection 3.4.3, we will adjust the application mapping chromosome to a heuristic algorithm. The policy cluster is a selected function out of a predefined set provided by the designer.

3.3 Search space

During the DSE, a designer is often interested in the best selection out of a predefined scope/context. The search space within the GA allows the designer to specify the points of interest that ought to be explored. Often, only a selected set of hardware components should be explored, which have to be specified to the GA. The exploratory bounds of the algorithm also have to be specified, otherwise

Algorithm 4 Crossover cluster division

function CROSSOVER(parent1, parent2)
  $c_1, c_2 \leftarrow \text{CROSSOVER}_{\text{COMPONENT}}(\text{parent1}_{\text{cluster}_1}, \text{parent2}_{\text{cluster}_1})$ ▷ subsubsection 3.4.1
  $f_1, f_2 \leftarrow \text{CROSSOVER}_{\text{FLOORPLAN}}(\text{parent1}_{\text{cluster}_2}, \text{parent2}_{\text{cluster}_2})$ ▷ subsubsection 3.4.2
  $a_1, a_2 \leftarrow \text{CROSSOVER}_{\text{APP\_MAPPING}}(\text{parent1}_{\text{cluster}_3}, \text{parent2}_{\text{cluster}_3})$ ▷ subsubsection 3.4.3
  $p_1, p_2 \leftarrow \text{CROSSOVER}_{\text{POLICY}}(\text{parent1}_{\text{cluster}_4}, \text{parent2}_{\text{cluster}_4})$ ▷ subsubsection 3.4.4
  $\text{child}_1 \leftarrow \text{NEW\_CHILD}(c_1, f_1, a_1, p_1)$
  $\text{child}_2 \leftarrow \text{NEW\_CHILD}(c_2, f_2, a_2, p_2)$
  return child_1, child_2
end function

deriving design points with an infinite number of components would be possible. Not specifying the bounds also results in a design space that is too large to be explored. When components are placed on a grid with unbounded size, neither the exploration of all possible component locations is feasible nor the expectation that the algorithm will converge to an optimal solution. Defining the bounds within the search space will limit the number of design point candidates. Thus, the search space consists of:

(i) a list of predefined computational components,

(ii) the maximum grid size where components can be positioned,

(iii) a limit on the total number of components to select

(iv) a fixed set of applications that each design point has to run,

(v) a set of predefined policy functions. Since there is no genetic programming involved in this project, the policy functions are not constructed during the exploration phase. The GA will only explore the given policies out of a predefined set of options.

In general, the search space will provide a boundary for the GA to indicate what has to be explored and to keep the explored design instances feasible. The specified search space has significant influence on the genetic operators to only allow chromosome alterations within the specified bounds.

3.4 Crossover

When two individuals have been selected by the GA to create offspring, it will do so using the crossover operator. To determine which candidates will generate offspring, the algorithm allows two individuals to mate through a defined crossover probability. The population will be split into two halves, each individual of one half will be paired with an individual of the other half. For each pair, the probability will determine if offspring is created by this pair. Each pair of parents that will create offspring will always create two children.

While normal crossover operators might utilize a more straightforward k-point operator on the entire chromosome, the chromosomes of this GA consist of clusters, requiring the crossover operator to tackle this slightly differently. For each of the clusters (i.e. components, floorplan, application mapping, and policy), a crossover function has to be defined. Each of these crossover functions will provide two children gene clusters, which are all combined to create the two children as illustrated in Algorithm 4. In the following subsections, the crossovers for each individual gene cluster will be defined.

3.4.1 Component cluster

The crossover operator for the component cluster is realized via a more traditional and popular crossover method called the k-point crossover. This type of crossover is most useful when dealing with array or string chromosomes, making it very applicable for this component gene cluster. For the two parents, k random unique indices will be selected from their gene representation. From each of these points up to the next point, the genes from one parent will be selected for child_1 and the genes from the other parent will be used for child_2. Figure 9 illustrates this method of crossover for both the single-point crossover and the two-point crossover, which can be generalized to a k-point crossover for any k that is smaller than the size of the gene cluster.

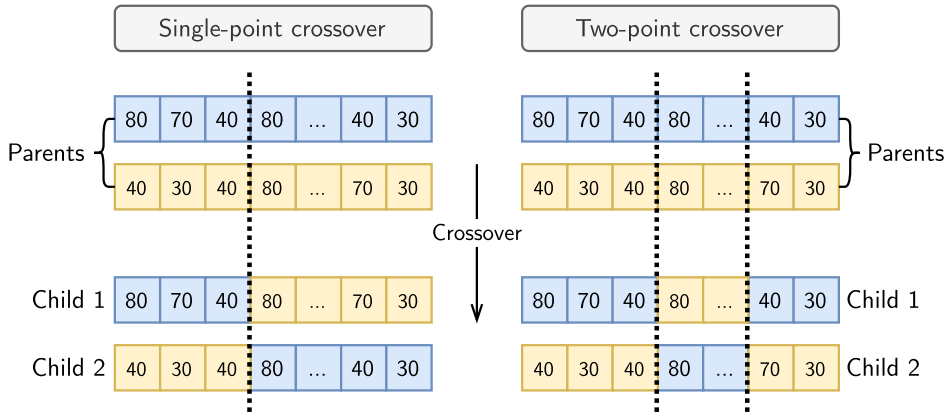


Figure 9: Component gene cluster crossover is done either via single-point, two-point or k-point crossover, where the swap-over points are selected at random.

This crossover method will utilize the gene clusters of both parents equally in order to create offspring without any loss of information from the parents. It often occurs that two individuals that are selected for mating (i.e. crossover) do not have an equal number of components thus differ in array length. For the component cluster, this does not introduce any errors, since child_1 will have the same length as parent_1 while child_2 will have the same length as parent_2. No repair method is required for the component cluster itself.

3.4.2 Floorplan cluster

Defining a crossover method for the floorplan cluster is more difficult due to the number of constraints that it has. There are two main constraints that have to be tackled

(i) all the individual genes within this cluster have to be unique,

(ii) the length of the children from this crossover should be of equal length as the component crossover operator.

The first constraint prevents the utilization of a k-point crossover operator since it is not preventable that two genes after the crossover will remain unique. When two components from different parents are positioned on the same location, it most likely will be represented within the cluster on a different index.

For this reason, a different crossover operator is utilized, which will be referred to as a k-element unique swap-over. Remaining true to the uniqueness constraint plays a central role in this crossover method. For parent_1, all the unique locations in parent_2 that are not in parent_1 will be gathered in a set of locations. These are the only locations that can freely be used for the crossover since they will not introduce any duplicate within the cluster. Given this constructed set of unique locations from the other parent, k genes from the cluster will randomly be swapped over to a location from this set. The same method is used while swapping the parents for the crossover for the other child. This crossover algorithm is illustrated in Figure 10.

There are several edge cases that might occur within this method. It might, for example, be the case that one (or both) parents can not select any unique locations from the other parents, since they are not present. This introduces a more general error in that k elements can not be defined beforehand, since there might be 0 elements to swap over. For this reason, k will not be used as a parameter for this operator (as is the case for the k-point crossover), but k will actually be divided in k_1 and k_2 where these values will uniformly be determined based on the length of the set of unique locations from the other parent. In the case that there are 0 unique elements, there will simply be no components swapped over. Algorithm 5 shows in more detail how the k-element unique swap-over is implemented.

This method does always create two children, where child_1 will have the same length as parent_1 and child_2 the same length as parent_2. Since the children of the component cluster operator are of equal length as the parents and no duplicates are introduced, no repair method is needed to maintain the validity of a floorplan cluster after this operator.

3.4.3 Application mapping cluster

The application mapping gene cluster has a significantly more difficult challenge when designing a crossover between two parents. The children of this crossover have the following constraints:

(i) the applications can only be mapped to n components, where n has to correspond to the offspring from the component and floorplan gene cluster crossovers,

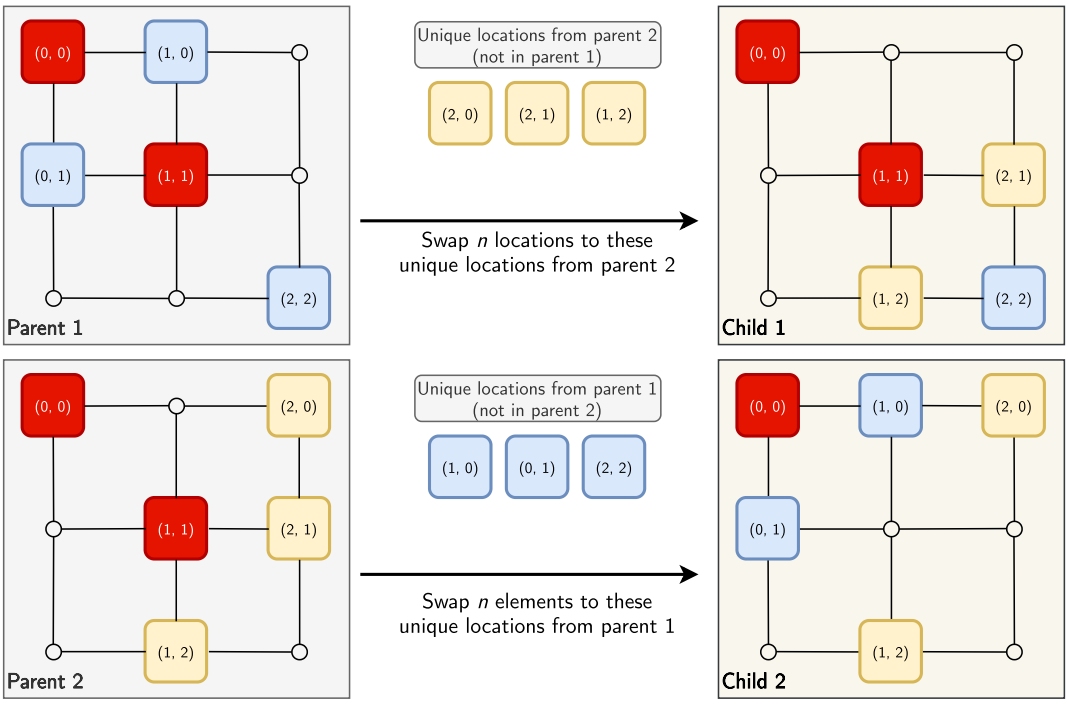


Figure 10: k-element unique swap-over operator used to mate (crossover) two parents on the aspect of their component locality. Parent 1 looks at all uniquely different locations in parent 2 (i.e. locations not present in its own floorplan gene) and will swap k (2 in this example) random components to these unique locations and will do the vice-versa for parent 2. The colours of the components highlight the uniqueness of each component location.

(ii) all applications have to be mapped (i.e. the length of the application mapping cluster should remain fixed),

(iii) an application should only be mapped on a component that can handle the computational needs of the application,

(iv) the sum of all computational needs of applications mapped on a single component should be less than the computational capacity of the component.

The criteria that the mapped applications fit the computational capabilities of a component makes this problem similar to another well-known problem called the bin packing problem, which is a combinatorial NP-hard problem. Since the constraints are of such significance, using more traditional crossover operators that work with arrays will result in too many false individuals. When taking a step back, the question might also arise if it would make sense at all to crossover the actual application mapping itself from two design points. Looking at a more extreme example, let's say that parent_1 has one component, where all applications are mapped to that single component and parent_2 has the same number of components as there are applications and maps a single application to each component. Since the number of components will not change for the component

Algorithm 5 k-element unique swap-over

function SWAPOVER(P_1, P_2) ▷ P_i is the (copied) list of locations from parent$_i$
  $L_1 = P_2 \setminus P_1$ ▷ unique locations from parent$_2$
  $L_2 = P_1 \setminus P_2$ ▷ unique locations from parent$_1$
  $k_1 \leftarrow \begin{cases} 0 & \text{if } |L_1| = 0 \\ \text{random integer between 1 and } |L_1| & \text{if } |L_1| > 0 \end{cases}$ ▷ #elements to swap
  $k_2 \leftarrow \begin{cases} 0 & \text{if } |L_2| = 0 \\ \text{random integer between 1 and } |L_2| & \text{if } |L_2| > 0 \end{cases}$ ▷ #elements to swap
  for k$_1$ times **do** ▷ swap k_1 elements from P_1 to L_1
    $i \leftarrow$ random not-earlier selected index of P_1
    $P_1[i] \leftarrow$ random element from L_1
    Remove selected element from L_1
  end for
  for k$_2$ times **do** ▷ swap k_2 elements from P_2 to L_2
    $i \leftarrow$ random not-earlier selected index of P_2
    $P_2[i] \leftarrow$ random element from L_2
    Remove selected element from L_2
  end for
  return P_1, P_2 ▷ altered copies of parent$_1$ and parent$_2$ (i.e. the children).
end function

cluster crossover, using a crossover for these application mappings would not make sense either. Moreover, the information of parent$_2$ cannot be embedded into parent$_1$, since it only has a single possible application mapping (that is all applications are mapped to a single component).

Table 4: Four different heuristic algorithms to provide solutions for the bin packing problem [19] with their worst-case complexity, where n is the number of applications.

Algorithm	*Complexity*
Next-fit	$\mathcal{O}(n)$
First-fit	$\mathcal{O}(n \log n)$
Best-fit	$\mathcal{O}(n \log n)$
Worst-fit	$\mathcal{O}(n \log n)$

For this reason, the application mapping will be handled slightly differently than presented in Figure 8. Rather than the actual mapping of applications to components, the gene cluster will consist of a function that will create such an application mapping. We can utilize the well-known algorithms that provide heuristic solutions for the bin packing problem as shown in Table 4. This will provide a design point with an initial application mapping via the specified algorithm within

this gene. The crossover operator itself is then very similar to the policy cluster. Given two parents, it will assign the application mapping function of parent_1 to child_1 and assigns the function of parent_2 to child_2.

3.4.4 Policy cluster

The policy cluster is similar to the application mapping cluster in that it represents a selection of a predefined set of algorithms to choose from. The set of policy functions has to be defined within the search space (subsection 3.3). Many more of these types of functions can be specified in the search space. The crossover operator only looks at which function is selected out of the predefined set and will do a crossover operator similar to the application mapping gene cluster. Child_1 will receive the same policy as parent_1 and child_2 will have the same policy as parent_2.

3.5 Mutation

The mutation operator is used to maintain diversity when moving towards the next generation. Mutation represents the exploratory part of the algorithm and allows for the introduction of new design points by randomly altering one or more gene clusters of an individual of the offspring. Getting stuck in local optima should also be avoided by the mutation operators. Whether or not an individual of the offspring is mutated is based on the predefined mutation probability.

Since the chromosome is separated into four different clusters, the mutation operators will be divided into four different mutation operators, one for each cluster. When an individual is mutated, it is randomly selected which clusters will be mutated. This is realized by first sampling a random integer k between 1 and 4 randomly, and then randomly select k clusters that will be mutated.

3.5.1 Component cluster

Mutating the component cluster has a lot of influence on the performance and energy consumption of a design point. The component gene cluster can be mutated in two different ways

(i) altering the computational capabilities of a random component or

(ii) adding or removing a random component.

The latter allows for the exploration of different numbers of components, while the former the type of components. The combination of these two is crucial for finding optimal design points. One challenge that the component cluster mutation operator has to cope with is to remain within the specified search space. When a budget has been set to avoid an infinite addition of components, the operator has to remain within the specified bounds. It might also be the case that the mutate operator will try to remove a component from a design point with only a single component available, which creates an invalid design point. All these problems can be handled by introducing the so-called *death penalty* (i.e. removing invalid individuals). This has a more strict way of making sure to stay within the bounds of the defined search space.

It is uniformly determined which of the two different mutation ways is chosen when an individual's component cluster is selected to be mutated. Since mutations should not incorporate a too large exploratory step, it is not possible that an individual is mutated to receive an extra component besides altering one of its capacities.

3.5.2 Floorplan cluster

The floorplan cluster has to cope with the fact that the mutated individual should remain true to the uniqueness requirement and that the individual remains within the defined floorplan boundaries specified in the search space. However, the implementation of a mutate operator is quite straightforward. The location of a single random component is selected and this location is altered to a different, not already occupied location within the possible grid.

3.5.3 Application mapping and policy clusters

Since the application mapping cluster embeds only a selection of a predefined set of functions, it actually follows the same mutation operator as the policy cluster. The mutation operator simply changes the function to a random not already chosen function specified in the design space. It might be the case for the application mapping cluster that the mutation will result in that the new function might not be able to map the applications to the components, thus creating an invalid design point. When this occurs, the individual is removed (sometimes called the *death penalty*).

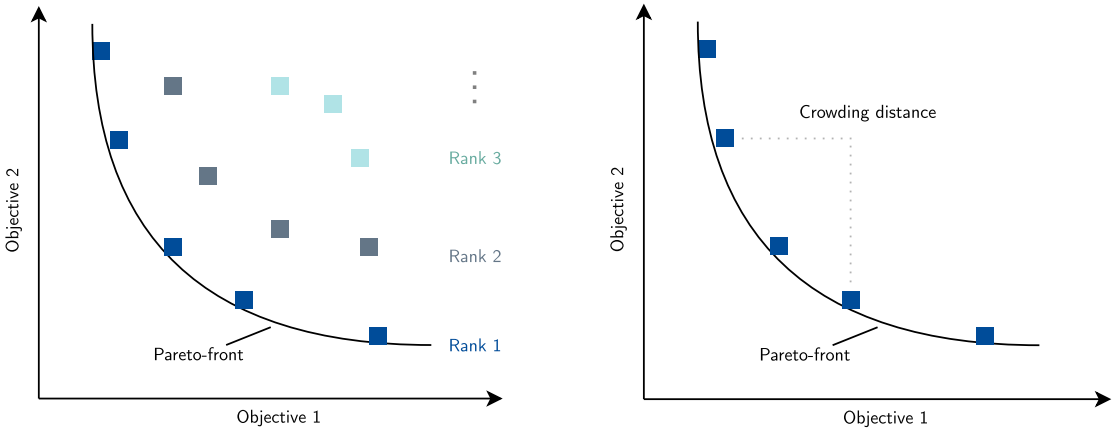
3.6 Selection

Selection is a stage of the GA where n individuals from the complete population (i.e. both parents and offspring) are chosen for the next population. The selection process is determined by the fitness values, that are determined via the evaluation operator. Throughout each generation, the number of individuals remains constant. Several challenges arise during the selection operator. The selection should be performed in a way that diversity is retained within the population, such that the GA will not get stuck in a local optimum. For a multi-objective optimization problem (as is the case for the DSE), individuals can not be ordered in a single list in ascending or descending order based on their fitness. Due to the multiple objectives, individuals can only be compared according to their domination of one and other.

3.6.1 NSGA-II and NSGA-III

Non-dominated sorting is the core concept of the Non-dominated Sorting Genetic Algorithm (NSGA) and its successor NSGA-II. Since NSGA-II is more successful than the original NSGA, this subsection will mainly focus on the successor algorithm. The selection of the algorithm is based on the combination of non-dominated sorting and the crowding distance of individuals as both shown in Figure 11.

Non-dominated sorting will classify each individual to a certain rank based on its Pareto-domination characteristics. Individuals of rank 1 are not dominated by any other individual and



(a) Non-dominated sorting ranks

(b) Crowding distance

Figure 11: Illustrations of the ranking and determination of the crowding distance as presented in [20] with two objectives. Within non-dominated sorting, all solutions are sorted regarding their non-domination rank. The crowding distance is the Manhattan distance between the left and right neighbours.

are the actual Pareto set of the current generation. Rank 2 individuals are those that are not dominated when individuals from Rank 1 are removed from the population. This step can be continued to Rank n, at which all individuals have a rank assigned to them. With this method, it is possible to select the whole population into certain ranks, where a lower rank corresponds with a better performance in the design objectives. This idea is best illustrated in Figure 11a with two objectives.

The crowding distance is a way to determine how diverse an individual is compared to others of the same front (more specifically its direct neighbors). When there is a short distance between an individual and its neighbors, it means that these three individuals are closely positioned together, making them not diverse. Individuals that have a high distance indicate that they are more different from the other points within the same rank. The crowding distance will help with maintaining a diverse population.

The NSGA-II combines these two methods, as can be seen in Algorithm 6. The algorithm gives preference to the individuals of lower ranks and will select the individuals of rank 1 first (which is the Pareto-optimal set) and will keep adding the individuals of each consecutive rank until a rank p will not fit in the next population. Once this is the case, the crowding distance is computed for each individual within rank p. All individuals of rank p are then sorted based on their corresponding crowding distance. The algorithm will add individuals in decreasing order of this front until the population is full (i.e. the same size as the previous population). Note that the outer individuals of a single rank (which are the ones with fewer neighbors than the other individuals) can be seen as diverse points and will therefore have a crowding distance of ∞ assigned, making those the first individuals to be selected out of this rank.

Algorithm 6 NSGA-II [20] selection

```
function NSGA2(P_t, O_t)                          ▷ parent population and offspring
    P_{t+1} ← {}                                  ▷ next population
    fronts ← NON_DOMINATED_SORTING(P_t ∪ O_t)     ▷ sorted fronts (Figure 11a)
    for each F_i in fronts do
        if |F_i ∪ P_{t+1}| < |P_t| then           ▷ if front fits in new population
            P_{t+1} ← P_{t+1} ∪ F_i               ▷ add whole front to new generation
        else                                      ▷ crowding distance selection
            for each objective do
                sort F_i with regard to objective
                set distance of first and last individual to ∞
                for each solution do
                    add crowding distance times weight of objective
                end for
            end for
            Sort F according to descending crowding distance
            Add first (|P_t| − |P_{t+1}|) individuals to P_{t+1}   ▷ add individuals with highest distance
        end if
    end for
    return P_{t+1}
end function
```

Earlier research [10, 18] extended the NSGA-II algorithm to function better as a many-objective optimization algorithm (which are the problems with three up to 15 objectives). The result is an improved algorithm of NSGA-II called NSGA-III. While the core remains similar to NSGA-II, it will improve and alter the part that calculates the crowding distance via the use of reference points. The NSGA-III algorithm within this project creates these reference points uniformly by using random reference points on a hyperplane that will intersect each axis of the objective space at 1. When looking at comparisons within earlier research [9], it is generally observed that NSGA-III will perform better for problems with more than two objectives. However, this is not always the case [17], which shows that NSGA-II can in some cases perform better than NSGA-III in many-objective optimization problems. Both algorithms are provided by the DEAP framework [11], which are the versions that are being utilized in this project.

4 Adaptivity-aware real-time scheduling policies

This work package builds on the results of other work packages, in particular the design space exploration (DSE, Task 3.2, see Section 3), and the domain-specific language (DSL, Task 1.1). The DSL provides directed acyclic graphs (DAGs) representing the tasks and data dependencies of the application, while the DSE specifies characteristics of the utilized hardware architecture.

Scheduling involves (i) assigning graph nodes to processing elements (PEs) and (ii) determining the order in which the nodes are executed on the respective PE. Our approach aims at bridging the gap between fault tolerance and analyzability. On one hand, the system should be able to dynamically react to faults, but on the other hand it should be possible to estimate the worst case execution time (WCET) of graph executions with high precision. Therefore, we currently do not compute strict timings, only a task assignment and execution order so that the actual point in time a task is executed at runtime depends on the availability of data. The task assignment and execution order can be used in the static analysis of a graph execution while the data-driven execution approach allows to temporarily insert additional tasks into the schedule at runtime whenever a re-execution is required.

4.1 Scheduling prerequisites

In the following, we provide an overview of properties required for the scheduling of task graphs and describe how these properties are acquired:

(i) **Available processing elements**: An important aspect for scheduling is the set of available PEs which are considered in the scheduling process. This set may contain homogeneous or heterogeneous PEs. Furthermore, it might be useful to limit the set of considered PEs, for example if another program is supposed to run on the hardware architecture or if some PEs should serve as spare components. Since the number and characteristics of available PEs is related to the hardware architecture, we obtain this information from the DSE.

(ii) **Runtime estimation of tasks**: Another necessary property for the scheduling are runtime estimations for all tasks in the graph. We determine estimations for combinations of tasks and PE types by performing various measurements.

(iii) **Data transfer rates**: Depending on the hardware architecture, it might be important for scheduling algorithms to consider data transfer times between different PEs in order to compute reasonable schedules. With regards to how PEs are connected and which memories they share, there could be a single, constant bandwidth that applies to all data transfers, or different bandwidths for pairs of PEs. Other hardware-dependent properties, which might be required for the scheduling, are transfer initialization and termination times, for example. We can determine this information (whenever it is relevant) from the hardware architecture provided by the DSE.

(iv) **Data passed between tasks**: Data transfer rates alone are not sufficient to estimate transfer times. It is also important to have knowledge about the size of the data transferred from one PE to another. This information is available in the graphs provided by the DSL.

(v) **Memory constraints**: Lastly, depending on the hardware architecture given by the DSE, there might be memory constraints that must be considered in the scheduling process. An example is a distributed architecture where some PEs only have access to small local memories. In such cases, scheduling has to ensure that tasks are assigned to PEs with enough memory

to store the input, output and temporarily created data. Purely shared-memory architectures do not require such a constraint.

4.2 Support for different redundancy configurations

In this section, we describe scheduling under different redundancy configurations which allow to detect errors in task executions. Since the ADMORPH project targets dynamically evolving systems that might need to change redundancy at runtime, we consider five different configurations in the scheduling process:

(i) **Non-redundant** task execution

(ii) **Double** task execution **on different PEs** with comparison tasks

(iii) **Triple** task execution **on different PEs** with voting tasks

(iv) **Double** task execution **on the same PE** with comparison tasks

(v) **Triple** task execution **on the same PE** with voting tasks

All redundant configurations allow to detect transient errors in task executions. Depending on whether tasks are executed two or three times, such errors can be corrected by immediately re-executing tasks or they are automatically resolved via voting. If configurations with redundancy over different PEs are used, the system is also able to determine whether a PE produces consistently wrong results. Treating such a permanent fault is future work, see Section 4.4.

Because of the varying number of task executions in the redundancy configurations, one single schedule is not sufficient. First, we want to focus on the redundancy configurations that involve task executions on different PEs. To compute schedules with redundancy over multiple PEs, existing scheduling algorithms must be modified. Otherwise, redundant tasks are potentially assigned to the same PE. We chose to extend the heterogeneous earliest finish time (HEFT) scheduling heuristic [31] since it is suitable for a large variety of hardware architectures. Algorithm 7 shows the modified heuristic for the case of two redundant task executions. The general principle behind this extension also works for three redundant executions per task. By default, HEFT iterates over all graph nodes and considers all PEs as possible candidates for task assignments. In order to generate suitable redundant schedules, it is sufficient to check for each task whether one of its associated redundant tasks is already assigned to the currently considered PE so that this PE can be temporarily excluded from the scheduling process. Line 12 in Algorithm 7 contains the corresponding check.

Furthermore, we applied two additional modifications to the heuristic. First, the modified HEFT algorithm assigns each comparison task directly after one of the associated redundant task so that errors are detected as early as possible. To realize this, our version of HEFT always schedules two redundant tasks and their associated comparison tasks successively (Line 10). Thus, redundant tasks and comparison tasks can be skipped when the scheduling list is created. Despite this modification, tasks are still scheduled in a topological order so that correct schedules are

Algorithm 7 HEFT Scheduling with Redundant Executions

```
1:  procedure HEFT_REDUNDANT(s : graph_section)
2:      compute mean values for all node and edge weights in s
3:      compute the upward rank for all nodes in s
4:      create scheduling list as in standard HEFT ignoring redundant and comparison nodes
5:      while the scheduling list is not empty do
6:          let n be the first node in the list
7:          remove n from the list
8:          let n' be the redundant node belonging to n
9:          let n_c be the comparison node belonging to n and n'
10:         for x in [n, n', n_c] do
11:             for each PE p do
12:                 if x is n' and n was assigned to p then
13:                     proceed with next PE
14:                 else if x is n_c andc both n and n' were not assigned to p then
15:                     proceed with next PE
16:                 end if
17:                 compute the earliest finish time (EFT) of x on PE p
18:             end for
19:             assign x to the PE with the lowest earliest finish time
20:         end for
21:     end while
22: end procedure
```

produced. The desired assignment is then ensured by another conditional block inside the inner loop (Line 14).

The second additional modification consists of a slightly different EFT computation. By default, the EFT is computed with the following formula (see [31]):

$$EFT(n_i, p_j) = w_{i,j} + \max\left\{avail[j], \max_{n_m \in pred(n_i)}(AFT(n_m) + c_{m,i})\right\}.$$

This formula only considers direct predecessors for communication costs ($c_{m,i}$). However, in redundant graphs the direct predecessors of non-comparison tasks are comparison tasks. In this case, it makes sense to also consider the preceding tasks of the comparison tasks since the corresponding PEs performed the actual computation and have the data in their local memory or cache.

For redundant executions on the same PE, it is possible to use standard DAG scheduling heuristics without modification. Redundant tasks and the following comparison or voting task can be executed without interruption by other task executions or transfers and can thus be considered as one longer execution in the scheduling process. In contrast to redundant execution over different PEs, these redundancy configurations have the advantage that on distributed hardware architectures

fewer data transfers are required. Since schedules with redundancy on the same PE are generated with standard heuristics, it is possible to use non-redundant schedules for the respective redundancy configurations. The downside of this approach is that the non-redundant schedule is possibly less suited for a redundant execution than a separately computed schedule because of the difference in the ratio of transfer to computation.

4.3 Changing the redundancy of parts of an application

In order to enable the software to change the redundancy configuration of parts of an application at runtime, we divide graphs into different sections which are scheduled independently. These sections are executed one after another with barriers in between. The barriers serve as checkpoints to switch to a different redundancy configuration, i.e. to a different schedule. It is important to choose the section boundaries and the order in which sections are executed in a way that the graph remains executable. If a graph consists of multiple sections, it is possible to check whether the sectioning is correct by looking at the dependencies between sections. There is a dependency between two sections a and b (with $a \neq b$) if and only if there is a data dependency between a task from a and a task from b. A correctly sectioned graph must not contain any cyclic dependencies between sections, i.e. the dependencies between sections must also form a DAG. Otherwise, it would not be possible to specify an order in which sections are executed. We call the DAG consisting of sections as its nodes and section dependencies as its edges *Meta-DAG*. After a correct Meta-DAG has been identified, the execution order of sections (given by their indices) can be checked. To be correct, the order of sections must follow the dependencies between sections, i.e. it must represent a topological sort of the Meta-DAG.

Graph sections are always DAGs on their own. Therefore, the scheduling techniques described in the previous section can be applied to individual graph sections without any modifications. Scheduling sections independently clearly has an impact on the performance of graph executions. This has two reasons. First, the more sections a graph is divided into, the less information about the overall graph structure scheduling heuristics can use. Second, sections prevent scheduling algorithms to optimize the execution order of tasks over section boundaries. These two factors lead to a high probability that the execution of a graph with many small sections runs slower than the execution of the same graph with fewer but larger sections.

4.4 Ongoing and future work

As described above, the choice of graph sections has an influence on the makespan of schedules. Hence, there is a trade-off between the granularity of dynamic fault tolerance and the overall performance of graph executions. At the moment, graph sections must be specified manually, but we also want to explore methods to determine suitable section boundaries automatically. However, an entirely automatic sectioning would greatly reduce flexibility. Thus, the method should allow to specify the redundancy requirements of an application so that the sectioning routine can determine the number of sections and section bounds accordingly. A bachelor student is currently working on automatic sectioning as the main part of the thesis.

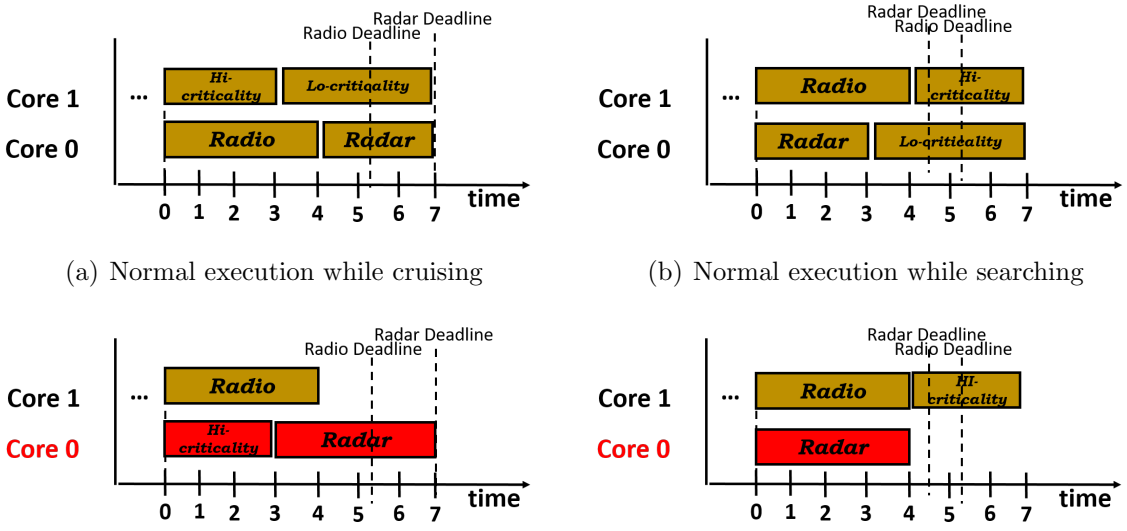
Another ongoing task is the consideration of a graph execution deadline in the scheduling and analysis process. The extended HEFT scheduling heuristic described above computes multiple schedules for each graph section to support different redundancy configurations. Our goal is to determine combinations of graph sections that can be executed redundantly, and how many faults may occur so that the deadline is still met. This is the topic of another bachelor's thesis, which is currently being worked on.

The third aspect we want to highlight is the consideration of task migrations due to permanent faults. In many cases, it is clear when a PE is permanently damaged, for example when the PE does not respond anymore. However, permanent faults might be more subtle, for example when they only cause consistently wrong results and thus there might be consecutive re-executions. As mentioned above, considering the consequences of permanent faults in the scheduling and analysis is future work.

5 Models of computation and derived architectures to allow seamless reconfiguration

This section relates to the activities and results of Task 3.4, i.e. introducing models to formally describe adaptivity at the application, architecture and system levels. Adaptability is a key attribute that enables embedded computer systems to be highly performant and at the same time react to internal or external events, such as crossing the boundary of operational envelope due to a change of the environment, or a software/hardware failure, e.g. due to dormant bugs/material fatigue respectively. In the literature of the real-time systems, there is a plethora of models that address a category of such events, while expressing the real-time aspects of the systems. For example, mixed-critical systems model the difference of worst-case execution time for various levels of assurance, whereas mission-based systems reflect the change of priorities with respect to a given mission. Nevertheless, real-life systems typically face more events from multiple planes, e.g. functional, timing, operational environment, security, etc. For example consider an embedded computer of a search-and-rescue civil aircraft that contains the functionality for the radar and the radio communication along with two other tasks, one of high-criticality and of low-criticality functionality. An example of a schedule, when the aircraft is cruising, that meets the timing requirements (called deadlines) on a dual-core architecture is illustrated in Figure 12(a). 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 there are different timing requirements for the two different phases of the mission and possibly different frequencies/periods for the radar functionality. 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 12(b), along with one out of many valid schedules. In addition, when a core overheats, the operating frequency of that core can be reduced, to avoid permanent damage, resulting in potentially longer executions. As a result, there are different execution demands for the two different hardware states, leading to the low-criticality task to be dropped, in both cruise mode (Figure 12(c)) and search mode (Figure-12(d)) as in all possible schedules at least one task would violate its deadlines, if all the

tasks where tries to execute, thus the lo-criticality is removed.



(a) Normal execution while cruising

(b) Normal execution while searching

(c) Abnormal execution while cruising

(d) Abnormal execution while searching

Figure 12: (Motivational example) Execution of the cruise/search modes in normal/abnormal timing conditions

The aim of Task T3.4 is to propose a set of features for models of computation and their derived constraints for hardware/software architectures (targeting Multi-core, FPGA and distributed) to allow seamless reconfiguration to achieve dynamic adaptation. This task is linked to the architecture definition and physical constraints introduced in Work Package 5 to implement those constraints specified by the coordination language. The model should include the possibility of task graphs to be updated (including reconfiguration tasks, on- demand redundancies and communications re-mapping) and for certain levels of optimization to be performed at run time. Hence, The purpose is not provide algorithms that solve the various optimisation problems, but models that can formally describe adaptation requirements, upon which models optimisation algorithms can be devised.

To provide real-time guarantees for an application, deployed on an architecture, a formal model for the application is necessary. An application consists of a set of computation tasks. There is one significant distinction, encountered especially in the Scheduling Theory community, for the set of computation tasks. Independent tasks are those tasks which, if ready, can be scheduled in any order, while for dependent tasks there is an inherent scheduling order imposed. For example, compressing an image according to the JPEG standard requires that the image is split in blocks before it is transformed using the Discrete Cosine Transformation (DCT) method. So, from a task perspective, there is a data-dependency between the tasks that form the blocks and the DCT transformation tasks. When all of these computation tasks are executed, the application is considered to have completed an execution, that is, the input data have been consumed and the desired output has been produced. The output is considered to be correct if during the execution there are no data-races, deadlocks etc.

This report considers applications that are composed of tasks that are executed iteratively, potentially with different periods. An application composed of independent tasks are typically modeled with task models.

5.1 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*

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

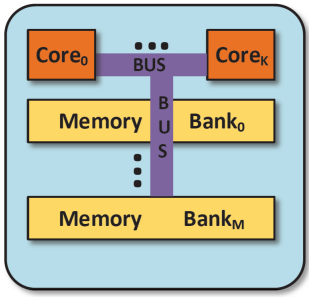
(iv) D *the set of deadlines of each task*

5.2 Computing architecture model

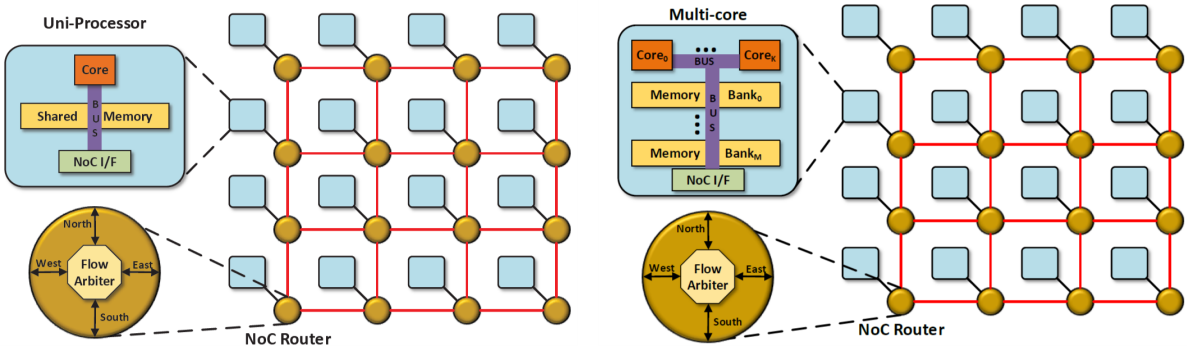
At the time this report was written, several research and commercial computing platforms with multiple Processing Elements (PE) were available. These broadly can be categorised according to i) the type of PEs, ii) the on-chip memory organisation and iii) the network, in case of multiple embedded devices as illustrated in Figure 13(a).

Homogeneous (as opposed to heterogeneous) are the platforms where the 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 the shared memory is organised, there are three main categories, i.e. centralised, distributed and mixed. In 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 the time does 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 NoC, to access various 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.

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.



(a) Multi-core architecture



(b) Distributed architecture

(c) Mixed architecture

Figure 13: Possible architectures

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, i.e. (i) the centralised architectures $MA = (\{\{k_1, \ldots, k_N\}\}, \emptyset, \mathcal{M}, \emptyset)$ which do not have any network (Figure 13(a)), (ii) the distributed architectures $DA = (\{\{k_1\}, \ldots, \{k_N\}\}, \mathcal{L}, \mathcal{M}, \mathcal{N})$ with one core k per cluster (Figure 13(b)), and (iii) the mixed architectures $MX = (\mathcal{C}, \mathcal{L}, \mathcal{M}, \mathcal{N})$ (Figure 13(c)). For the architectures that have a network it is assumed that each cluster has one network interface, with multiple channels.

5.3 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, at any point in time. 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 pre-computed at design time, or applying a schedulability test at runtime. In a similar manner, a change in the hardware capabilities, e.g. a failed core, can be reflected by a transition from an architecture model $\mathcal{GA}$ to the new architecture model $\mathcal{GA}$'.

Utilizing the pair of task model and architecture model to describe the state, to which the system has to arrive, can effectively describe all the required adaptations, including task graphs to be updated with reconfiguration tasks, on-demand redundancies, communications 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 and/or runtime. For the sake of coherence, we shall report more detailed examples in Deliverable D5.2, alongside with the use-cases, where it is more relevant.

6 Timing analysis for certification in heterogeneous processing platforms

Safety-critical applications need to cope with real-time requirements. In this case, runtime execution, timeliness and predictability become a safety issue. The inclusion of complex processing architectures, their increased level of shared resources, and the complexity of their interconnect infrastructures, makes it very difficult to calculate accurate bounds for the Worst-Case Execution Time. Adding run-time adaptation includes an additional challenge very difficult for certification guarantees.

Task 3.5, that at the time of writing has just started, focuses on analysing the timing behavior of adaptive systems. At the current stage, existing timing analysis tools are being evaluated to assess their applicability with respect to the models produced in Task 3.4. So far, the most promising candidate tool for certification are AADL-based tools, e.g. CHEDDAR, due to its maturity and applicability in the avionics domain.

7 Providing formal guarantees on the adaptation layer

A recent survey on the state of industrial practice in real-time systems showed that a significant fraction of real-time tasks are allowed to miss a finite number of deadlines [2]. The research community spent years defining and analysing models of tasks that can miss deadlines, from soft real-time systems, to tasks with a skip-factor and approximating the deadline miss probability for a given system. One of such models in which tasks may miss deadlines is the weakly-hard task

model [5]. This is the model we adopt for our investigation, as it allows us to determine precisely the characteristics the hardware should guarantee (thanks to morphing and adaptation) to still preserve an acceptable behaviour.

In WP1, we defined the requirements of a real-time system on the basis of deadline misses and what can a control system tolerate. To do so, we used a model called *weakly-hard* task model, which constraints the number of deadline hits and misses that a task can experience.

In particular, weakly-hard tasks behave according to patterns of hit and missed deadlines that are (mainly) window-based. The originally proposed constraint model specifies alternatively (for a window of subsequent jobs): (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 third of these models – often called the (m, K) model – gained attention in the research community, generating results on scheduling algorithms [12], real-time and schedulability analysis [30, 26, 13], verification [15, 3] and runtime monitoring [34] of constraint satisfaction, derivation of task model parameters [35], together with applications to domains like telecommunication [1, 16] and control systems [29, 25]. The fourth model has also proved relevant to perform a conservative analysis of the stability of control systems [23]. Furthermore, the relation between weakly-hard constraints has been partially investigated [32, 34].

This model fits the ADMORPH ecosystem, as reconfiguration should occur when the system enters a critical state, in which deadlines cannot be missed any more, for example because this may interfere with the correct operation of a control system. Given the requirements found in WP1 for the specific case studies, we can then analyse the real-time component of the system and the guarantees that the adaptation layer must provide.

Despite the research effort, many questions remain unanswered, both in terms of the relation between different types of weakly-hard constraints, and regarding the type of analysis that can be performed using weakly-hard tasks. We now present some of these questions.

(i) The first question we would like to answer is what happens if a real-time analysis technique, like the one presented in [13], tells us that a task satisfies two constraints simultaneously. In some cases, one of the two constraints *dominates* the other, meaning that satisfying the dominant constraint also guarantees the satisfaction of the dominated one. But this is not always the case. For example, suppose that a task satisfies simultaneously two constraints: $\lambda_1 = (2, 5)$ and $\lambda_2 = (3, 7)$. λ_1 specifies that the task may miss a maximum of 2 deadlines in every window of 5 consecutive jobs. λ_2 says that the same task may miss a maximum of 3 deadlines in every window of 7 consecutive jobs. Neither of the constraints dominates the other and a (generic) sequence of job activations may satisfy λ_1, λ_2, both, or neither. Denoting a deadline miss with a 0 and a deadline hit with a 1, the sequence 0011100 satisfies λ_1 but not λ_2 and the sequence 0001111 satisfies λ_2 but not λ_1. The choice of which constraint is to be used, for example to analyse the performance of a control system following the method presented in [25], is then left to the practitioner, while it would be best to consider *both* constraints simultaneously.

(ii) The second question we raise comes from the consideration that in practice it may be easier

to guarantee that some prescribed job will hit their deadline rather than ensuring that the number of misses follows a given pattern. This is the case of running jobs in a protected execution environment when there is a need to hit the corresponding deadline as opposed to running alongside other tasks. As an example, mixed-criticality allows the scheduler to raise the criticality level and thus guarantee that the highly-critical tasks meet the corresponding deadlines [7]. We can treat the weakly-hard task as highly critical and raise the criticality level when a deadline hit must be enforced. Alternatively, we can increase the budget of a reservation-based scheduler [8]. The consideration here is that often it is easier to guarantee a certain hit pattern rather than a miss pattern. Despite that, the first two types of weakly hard tasks, that constrain the number of hits, have not been receiving much attention from the research community.

(iii) The third consideration we bring forward is one of *scalability*. Many of the research results, for example in the control domain [25, 21, 22], use short windows. However, for practical applications it may be relevant to use a large window size, as done for example in the experimental analysis in [3]. In fact, the original motivation behind the weakly-hard task model [5] uses a practical example from the avionics domain in which a deadline may be missed 11 times in every consecutive 295 jobs. It seems reasonable that systems that are built and certified (for example in the automotive domain) would not experience many deadline misses, and that using a short window size would lead to very conservative results.

To address these questions and empower researchers with a tool to investigate related ones, we are building a software library for the analysis of weakly hard tasks where scalability is treated as a first-class citizen. More precisely, the contribution of the library will be the following:

(i) Leveraging an automata-based representation to describe a task subject to a *set* $\Lambda = \{\lambda_1, \ldots, \lambda_L\}$ of L weakly-hard constraints. Representing weakly-hard constraints with the corresponding directed labeled graph is not a novel contribution. Differently with respect to many research contributions, e.g., [33, 22], rather than addressing a single constraint, we want to construct an automaton that represents the entire set Λ and support all the four weakly-hard constraint types. Furthermore, we construct the *minimal* automaton, paving the way towards addressing the scalability requirement.

(ii) Handling known [4, 32, 35, 34] and *new* dominance relations to speed up the automaton generation. In particular, we want to provide a theoretical contribution on the relation between the first two types of weakly hard constraints, respectively the number of hits and the number of consecutive hits in a window. This relation constitutes the final piece, that allows us to relate all the types of constraints with one another, and provide some ordering among them. Handling constraint domination allows us to further improve the library's scalability.

(iii) Scaling to larger constraints and constraint sets. First, we plan to analyse the scalability of our library compared to the state of the art whenever possible, i.e., for single constraints. Second, we want to look at sets of constraints and perform a sensitivity analysis, to determine

which parameters affect the execution time for the translation of a set of constraints into the corresponding automaton.

7.1 Background and related work

Here we analyse a single real-time task. A real-time task τ is an entity composed of a (possibly infinite) sequence of jobs $(j_i)_{i\in\mathbb{N}^{\geq}}$, representing code that is executed repeatedly on a given hardware platform (not necessarily according to any temporal pattern or periodicity). A task is characterised by its relative deadline d, representing the time after which each job should be completed. The index i counts the job number. For a given job j_i, we denote with a_i its release time (the time in which the job becomes active in the hardware platform), and with f_i its completion time (the time in which the job terminates its execution). We also use d_i to represent the absolute deadline of the i-th job, meaning that $d_i = a_i + d$.

In general, a job can either complete its execution before its deadline or overrun it, resulting respectively in a deadline *hit* or *miss* (collectively denoted by the job's *outcome*).

Definition 3 (Deadline Hit)**.** *The i-th job of a task τ is said to hit its deadline if $f_i \leq d_i$.*

Definition 4 (Deadline Miss)**.** *The i-th job of a task τ is said to miss its deadline if $f_i > d_i$.*

The weakly-hard task model [5, 4] provides guarantees on the sequence of outcomes of a real-time task via four constraints, each specifying how deadline misses and hits are interleaved for a window of $k \geq 1$ consecutive jobs.

Definition 5 (Weakly-Hard Task)**.** *A weakly-hard task τ is a task that satisfies (at least) one of the following constraints:*

(i) $\tau \vdash \binom{x}{k}$ *(`AnyHit`): in any window of k consecutive jobs, the minimum number of hits is x;*

(ii) $\tau \vdash \left\langle {x \atop k} \right\rangle$ *(`RowHit`): in any window of k consecutive jobs, the minimum number of consecutive hits is x;*

(iii) $\tau \vdash \overline{\binom{x}{k}}$ *(`AnyMiss`): in any window of k consecutive jobs, the maximum number of misses is x; and*

(iv) $\tau \vdash \overline{\left\langle {x \atop k} \right\rangle}$ *(`RowMiss`): in any window of k consecutive jobs, the maximum number of consecutive misses is x;*

for some values of $x, k \in \mathbb{N}^{\geq}$, where $x \leq k$ and $k \geq 1$. We use the $\vdash$ symbol to indicate that all the possible sequences of outcomes of τ satisfy the right hand side.

All the types of constraints in Definition 5 have received attention in the real-time systems literature. In particular, the `AnyMiss` constraint has been extensively studied, and is commonly addressed as the (m, K) weakly-hard task model. However, each of these constraints has been studied separately and a task can simultaneously satisfy many of them, possibly of different types.

Exploiting different types of constraints – and possibly different parameters for the same type of constraint – leads to a better outcome for the analysis of the system. This follows from the space of possible sequences being pruned, thus allowing us to focus on proving that the real-time system behaves correctly in the relevant cases. In the following, we denote a set of L weakly-hard constraints with $\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_L\}$. To characterise the possible sequences of outcomes that satisfy a constraint, we borrow some elementary concepts from language theory, in particular the *binary alphabet* [14].

Definition 6 (Alphabet Σ of Job Outcomes)**.** *We define the alphabet of job outcomes* $\Sigma = \{0, 1\}$*, where* 0 *indicates a deadline miss and* 1 *represents a deadline hit.*

Using well-established notation, we denote the *character* $c_i \in \Sigma$ as the outcome of job j_i. A *word* w of length $|w| = N$ is a sequence of characters $w = \{c_1, c_2, \ldots, c_N\}$ that specifies a sequence of consecutive job outcomes for a task. Without loss of generality, we assume that all words are preceded and followed only by hits. We denote the subword of a word w from index a to b with $w(a, b) = \{c_a, c_{a+1}, \ldots, c_b\}$. Finally, Σ^N denotes the set of all possible words of length N.

With a slight abuse of notation, we use $w \vdash \lambda$ to indicate that the word w satisfies the constraint λ. Obtaining the set of words satisfying λ follows directly from the definitions of the alphabet and the constraint itself [5, 4].

Definition 7 (Satisfaction Set $\mathcal{S}_N(\lambda)$)**.** *The set of all length* N *words* w*, satisfying the weakly-hard constraint* λ*, is denoted by* $\mathcal{S}_N(\lambda)$*. Formally,* $\mathcal{S}_N(\lambda) = \left\{w \in \Sigma^N \mid w \vdash \lambda\right\}$*,* $N \geq 1$*.*

Trivially, all words in $\mathcal{S}_M(\lambda)$ are subwords of words existing in $\mathcal{S}_N(\lambda)$, if $M \leq N$. To simplify notation we define the set containing all words of infinite length as $\mathcal{S}(\lambda) \equiv \mathcal{S}_\infty(\lambda)$.

Using satisfaction sets, it is possible to formally define a partial ordering between two constraints λ_i and λ_j. We denote the logical conjunction with $\wedge$ and the logical disjunction with $\vee$. The following notions of constraint *domination* and *equivalence* [5, 4] are used extensively throughout the remainder of the paper (jointly denoted *constraint dominance*).

Definition 8 (Constraint Domination)**.** *Given two arbitrary weakly-hard constraints* λ_i *and* λ_j*,* λ_i *dominates* λ_j *(denoted* $\lambda_i \prec \lambda_j$*) if all words satisfying* λ_i *also satisfy* λ_j*, i.e.,* $\mathcal{S}(\lambda_i) \subset \mathcal{S}(\lambda_j)$*. Correspondingly,* $\lambda_i \preceq \lambda_j \Leftrightarrow \mathcal{S}(\lambda_i) \subseteq \mathcal{S}(\lambda_j)$*.*

Definition 9 (Constraint Equivalence)**.** *Given two arbitrary weakly-hard constraints* λ_i *and* λ_j*,* λ_i *is equivalent to* λ_j *if they respectively dominate each other. Formally,* $\lambda_i \equiv \lambda_j \Leftrightarrow \lambda_i \preceq \lambda_j \wedge \lambda_j \preceq \lambda_j$*. Two constraints are equivalent if they share the same satisfaction set, i.e.,* $\lambda_i \equiv \lambda_j \Leftrightarrow \mathcal{S}(\lambda_i) = \mathcal{S}(\lambda_j)$*.*

The notion of constraint dominance has attracted attention from different areas, and is still occasionally researched [34, 32]. To provide dominance results, we first define the *weakest* and *hardest* constraints [5, 4].

Definition 10 (Weakest Constraint $\underline{\lambda}$)**.** *The weakest constraint* $\underline{\lambda}$ *is defined as the constraint satisfied by* any *word. Formally,* $\mathcal{S}_N(\underline{\lambda}) = \Sigma^N$*,* $\forall N \in \mathbb{N}^{>}$*.*

Definition 11 (Hardest Constraint $\overline{\lambda}$)**.** *The hardest constraint $\overline{\lambda}$ is defined as the constraint satisfied* solely *by the word containing all deadline hits. Formally, $\mathcal{S}_N\left(\overline{\lambda}\right) = \left\{1^N\right\}, \forall N \in \mathbb{N}^{>}$.*

Using these definitions, we now review known constraint dominance relations. We refer the reader to [4] or any referenced paper for the corresponding proofs.

Lemma 1 (Known Equivalence Relations)**.** *The following equivalence relations hold:*

(i) $\binom{x}{k} \equiv \overline{\binom{k-x}{k}}$*, an* **AnyHit** *constraint with x deadline hits in a window of k jobs is equivalent to an* **AnyMiss** *constraint with $k-x$ hits in a window of k jobs,*

(ii) $\overline{\left\langle {x \atop k} \right\rangle} \equiv \overline{\langle x \rangle}$*, $\forall k \geq 1$, a* **RowMiss** *constraint is independent of the window size, i.e., it is equivalent to the same constraint with any k value,*

(iii) $\overline{\binom{x}{x+1}} \equiv \overline{\langle x \rangle}$*, a* **RowMiss** *constraint with x deadline misses is equivalent to an* **AnyMiss** *with x possible misses in a window of $x+1$ jobs [23],*

(iv) $\left\langle {1 \atop k} \right\rangle \equiv \binom{1}{k}$*, (trivially) a* **RowHit** *constraint is equivalent to an* **AnyHit** *when looking at the same window length and a single deadline,*

(v) $\left\langle {x \atop k} \right\rangle \equiv \overline{\lambda} \Leftrightarrow x > {}^{k}/_{2}$*, a* **RowHit** *constraint is equivalent to the hardest constraint when $x > {}^{k}/_{2}$.*

Using these equivalence relations, we can always translate `AnyMiss` and `RowMiss` constraints into a corresponding `AnyHit` constraint. However, there is no clear equivalence between `AnyHit` and `RowHit` constraints (beside the trivial case of a single deadline and the same window length). Finding such a relation is important because it would allow us to treat sets of different types of constraints reducing the analysis to a single type and therefore improving efficiency.

Denoting with $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$ respectively the floor and ceiling operators, we can then define some domination relations.

Lemma 2 (Known Domination Relations)**.** *The following domination relations hold:*

(i) $\binom{x_1}{k_1} \preceq \binom{x_2}{k_2} \Leftrightarrow x_2 \leq \max\{a, b\}$*, where $a = \lfloor \frac{k_2}{k_1} \rfloor x_1$ and $b = k_2 - \lceil \frac{k_2}{k_1} \rceil (k_1 - x_1)$; the* **AnyHit** *constraint with parameters x_1 and k_1 dominates all* **AnyHit** *constraints with parameters x_2 and k_2 if and only if $x_2 \leq \max\{a, b\}$ with a and b defined as above.*

(ii) For any two constraints $\left\langle {x_1 \atop k_1} \right\rangle, \left\langle {x_2 \atop k_2} \right\rangle \not\equiv \overline{\lambda}$*,* $\left\langle {x_1 \atop k_1} \right\rangle \preceq \left\langle {x_2 \atop k_2} \right\rangle \Leftrightarrow \left(k_2 < k_1 \wedge k_2 \leq x_1 - \lceil \frac{k_1 - k_2}{2} \rceil\right) \vee (k_2 \geq k_1 \wedge x_2 \leq x_1)$*; this specifies the domination between two* **RowHit** *constraints depending on their constraint parameters.*

(iii) $\left\langle {x_1 \atop k} \right\rangle \preceq \binom{x_2}{k} \Rightarrow \{x_2 \leq 4x_1 - k - 2,\ x_2 \leq x_1,\ x_2 \geq 0\}$*; for a fixed and equal window k, if a* **RowHit** *constraint with consecutive deadlines hits x_1 dominates an* **AnyHit** *constraint with x_2 deadlines hits, then the indicated relation between the constraint parameters hold.*

(iv) $\overline{\langle x_1 \rangle} \preceq \overline{\langle x_2 \rangle} \Leftrightarrow x_1 \leq x_2$*; a* **RowMiss** *constraint with a lower number of deadline misses dominates a* **RowMiss** *with a higher number of deadline misses.*

(v) $\overline{\binom{x+p}{k+p}} \preceq \overline{\binom{x}{k}}$ *if* $p > 0$*;* ***AnyMiss*** *constraints can be dominated by other* ***AnyMiss*** *constraints when particular relations hold for values of their parameters [32].*

The ability to translate constraints into `AnyHit` equivalents makes Lemma 2(i) very powerful to compare different weakly hard constraints. Finally, Lemma 2(iii) is the only known result that relates the `RowHit` constraints with the other types. However, its applicability is limited to the case in which the two constraints share the same window size. From the presentation of the existing constraint dominance relations, we gather that there is an important piece missing to achieve a comprehensive weakly-hard analysis.

7.2 Theoretical analysis

We provide here the statements of two theorems that we proved during our theoretical analsyis, omitting the proofs, that will be presented in a scientific paper that is currently under review.

Our first theoretical contribution is the proof of a condition regarding the domination of a `RowHit` constraint over a `AnyHit` constraint. The proof is based on restricting the `AnyHit` constraint's minimum number of hits in order to ensure that its satisfaction set includes the one of the `RowHit` constraint.

Theorem 1 (`RowHit`–`AnyHit` Domination)**.** *Let* λ_1 *be a* ***RowHit*** *constraint* $\lambda_1 = \left\langle \binom{x_1}{k_1} \right\rangle \not\equiv \overline{\lambda}$ *and* λ_2 *be an* ***AnyHit*** *constraint* $\lambda_2 = \binom{x_2}{k_2}$*. Then*

$$\lambda_1 \preceq \lambda_2 \Leftrightarrow x_2 \leq \max\left\{x_1\, q,\, k_2 - (1+q)\, z_1\right\},$$

where $z_1 = k_1 - 2\,x_1 + 1$ *and* $q = \left\lfloor {k_2}/{(z_1 + x_1)} \right\rfloor$*.*

The second theoretical contribution is the proof of a condition regarding the domination of an `AnyHit` constraint over a `RowHit` constraint. The proof is based on restricting the `RowHit` constraint's minimum number of consecutive hits in order to ensure that its satisfaction set includes the one of the `AnyHit` constraint.

Theorem 2 (`AnyHit`–`RowHit` Domination)**.** *Let* λ_1 *be an* ***AnyHit*** *constraint* $\lambda_1 = \binom{x_1}{k_1}$ *and* λ_2 *be a* ***RowHit*** *constraint* $\lambda_2 = \left\langle \binom{x_2}{k_2} \right\rangle \not\equiv \overline{\lambda}$*. Then*

$$\lambda_1 \preceq \lambda_2 \Leftrightarrow x_2 \leq \min\left\{\left\lfloor {k_2}/{(z_1+1)} \right\rfloor,\, \left\lceil {x_1}/{z_1} \right\rceil\right\},$$

where $z_1 = k_1 - x_1$*.*

The two theorems above complete the relation graph between the different types of weakly-hard constraints. Now that we have a complete picture, we can start investigating sets Λ of L constraints, $\Lambda = \{\lambda_1, \ldots, \lambda_L\}$.

Finally, we can extend the theory to the case in which τ is subject to an arbitrary set of constraints of the form presented in Definition 5. First, we extend the satisfaction from Definition 7 and obtain

$$\mathcal{S}_N(\Lambda) = \bigcap_{\lambda \in \Lambda} \mathcal{S}_N(\lambda) \tag{1}$$

where $\bigcap$ is the generalised intersection. We use $\tau \vdash \Lambda$ to denote that τ satisfies all the constraints in the set Λ. This implies that each word $w \in \mathcal{S}_N(\Lambda)$ must belong to the satisfaction set of all the constraints in Λ. Trivially, Equation (1) allows to use our extended satisfaction sets in Definitions 7 and 8 to define constraint equivalence and domination for sets of constraints.

Constraint dominance significantly reduces the problem complexity when working with sets of weakly-hard constraints, Λ. If the constraint set supports different types of weakly-hard constraints, it can be beneficial to find an equivalent set of constraints with minimal cardinality.

To minimise the number of constraints in the problem formulation, the constraint dominance is utilised in order to find the minimal cardinality, equivalent subset. Utilising the comprehensive picture the theorems provide, we propose the notion of a *dominant set*, thus simplifying the analysis of weakly-hard systems subject to multiple constraints.

Definition 12 (Minimal Dominant Set)**.** *The minimal dominant set Λ^* of a set of weakly-hard constraints Λ is defined as the smallest cardinality subset of Λ representing an equivalent set of constraints. Formally, $\Lambda^* \subseteq \Lambda$ where*

(i) $\lambda_i, \lambda_j \in \Lambda^* \;\Rightarrow\; \lambda_i \not\equiv \lambda_j,\ \forall i \neq j,$

(ii) $\lambda_i, \lambda_j \in \Lambda^* \;\Rightarrow\; \lambda_i \not\preceq \lambda_j,\ \forall i \neq j,$

(iii) $\lambda_i \in \Lambda \setminus \Lambda^* \;\Rightarrow\; \exists \lambda_j \in \Lambda^*\ s.t.\ \lambda_j \preceq \lambda_i.$

From Definition 8, a weakly-hard constraint λ_i dominates λ_j if and only if $\mathcal{S}(\lambda_i) \subseteq \mathcal{S}(\lambda_j)$. Thus, excluding all the dominated constraints from Λ does not change the resulting satisfaction set. The equivalence between the constraint set and its minimal dominant set is trivial considering the respective satisfaction sets:

$$\mathcal{S}(\Lambda^*) = \bigcap_{\lambda \in \Lambda^*} \mathcal{S}(\lambda) = \bigcap_{\lambda \in \Lambda} \mathcal{S}(\lambda) = \mathcal{S}(\Lambda).$$

7.3 Ongoing and future work

As future activity in the task, we are implementing a software library[6] 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 sequences of arbitrary length that satisfy a set of weakly-hard constraints.

We are building the library having scalability as a first-class citizen, to allow the analysis of relevant large case studies like the ones defined in work package 5.

[6]The library will be distributed as open source software.

8 Automatic validation of safety and security cases for adaptive systems

The PikeOS real-time operating system is intended to be used in safety and security-critical applications with certification needs. The key feature of PikeOS is the capability to safely and securely execute applications with different safety and security assurance levels concurrently on the same platform. Applications can range from simple control loops up to complete para-virtualized and hardware virtualized guest operating systems like Linux. An application with a low assurance level may contain unintended bugs (i.e. safety issues) or malicious code (security issues). In both cases PikeOS ensures that other applications as well as PikeOS itself cannot be compromised, providing a correct domain separation using the PikeOS partitioning concept. Separation of applications is realized by means of spatial and temporal partitioning. A partition is a logical container created and maintained by PikeOS. PikeOS allocates resources to a partition according to configuration data (e.g. memory, CPU time, I/O access rights). A partition can host one or more applications sharing the partition's resources. The part of the PikeOS operating system that enforces partitioning at run-time is called the PikeOS hypervisor. Additional components like device drivers, communication protocol stacks or file systems may be attached to the hypervisor but they are not part of the hypervisor, however, they may be able to compromise the hypervisor and therefore need an adequate design assurance level. Partitioning ensures that faults of applications will only have partition local impact, if the partition does not have specific privileges which break partitioning. Software components which have access to hypervisor or hardware features which may break partitioning must be developed with a design assurance level which is appropriate for the use case. These components are called trusted components; a fault in a trusted component may have module global impact.

Along with the PikeOS operating system components and and Integrated Development Environment (IDE), SYSGO provides a "Safety and Security Manual" that specifies the safety and security requirements (also called exported constraints or usage domain definition) and guidance for building a safe and secure system using PikeOS operating system.

Currently, the safety and security requirements of PikeOS exported resources are specified in natural language in the PikeOS safety and security manual. The system integrator and developer when using the exported resources, make manual validation of system requirements against the safety/security manual. This process is labour intensive, prone to human errors and is not repeatable.

In this task, we investigate methods to automate the validation of system configurations such that they satisfy the safety and security requirements. This automated process allows to find valid configurations during the dynamic reconfiguration process.

The idea is to identify candidate tools that could be used for automation of safety / security cases. In the following we report the first assessment we did on Eclipse Modeling Framework (EMF) for describing PikeOS system configuration and using it for validating an important security property - information flow policy between partitions based on the resource allocated.

Generally to construct a safety and/or security case that can be automatically verified one can go into two directions. One direction is ex-post verification of the safety / security case

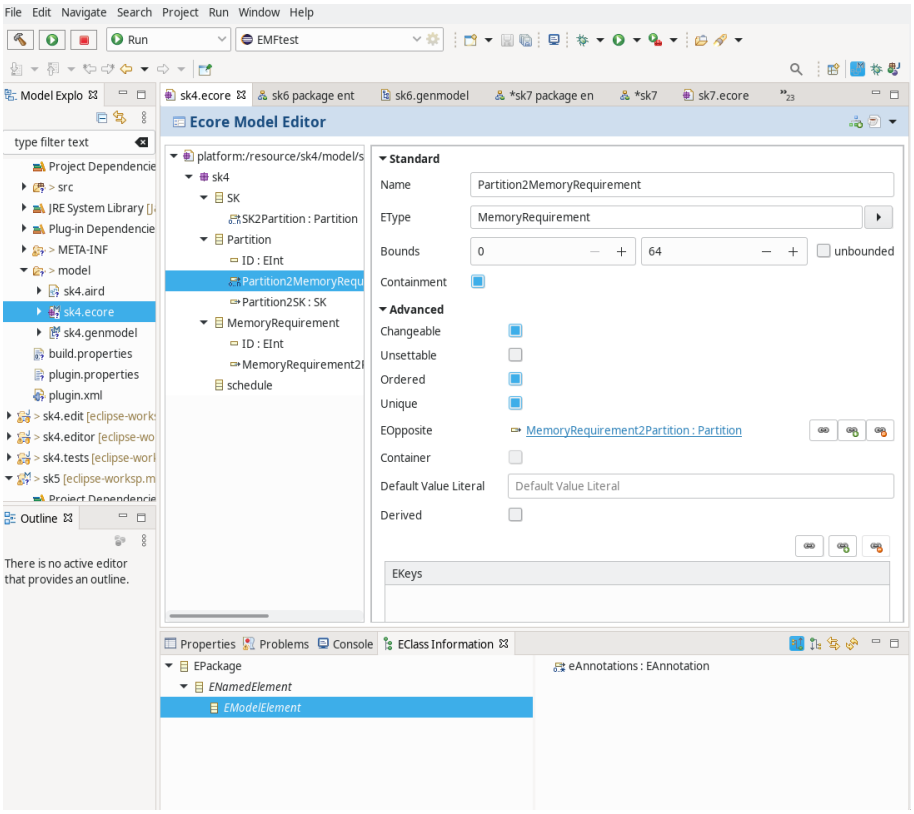


Figure 14: Use of EMF

itself. The second direction is correctness by construction, that is only to allow configurations that maintain certain safety / security properties. For instance, one check for correctness-by-construction properties (e.g. check for overlapping memory requirements).

A ground baseline that is already in place, is that every user configuration generated by a GUI tool of the separation kernel must be expressible as XML that complies to a certain XSD (XML schema definition).

To understand the existing tool landscape, in addition we have looked as EMF, which is a modeling framework that comes with a rich ecosystem of editing tools. In the context of ADMORPH we have specifically looked at the user interface and model editor of EMF with the aims of investigating whether it can help users of the separation kernel for generation of safety / security cases. The idea is that each concrete configuration of the separation kernel by a user / customer is a model, and that a space of possible configurations is the metamodel (space of correct-by-construction models). For the metamodel, we can specify containment relations such as e.g. relation of memory requirements to partitions (Figure 14).

After having specified these relations in the metamodel, eclipse can automatically generate a model editor where the user can by point-and-click generate models safisfying the constraints of the metamodel. Moreover, it is possible to add additional user-defined constraints in the Acceleo Query Language, e.g. such as that each memory region only shall be assigned to one partition (this

requirement only makes sense when no memory sharing is intended, so it depends on the user). This functionality is interesting because it would allow to combine user-provided or vendor-provided additional constraints to the ground model by the use of certain "views", that is technically additional constraints imposed by a safety case could be encoded here. One could also use such a metamodel/view pairing to indicate a safety envelope (metamodel) for dynamic configuration (taking the role of views).

A possible downside of an EMF+Acceleo combination is that it uses relatively "heavy" machinery, that is the Eclipse and Sirius frameworks, which might be difficult to tool-qualify in certifications.

Also, for any practical tool for safety cases, user acceptance and the status of the tool landscape ecosystem is key. Before "jumping" to this framework, we possibly intend to get further feedback and/or to look at alternative formalization approaches.

9 Conclusion

In conclusion, our work on analysis techniques for adaptive and morphing systems has greatly progressed.

On one hand, the ability to simulate the behaviour of these system is a signficant step towards understanding and providing guarantees on the reconfiguration and adaptation procedure and towards the implementation of adaptation policies. Our simulator is also the key element that allows us to fast-prototype our analysis techniques and test them on realistic data and scenarios before the application to the real hardware and software.

The research on adaptive scheduling policies has progressed steadily allowing us to distinguish between redundant and non-redundant execution and to introduce different degrees of redundancy and thus safety.

We defined a task model geared towards reconfiguration and certification and we have taken the first step into the formal analysis of problems that may occur and can cause tasks to miss their deadlines. We believe this will give us the foundations that we need to implement adaptive and morphing systems in practice.

10 References

[1] L. Ahrendts, S. Quinton, T. Boroske, and R. Ernst. Verifying weakly-hard real-time properties of traffic streams in switches networks. In Sebastian Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, volume 106, pages 15:1–15:22, 2018.

[2] Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert Davis. An empirical survey-based study into industry practice in real-time systems. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 3–11, 2020.

[3] Amir Behrouzian, H.A. Ara, Marc Geilen, Dip Goswami, and Twan Basten. Firmness analysis of real-time tasks. *ACM Trans. Embed. Comput. Syst.*, 19(4), July 2020.

[4] Guillem Bernat. *Specification and analysis of weakly hard real-time systems*. PhD thesis, Department de les Ciéncies Matemátiques i Informática, Universitat de les Illes Balears, Spain, 1998.

[5] Guillem Bernat, Alan Burns, and A. Liamosi. Weakly hard real-time systems. *IEEE Transactions on Computers*, 50:308 – 321, 2001.

[6] Cristiana Bolchini, Matteo Carminati, Marco Gribaudo, and Antonio Miele. A lightweight and open-source framework for the lifetime estimation of multicore systems. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 166–172. IEEE, 2014.

[7] Alan Burns and Robert Davis. Mixed criticality systems – a review. *Department of Computer Science, University of York, Tech. Rep*, pages 1–69, 2013.

[8] Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn Brandenburg. Response-time analysis of ROS2 processing chains under reservation-based scheduling. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133, pages 6:1–6:23, 2019.

[9] Guillermo Campos Ciro, Frédéric Dugardin, Farouk Yalaoui, and Russell Kelly. A nsga-ii and nsga-iii comparison for solving an open shop scheduling problem with resource constraints. *IFAC-PapersOnLine*, 49(12):1272–1277, 2016.

[10] Kalyanmoy Deb and Himanshu Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: solving problems with box constraints. *IEEE transactions on evolutionary computation*, 18(4):577–601, 2013.

[11] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. Deap: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13(70):2171–2175, 2012.

[12] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m, k)-firm deadlines. *IEEE Transactions on Computers*, 44(12):1443–1451, December 1995.

[13] Z.A.H. Hammadeh, R. Ernst, S. Quinton, R. Henia, and L. Rioux. Bounding deadline misses in weakly-hard real-time systems with task dependencies. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 584–589, 2017.

[14] 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.

[15] C. Huang, W. Li, and Q. Zhu. Formal verification of weakly-hard systems. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '19, page 197–207, New York, NY, USA, 2019. Association for Computing Machinery.

[16] 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.

[17] Hisao Ishibuchi, Ryo Imada, Yu Setoguchi, and Yusuke Nojima. Performance comparison of nsga-ii and nsga-iii on various many-objective test problems. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pages 3045–3052. IEEE, 2016.

[18] Himanshu Jain and Kalyanmoy Deb. An evolutionary many-objective optimization algorithm using reference-point based nondominated sorting approach, part ii: handling constraints and extending to an adaptive approach. *IEEE Transactions on evolutionary computation*, 18(4):602–622, 2013.

[19] David S Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[20] Oliver Kramer. *Genetic algorithm essentials*, volume 679. Springer, 2017.

[21] 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.

[22] 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.

[23] Martina Maggio, Arne Hamann, Eckart Mayer-John, and Dirk Ziegenbein. Control-system stability under consecutive deadline misses constraints. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:24. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

[24] Maurizio Palesi and Tony Givargis. Multi-objective design space exploration using genetic algorithms. In *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 67–72, 2002.

[25] 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.

[26] Paolo Pazzaglia, Youcheng Sun, and Marco Di Natale. Generalized weakly hard schedulability analysis for real-time periodic tasks. *ACM Trans. Embed. Comput. Syst.*, 20(1):3:1–3:26, 2021.

[27] Andy D. Pimentel. Exploring exploration: A tutorial introduction to embedded systems design space exploration. *IEEE Design & Test*, 34(1):77–90, feb 2017.

[28] Roberta Piscitelli and Andy D Pimentel. Design space pruning through hybrid analysis in system-level design space exploration. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 781–786. IEEE, 2012.

[29] P. Ramanathan. Overload management in real-time control applications using (m, k)-firm guarantee. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):549–559, 1999.

[30] Youcheng Sun and Marco Di Natale. Weakly hard schedulability analysis for fixed priority scheduling of periodic real-time tasks. *ACM Trans. Embed. Comput. Syst.*, 16(5s):171:1–171:19, 2017.

[31] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[32] Gang Tu, Jun-lin Li, Fu-min Yang, and Wei Luo. Relationships between window-based real-time constraints. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, pages 394–399, 2007.

[33] 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.

[34] Shih-Lun Wu, Ching-Yuan Bai, Kai-Chieh Chang, Yi-Ting Hsieh, Chao Huang, Chung-Wei Lin, Eunsuk Kang, and Qi Zhu. Efficient system verification with multiple weakly-hard constraints for runtime monitoring. In Jyotirmoy Deshmukh and Dejan Ničković, editors, *Runtime Verification*, pages 497–516, Cham, 2020. Springer International Publishing.

[35] W. Xu, Z.A.H. Hammadeh, A. Kröller, R. Ernst, and S. Quinton. Improved deadline miss models for real-time systems using typical worst-case analysis. In *27th Euromicro Conference on Real-Time Systems*, pages 247–256, 2015.