Organic Programming of Dynamic Real-Time Applications

Dissertation

A thesis submitted to the
Faculty of Computer Science, Electrical Engineering and Mathematics
of the
University of Paderborn

Lial Khaluf

Paderborn, Germany
11.09.2018

Acknowledgement

I would like to thank my advisor Prof. Franz Rammig for his scientific support, and references he has provided me.
Thank you for discussions we had and for his encouragement.

I would like also to thank my family for their spiritual support and encouragement.

Acknowledgement

I would like to thank my advisor Prof. Franz Rammig for his scientific support, and references he has provided me.
Thank you for discussions we had and for his encouragement.

I would like also to thank my family for their spiritual support and encouragement.

Index

Abstract

Systems as e.g., control systems, software systems, etc. are tending nowadays to inspire their behavior from organic systems. Existing approaches have tried to develop a system behavior and reactions to environmental circumstances that can be applied at run time. Sometimes, they even can learn from previous states. In the real-time domain, systems are still limited by pre-defined behaviors when adapting themselves to the environment. The pre-defined behaviors can be applied at run time, and optimize system performance. However, new behaviors cannot be added at run time. This limits the system ability to react to newly emerging environmental changes. In our thesis, we provide an approach, which is able to react at run time and preserve all real-time constraints. Reactions can happen at run-time, in the sense of adding new tasks or updating existing ones. Our approach can be applied to all kinds of real-time systems. It consists of an adaptation algorithm that on its part behaves as a real-time task. Under the assumption that the tasks of the real-time environment exist in form of multiple variants, it provides a selection mechanism for other all of these tasks in the system. This is done by potentially exchanging the current variant of tasks in order to optimize global system goals, whenever an adaptation is required. Adaptation can include adding or updating a tasks or set of tasks, in addition to task deletion. System goals aim to reduce costs under the constraint of meeting all real-time requirements. In accordance to the concept of "Organic Programming", we make use of the concept of cells. A cell is an extension of a task. Each cell may have several variants, and may experience several updates, where each update may in turn consist of several variants. All variants of a cell share the same fundamental functionality, however under different timing requirements and different costs. We provide a proof that the algorithm is bounded and estimate analytically the time complexity.

Chapter 1 Introduction

Turning any physical process into an online process is the current trend in many kinds of businesses. This evolution is taking place by transforming the current physical systems into Cyber Physical Systems. In these systems, digital extensions of the human and physical factors involved in the processes are used for an appropriate communication to fulfill the tasks defined by the processes. The correct functionality of the system is influenced by its reaction to internal and external events in real-time. Here, internal events could be triggered for example by a change in local environment. External events could be triggered by a change in global environment.

Cyber Physical Systems (CPS) introduce a new way to integrate cyber world with physical world in that computations take part to influence the physical processes and vice versa. This applies in general to control processes, as for example, in the medical sector, in the automotive sector, in energy sectors, etc. The nature of such processes belongs normally to embedded systems where timing constraints should be achieved. However, Cyber Physical Systems add several advantages over the traditional systems, as e.g., self-adaptability to failure as well as unexpected conditions [1].

In this sense, the system is evaluated by its ability to adapt itself to environmental changes in real-time. Many approaches have been introduced to solve this challenge. However, the existing approaches have several limitations related to the ability of reacting to unexpected events, or reacting in a non-predefined way. In order to overcome these deficiencies, we introduce in this thesis a solution that mimics the organic behavior of objects in our real world.

Real world objects have the ability to change their structure or behavior when they react to any environmental event, as cells do in an organism [2]. For this reason, our solution does not limit itself to a predefined set of events or reactions. It is assumed to allow the system to grow at run time. In other words, to have new resources, new events and reactions at run time. Currently, we apply our algorithm on a central node, with the ability to import the needed information from a remote node. This information consists of the different reactions that the system may apply against specific events that may result from an internal or external environmental change. The reactions are developed by different sources, and added to the system at run time. The infrastructure of the system, has the ability to be enlarged and have many identical nodes (network of nodes). These nodes can exchange information through a network. This aspect, however, is not discussed in the present thesis.

The solution we provide applies for all kinds of real-time systems. This is done by providing the system with organic properties on the level of real-time tasks. The tasks in this case are transformed into cells, called real-time cells, by

making a slight modification to the general structure of real-time tasks. The modification adds a set of meta data. The set enables to change the content of a task (structure and behavior) at run time.

In this chapter, we introduce a scenario for applying the solution described in the thesis. In chapter 2, we provide an introduction to real-time systems, with the specified concepts used in the thesis. It then describes a survey on the optimization problems including the solution technique used in the thesis. Chapter 3 explains the main concept of the problem discussed in the thesis. Chapter 4 presents the related work. Chapter 5 describes the problem, which we solve in the thesis, starting by the simplest case and ending by the most complex case. Chapter 6 provides a proof on the boundedness of the solution, and a calculation for the time complexity. Chapter 7 includes a conclusion on it.

1.1 Scenario

Let us assume a telerobotic surgery system [3], where the surgeon is performing the surgical operations remotely with the help of a robotic surgery system, a set of surgical instruments, a set of endoscopic tools, a set of medical, technical, and energy resources, and a deterministic network as illustrated by Figure 1.

The surgical operations are taking place online, where the surgeon deals with the digital extension of the patient and the patient is operated by the digital extension of the surgeon. This structure defines a Cyber Physical System, consisting of a Master and a Slave, see Figure 2. On the Master side, the robotic surgical system provides a vision system that translates the information coming

from the Slave side into a digital extension of the patient. On the Slave side, the system provides a controller which translates the decisions coming for the Master side into instructions to be applied by the robotic arms, endoscopic tools and other instruments which will in turn act as a digital extension of the surgeon. The ability of the system showed in Figure 1 to adapt itself to the evolutions of surgical operations is limited by the surgeon’s ability to react to these evolutions with the required speed so that the operations are performed successfully. This results from the fact that the surgeon performs the operations by interacting directly with the digital extension of the patient through the master console.

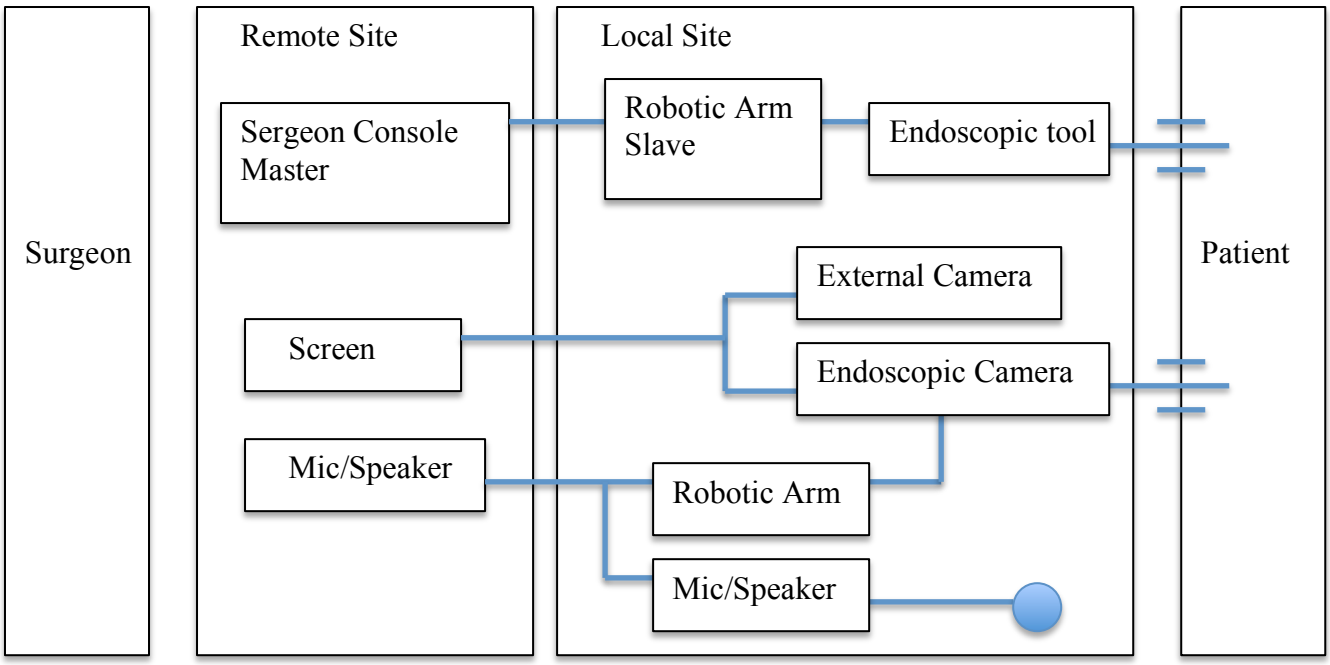


Figure 1: Telerobotic Sergery [3]

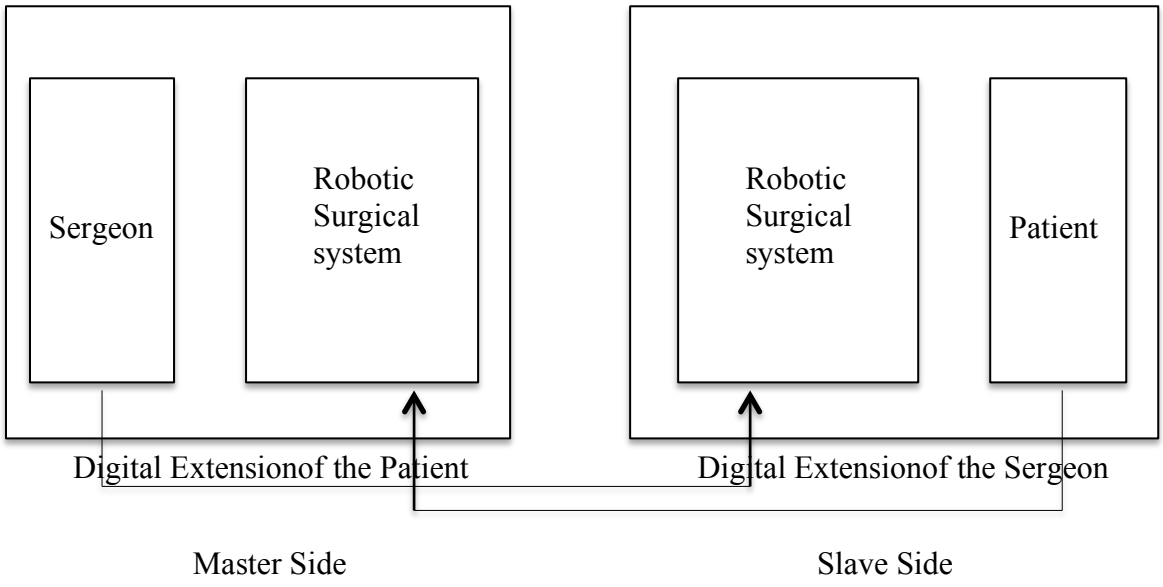


Figure 2: Cyber Physical System of Telerobotic Surgery

To overcome this limitation, we assume that the surgeon is only responsible for deciding which surgical actions should take place during the operation. However, the actions steps and characteristics are predefined and performed by the system and according to the system parameters. The previous assumption defines the surgical operation to be a set of surgical actions that are triggered online and must be accomplished in real-time. This set should be able to change

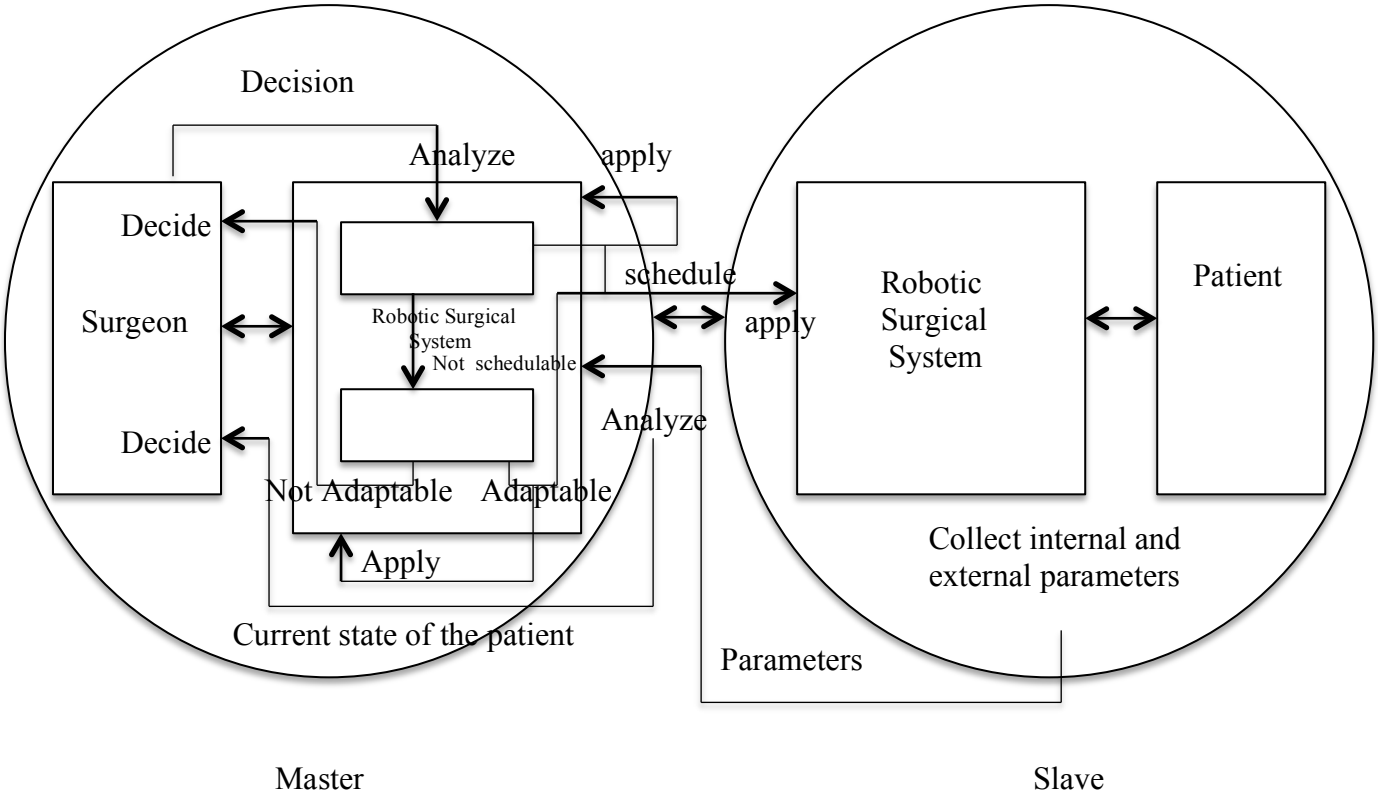


Figure 3: Self-Adaptable System of robotic surgery

its structure and behavior at run-time to enable the system to adapt itself to environmental changes on the slave side. Environmental changes could be internal, reflecting the changes in the patient state or external, reflecting the changes in the instruments, tools, and energy resources. In other words, these changes represent the events that might trigger the decision of adaptation. However, the adaptation process should preserve all real-time constraints. Here, the overhead imposed by the adaptation process itself has to be considered as well. The evolution of the system is event-based and time-based, and this in turn defines the system to be a hybrid system.

To perform the surgical operation successfully, the robotic surgical system collects all internal and external parameters that reflect the environment state on

the Slave side. The parameters are then analyzed by the system on the Master side and represented using a vision system. This enables the surgeon to read the current state of the patient and to decide if a new surgical action should take place or a currently running surgical action should be updated. The decision is then studied by the system to see whether it has influence on meeting the real-time constraints of the surgical operation. If no negative influence exists, the decision is applied to the Slave side. However, if this is not the case, an adaptation algorithm is run to check whether there is a possibility to change the structure and behavior of the current surgical actions set in a way that enables to apply the decision and preserves all real-time constraints. If this succeeds, the set is modified and the decision is applied. Otherwise, the surgeon is informed about the necessity to make another decision, see Figure 3.

The surgical operation aims to accomplish the needed surgical actions successfully avoiding to fall in any dangerous situations that might result from inappropriate responses by the patient, and at the same time the operation is influenced by any factor that the time, quality or accomplishment may depend on. For this reason, the system collects the patient's measurements as e.g. pressure, temperature, view of the surgical field,... and calls them the internal parameters. It also collects the measurements of the environmental factors as e.g. the energy sources including light, temperature, etc ... of the surgical room, and the measurements of other resources as e.g. the numbers and kinds of surgical instruments, endoscopic tools, medical equipment,... calling them the external parameters.

The internal and external parameters are the primary factors to classify the surgical actions into:

- actions that could represent updates for handling possible evolutions of a surgical state
- or updates for handling a surgical state in different ways or with different instruments.

Here, we define a task to be a surgical action which is set to handle a specific surgical state characterized by a primary range of internal parameters. In this sense, the task consists of the required positioning and movement actions of the robotic arms, instruments and tools.

The primary range is the range of parameters that define an initial status of the patient.

A task update is a resulting task defined to handle a specific contingency of a surgical state characterized by a range of internal parameters different from the primary range of the origin task. This contingency might happen as a reaction of medications or infections or any other factors that might in turn change the range of internal parameters. Each task or task update may have several variants, which in turn are tasks dedicated to accomplish the same surgical action of that task or update, however, maybe characterized by different external parameters, and may differ by their positioning and movement actions or their order. As a result, variants and updates may differ also by time characteristics. Figure 4 shows the online growing set of tasks. The vertical axis denotes the variants within a class, the horizontal axis denotes the different classes and the third dimension axis denotes the different updates (update version bound to an entire class).

The adaptation process depends on the task updates and variants. Whenever the parameters are represented on the Master side, the surgeon analyzes the patient state and triggers a new task if a new surgical action is needed. This means choosing an appropriate task variant according to the capacity fraction and to the internal and external parameters measured by the system. The surgeon also might trigger a task update to handle any contingency of the surgical state, on which the task operates. This means choosing an appropriate variant that belongs to the task update according to the updated capacity fraction and to the external parameters measured by the system.

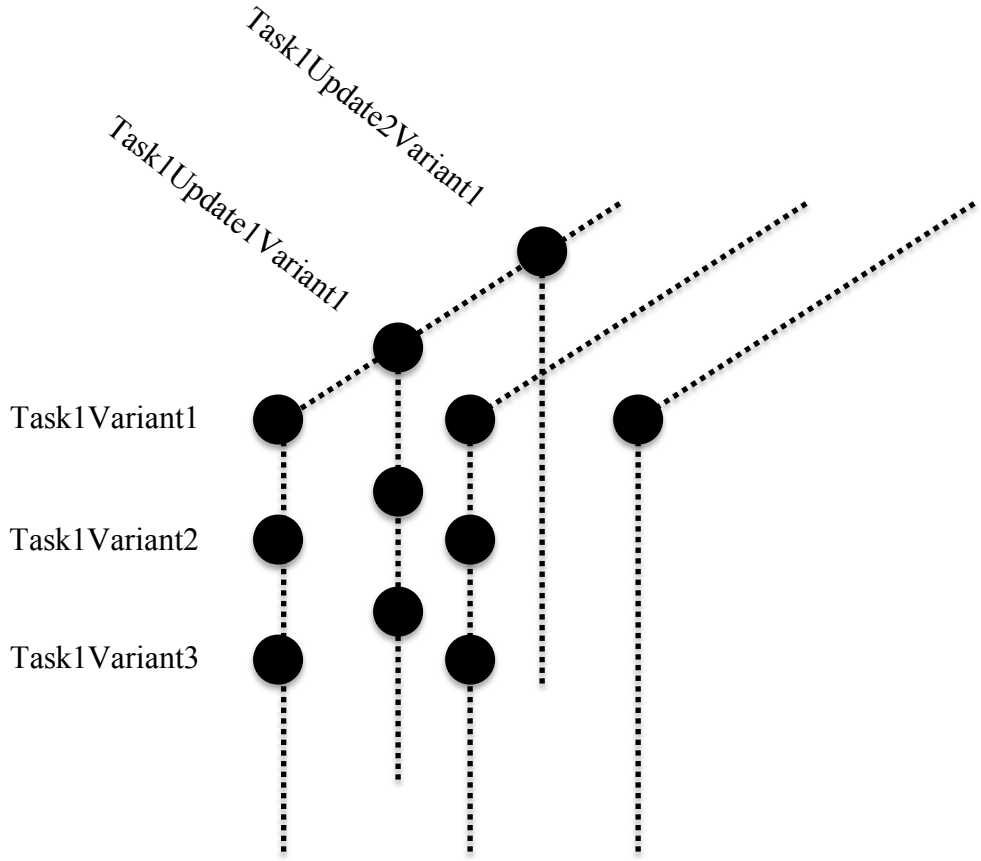


Figure 4: Task Updates and variants

In case of triggering a new task or triggering a task update, the decision is applied only, if the triggered variant may satisfy system restrictions. However, if no valid solution is found, the surgeon is informed about the necessity to think of a new decision. The result of the algorithm differs according to the measured values of related parameters and to the available set of variants. Both factors are not constant, since surgical instruments, endoscopic tools and other resources can be added to or removed from (e.g., due to defects) the Slave side at run-time. Also tasks, their updates and variants as illustrated by Figure 4 can be added to or removed from the Master side at run-time, and can be even exchanged between Masters if several systems are connected to each other as pointed out by Figure 5. The structure in Figure 5 connects Masters to each other, as well as each Master to one Slave and each Slave to one Master, where

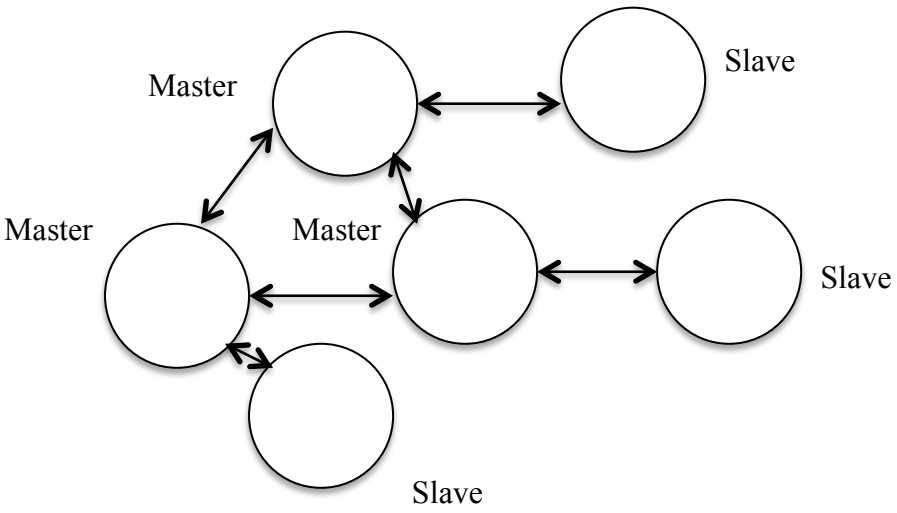


Figure 5: Connected self-adaptable systems of robotic surgery

each Master has one surgeon and each Slave has one patient. In this context, Masters represent different storage places for tasks. If the surgeon realizes the necessity for several updates or new tasks at the same time, the surgeon decides the order of handling them. The surgeon might also realize when choosing a new task that it depends on other new tasks, which might also have dependencies between each other. In this case, achieving these dependencies will be an additional condition for accepting the task.

In the context of the present thesis we abstract from the described case study as follows: In the present thesis we concentrate on real-time systems apart from their type or goals. Hereby, we make the following restricting assumptions:

- Adaptations happen relatively seldomly compared to the frequency of "normal" activities.
- System tasks differ according to the type of real-time system.
- Each task may exist in several variants. Each variant belongs to a set of variants, which in turn belong to a specific class of task updates.
- Adaptations may happen, when
    - A new task variant arrives to the system.
    - A new variant update arrives to the system.
    - A new set of dependent variants arrives to the system.
    - An update for a set of dependent variants arrives to the system.
    - A task variant shall be removed from the system.
    - A set of dependent variants shall be removed from the system.

- System keeps running in the old state until the adaptation is sucessfully applied.

- System is informed if adaptation fails.

Chapter 2 Foundations

2.1 Real-Time Systems

Real-time systems refer to systems that serve tasks, which should be accomplished within predefined deadlines. This is done using scheduling algorithms. The algorithms are defined and classified according to types, properties of tasks and dependencies between them. Here, we introduce the basic concepts of real-time systems [4].

2.1.1 Basic Concepts

- Real-time systems: are computing systems, in which the correctness of behavior depends not only the computation results but also on the response time. "Examples of real-time systems could be automotive applications, flight control systems, robotics, etc" [4]

- A real-time task: is characterized by the following properties, see Figure 6:

1- Arrival time (a): it is the time at which the task is released and becomes ready for execution, called also release time.
2- Execution time (C): is the time required to execute the task without interruption.
3- Absolute deadline (d): is the time that the task execution should not exceed.
4- Relative deadline (D): is the time difference between the absolute deadline and the arrival time.
5- Start time (s): is the time at which the task starts its execution.
6- Finishing time (f): is the time at which the task finishes its execution.

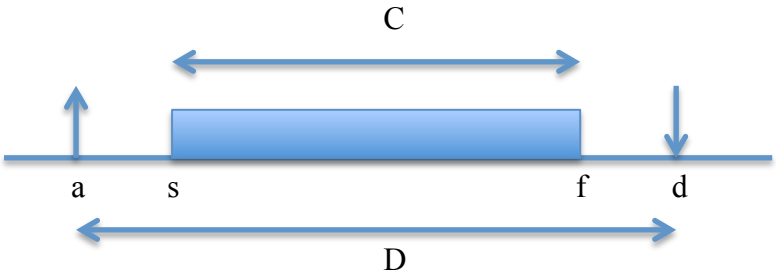


Figure 6: Real Time Task

7- Criticalness: is a parameter that indicates whether the task is hard or soft (explained in the next definitions).
8- Value (v): is a parameter, which indicates the importance of the task among the existing tasks

9- Lateness (L): L = f – d. The lateness of a task is negative, if the task completes its execution before its deadline.
10- Tardiness (E): E = max (0,L) is the time by which the tasks exceed their deadline.
11- Laxity or Slack time: X = d - a - C is the maximum time by which the execution of the task could be delayed after its arrival, so that it still meets its deadline.

The first four properties of a task are referred to as the time characteristics of the task.

- A soft real-time task: is a task that does not cause a catastrophic result when its deadline is not met, but might cause a decrease in the performance, e.g., handling input data from the keyboard.

- A hard real-time task: is a task that may cause catastrophic results in the system environment if its deadline is not met, e.g., sensory data acquisition.

- A periodic task (τ): is a task that is activated in regular time periods, where each activation is called an instance of the task.

- An aperiodic task (J): is a task that is activated in irregular time periods, where each activation is called an instance of the task.

- Precedence constraints: represent the precedence order that tasks might have to respect concerning the order of their execution. Precedence constraints are normally represented by a directed acyclic graph DAG (has no directed circles [5]). E.g., Figure 7 illustrates precedence constraints of the aperiodic tasks J_1, J_2, and J_3.

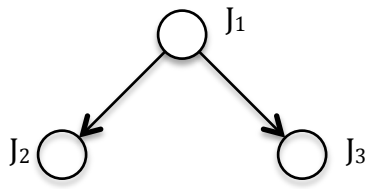


Figure 7: Precedence Constraints

J_1 is an immediate predecessor of J_2 and J_3. J_1 is called a beginning task. J_2 and J_3 are called ending tasks.

- Resources: a resource is a software structure that can be used by a task to enable its goal. E.g., a memory area.

- Processors: called CPUs. A CPU consists usually of an arithmetic logic unit, a control unit, and registers.
"The arithmetic logic unit performs operations, such as addition and subtraction on the data. The control unit coordinates computer activities. The registers are data storage cells that are used for the temporary storage of data." [6]

- Scheduling: A scheduling problem is defined by a set of tasks $\{J_1, J_2, .., J_N\}$, set of processors $\{P_1, P_2, .., P_M\}$, and a set of resources $\{R_1, R_2, .., R_S\}$, where precedence constraints might exist between the different tasks, and each task is characterized by a set of time characteristics. Here, scheduling means to assign the processors, and the resources to the tasks so that they can meet their timing constraints. Scheduling algorithms are classified into the following classes:

- Preemptive: allows to interrupt tasks during their execution, in order to execute other tasks which satisfy a condition that is predefined by the algorithm.
- Non-Preemptive: does not allow to interrupt tasks during their execution.
- Static: depends on parameters, which are predefined before the activation of tasks takes place.
- Dynamic: depends on parameters, which may change at run time.
- Off-line: runs on the set of tasks before their activation.
- On-line: is triggered at run time by the events of task arrival or termination.
- Optimal: minimizes the costs of tasks. Normally, cost is expressed as a function of parameters.
- Heuristic: tends to find an optimal schedule but with no guarantee to achieve this.

- Feasible schedule: is a schedule that achieves the execution meeting all predefined constraints. "Any set of tasks is said to be schedulable if there exists at least one algorithm that can produce a feasible schedule." [4]

In the following we describe the scheduling algorithms that are used in our approach.

2.1.2 Earliest Deadline First

- Earliest Deadline First (EDF): is an optimal, preemptive, dynamic priority scheduling algorithm, where at any point of time, the task with the shortest absolute deadline is executed. EDF can be applied on periodic as well as aperiodic tasks.
If and only if the following statement holds for a set of N periodic tasks:

$$\sum_{i=1}^{N} C_i/T_i \leq 1 \quad (1)$$

then the set is schedulable, where C_i is the execution time for the i_{th} task and T_i is the period of the i_{th} task. [4]

Example: consider the periodic tasks $\tau_1(C_1 = 1, T_1 = 4,\ D_1 = 4)$, $\tau_2(C_2 = 4, T_2 = 6,\ D_2 = 6)$

This set of tasks is schedulable under EDF as $\sum_{i=1}^{N} C_i/T_i = \frac{1}{4} + \frac{4}{6} < 1$.
Figure 8 shows the execution sequence of these tasks:

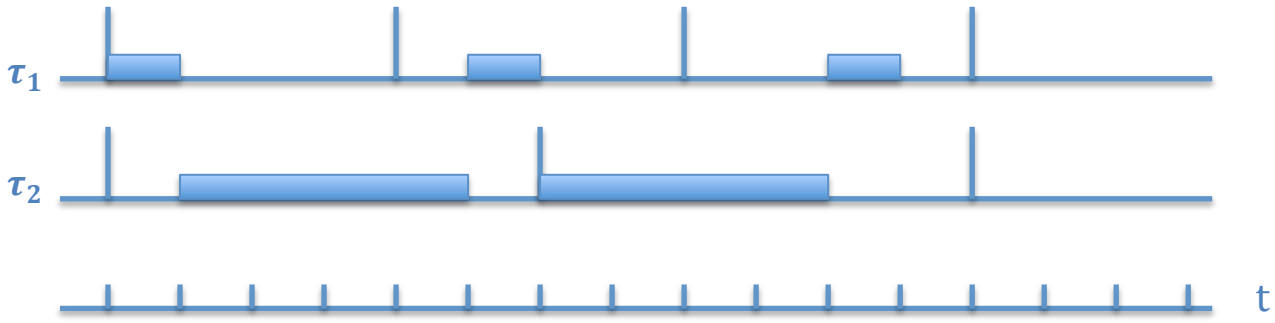


Figure 8: The Execution Sequence of Periodic Tasks

The case of the following algorithm [4] is to guarantee the schedulability of a set J of aperiodic tasks and a new task J_{New}:

"{Assuming that $J` = J \cup \{J_{New}\}$, and the tasks are sorted with respect to their absolute deadlines;
t = current-time;
$f_0 = 0$;
for $(each\ J_i \in J`)\{$
$f_i = f_{i-1} + C_i(t)$;
$if(f_i > d_i)\ return\ (INFEASIBLE);\}$
$return\ (FEASIBLE)$;
}"

" $C_i(t)$ is initialized to be equal to C_i and then updated whenever the task is preempted. It is the remaining worst case execution time of task i at time t. f_i is the worst case finishing time of the task J_i. f_{i-1} is the worst case finishing time of the task J_{i-1}."

2.1.3 Earliest Deadline First with precedence constraints

- EDF with precedence constraints EDF*: in [7] an algorithm is developed to solve the scheduling problem of aperiodic tasks with precedence constraints. This takes place by transforming the set J of the dependent tasks into a set J^* of independent tasks which can be then scheduled by EDF. In this sense, J is schedulable if and only if J^* is schedulable under EDF.
The mentioned transformation is achieved by modifying the arrival (release) times and deadlines of the tasks as follows:

- Modification of release times: if a task J_1 is an immediate predecessor for another task J_2 :$J_1 \rightarrow J_2$, then in order to get a feasible schedule, the following conditions should be satisfied: $s_2 \geq a_2$ & $s_2 \geq a_1 + C_1$.
For this reason, a new release time a_2^* is calculated: $a_2^* = \max(a_2, a_1 + C_1)$ by applying the following steps:

1- Set $a_i^* = a_i$ for any beginning task in the acyclic graph.
2- Choose a task J_i with a non modified a_i, where the arrival time for all its immediate predecessors J_l has been modified. If such a task is not found, then break the algorithm.
3- Set $a_i^* = \max[a_i, max(a_l^* + C_l : J_l \rightarrow J_i)]$.
4- Go to step 2.

- Modification of deadlines: if a task J_1 is an immediate predecessor for another task J_2 :$J_1 \rightarrow J_2$, , then in order to get a feasible schedule, the following conditions should be satisfied: $f_1 \leq d_1$ & $f_1 \leq d_2 - C_2$
For this reason, a new deadline d_1^* is calculated: $d_1^* = \min(d_1, d_2 - C_2)$ by applying the following steps:

1- Set $d_i^* = d_i$ for any ending task in the acyclic graph.
2- Choose a task J_i with a non modified d_i, where the deadline for all its immediate successors J_l has been modified. If such a task is not found, then break the algorithm.
3- Set $d_i^* = \min[d_i, \min(d_k^* - C_k : J_i \rightarrow J_k)]$.
4- Go to step 2.

In [8] we find a survey on aperiodic servers. These servers are appropriate to handle task sets that consist of both, periodic tasks and aperiodic ones.
The server introduced in the following is one of the techniques used to provide better response time for the soft real-time aperiodic tasks and also meeting the hard deadlines of periodic tasks. It is the simplest server to be implemented among the different approaches. We use TBS in our approach to handle periodic and aperiodic tasks with hard deadlines.

2.1.4 Total Bandwidth Server Algorithm

- Total Bandwidth Server Algorithm (TBS): is an online algorithm, used normally for servicing soft aperiodic requests, where the deadlines of hard periodic tasks can be preserved under EDF. TBS is the most appropriate server for implementation with respect to time complexity. The processor is fully utilized achieving a good aperiodic responsiveness [8]. In the following is a short introduction on how the server works, and its schedulability test:

- The algorithm is based on assigning absolute deadlines to the aperiodic requests depending on a known server utilization U_s.

$$d_k = \max(r_k, d_{k-1}) + C_k/U_s \qquad (2) \quad [8]$$

Where r_k is the release time for the k_{th} aperiodic task. C_k is the execution time of the k_{th} aperiodic task. $d_0 = 0$
The server utilization U_s is calculated as follows:

$$U_s = 1 - U_p \qquad (3)$$

The schedulability test of the TBS is defined as follows:

$$U_p + U_s \leq 1 \qquad (4)$$

Where,

$$U_p = \sum_{i=1}^{N} C_i/T_i \qquad (5)$$

Where C_i is the execution time of the i_{th} periodic task, T_i is the period of the i_{th} periodic task.

Example:

Let us assume that we have two periodic tasks described by Table 1, and two aperiodic tasks described by Table 2. The periodic tasks arrive at time t = 0. These tasks should run under EDF using the TBS algorithm

	C_i	$D_i = T_i$
τ_1	1	8
τ_2	4	16

Table 1: Periodic Tasks

	a_i	C_i
J_1	2	1
J_2	5	2

Table 2: Aperiodic Tasks

$$U_p = 1/8 + 4/16 = 1/8 + 1/4 = 3/8 = 0.375$$

The absolute deadlines of the periodic tasks are calculated according to the following rule

$$d_{i,j} = a_i + (j-1)\, T_i + D_i \quad (6)$$

Where i indicates the i_{th} periodic task, and j indicates the j_{th} instance of the task.

The absolute deadlines of τ_1 instances are calculated according to (6)

$d_{1,1} = 0 + (1-1) * 8 + 8 = 8$
$d_{1,2} = 0 + (2-1) * 8 + 8 = 16$
$d_{1,3} = 0 + (3-1) * 8 + 8 = 24$
.
.

The absolute deadlines of τ_2 instances are calculated according to (6)

$d_{2,1} = 0 + (1-1) * 16 + 16 = 16$
$d_{2,2} = 0 + (2-1) * 16 + 16 = 32$
$d_{2,3} = 0 + (3-1) * 16 + 16 = 48$
.
.

The utilization of the total bandwidth server is:

$$U_s = 1 - U_p = 1 - 0.375 = 0.625$$

The absolute deadline of J_1 is calculated according to (2)

$$d_1 = \max(2,0) + 1/0.625 = 3.6$$

The absolute deadline of J_2 is calculated according to (2)

$$d_2 = \max(5,3.6) + 2/0.625 = 5 + 3.2 = 8.2$$

Figure 9 explains the execution sequence.

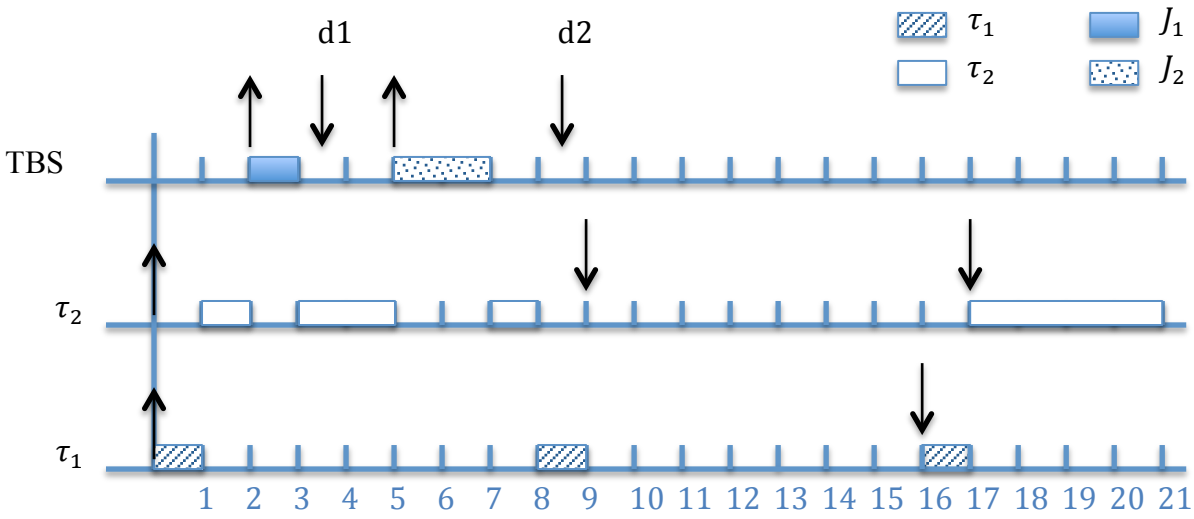


Figure 9: The execution sequence of the tasks set with the TBS

2. 2 Tasks and Variants

We generalize the concept of tasks and variants defined in the scenario. In our approach, a task is a mission that should be accomplished by a real-time system. Tasks may be added at run time to the system or may be removed. Existing tasks may be updated at run time. Usually, any modification at run time may cause exceeding one or more of deadlines (a non-feasible schedule). For this reason, our goal is to design a highly dynamic system that can adapt itself to environmental changes at run time. In other words, a system that can accept to

apply change on its tasks at run time. This aspect can be achieved by defining task variants. Each task variant consists of a structure and behavior that is dedicated to achieve a specific goal. The specific goal is shared by all variants of a task. Each task may have many variants. Variants of a task differ in time and cost characteristics. This difference may result from implementation of task procedure, or any resources, variables, or structures used by the task. This approach provides a wide range of choices when choosing best variants that may represent tasks at rune time, whenever a modification happens in the system. Best choice in our case depends always on time needed for accomplishing the task and cost of tasks.

Each task may have several updates according to system parameters. Each update have a separate set of variants.

The approach in [64] and [65], has defined different profiles with different resource requirements for each task. It enables choosing the best combination of profiles at run time to adapt the system to certain situations. However, these profiles are developed offline, and new ones cannot be added to the systems at run time, which decreases the system adaptation ability. The approach in this thesis applies the concept of organic programming by giving the ability to modify tasks online in a way that preserves all real-time constraints. The new form of a task is called a cell. Cells can be developed and added online to the system. Each cell may have different variants, which might update the task to which the cell is dedicated, or provide another method that delivers the same task of the cell, using different quality characteristics. This alternatives strategy enables to adapt the system to environmetal changes at runtime.

2.3 Optimization problems

In this chapter, we are going to present algorithmic problems, general foundations on optimization algorithms, as for example, evolutionary algorithms, genetic algorithms, genetic programming, etc. After that, we describe the knapsack problem, an NP-Hard combinatorial optimization problem, and its several types.

2.3.1 Algorithmic Problems

An algorithmic problem π is a mapping between a set of problem instances and a set of configurations π: $I \rightarrow 2^S$. I is the set of problem instances, and S is the set of configurations. For every instance, the set of configurations constructs the solution space. [10]

Example1: A decision problem is an algorithmic problem with a solution space $\{0,1\}$

Example2: In different network problems, the algorithmic problem could be a

mapping between undirected graphs and spanning trees. In this case I is the set of undirected graphs, and S is the set of spanning trees [10]. A spanning tree is a tree, which allows for only one connection between two nodes. [82]

In this section, we present the definition of complexity for different algorithm classes [10]:

Deterministic Algorithms: Here the complexity is defined by run time and storage space. An algorithm A solves a problem Π, if for an input that encodes an instance p ∈ I, the algorithm computes the solution s ∈ Π(p). The run time of the algorithm is denoted by T(A,p). The storage space is denoted by S(A,p).

Suppose ℓ(p) is the length of instance p, the worst-case complexity of A is:

"$T_{wc}(A,n) = \max\{T(A,p)|\ p \in I,\ \ell(p) = n\}$

$S_{wc}(A,n) = \max\{S(A,p)|\ p \in I,\ \ell(p) = n\}$" [10]

Randomized Algorithms: The randomization concept comes from the idea of having different values when throwing a coin. These values are attached to the input of the algorithm as an infinite sequence σ. In other words, the randomized algorithm is a deterministic algorithm with input (p, σ). Here the complexity is defined as follows:

"$T_{LV} = \lim_{k\to\infty} 2^{-k} \sum_{\sigma\in\{0,1\}^k} T(A,(p,\sigma))$

$S_{LV} = \lim_{k\to\infty} 2^{-k} \sum_{\sigma\in\{0,1\}^k} S(A,(p,\sigma))$

$T_{LV}(A,n) = max_{\ell(p)\,=\,n}\ T_{LV}(A,p)$

$S_{LV}(A,n) = max_{\ell(p)\,=\,n}\ S_{LV}(A,p)$" [10]

Feasible Algorithms: A feasible algorithm is an algorithm that can be solved with $T_{wc}(A,n) = O(n^{\alpha})$; $\alpha \geq 0$. A problem Π is said to be feasible if there is a feasible algorithm that can solve it.

Nondeterministic Algorithms: is dedicated to solve decision problems. The algorithm is said to solve a problem Π, if the following holds:

- If $\Pi(p) = 1$ for some instance p of the problem, then the algorithm finds a way for 1 to be computed.

- If $\Pi(p) = 0$ for some instance p of the problem, then then the algorithm finds no way for 0 to be computed.

Pseudopolynomial Algorithms: "a Pseudopolynomial algorithm that may solve a problem has a polynomial time in the length of the instance and in the size of

the numbers occurring in encoding the instance. The algorithm is efficient if the numbers involved are polynomial of the length of the instance. ” [10]

Approximation Algorithms: are dedicated to approximate the optimal solution. “The metric, which determines the deviation from the optimum may differ. For example, in a minimization problem, the error of an algorithm A is defined by:

$$\text{Error} = \max_{p \in I} \frac{c(A(p))}{c(opt(p))}$$

Where A(p) is the solution computed by algorithm A.

opt(p) is the optimal solution of p. ” [10]

2.3.2 Optimization: General foundations and Meta heuristics

“One of the most fundamental principles in our world is the search for an optimal state. It begins in the microcosm where atoms in physics try to form bonds in order to minimize the energy of their electrons [84]. When molecules form solid bodies during the process of freezing, they try to assume energy-optimal crystal structures. These processes, of course, are not driven by any higher intention but purely result from the laws of physics. The same goes for the biological principle of survival of the fittest [85] which, together with the biological evolution [86], leads to better adaptation of the species to their environment. Here, a local optimum is a well-adapted species that dominates all other animals in its surroundings. As long as humankind exists, we strive for perfection in many areas. We want to reach a maximum degree of happiness with the least amount of effort. In our economy, profit and sales must be maximized and costs should be as low as possible. Therefore, optimization is one of the oldest of sciences which even extends into daily life [87]”. [11]

After presenting algorithmic problems, and their different classes in the sense of complexity in the previous section, we provide an introduction for optimization algorithms. We use an optimization algorithm in our thesis as part of the solution for the present problem.

We describe here different types of optimization algorithms, as e.g, evolutionary algorithms: genetic algorithms, Learning classifiers systems, evolutionary strategy, evolutionary programming, and genetic programming. [11].

Evolutionary algorithms: are heuristics that aim to find a best solution. An evolutionary algorithm starts with an initial population. The population consists of several individuals. Each individual is characterized by a fitness value. The individuals with best values are selected to reproduce new individuals, as illustrated by Figure 10. [59]

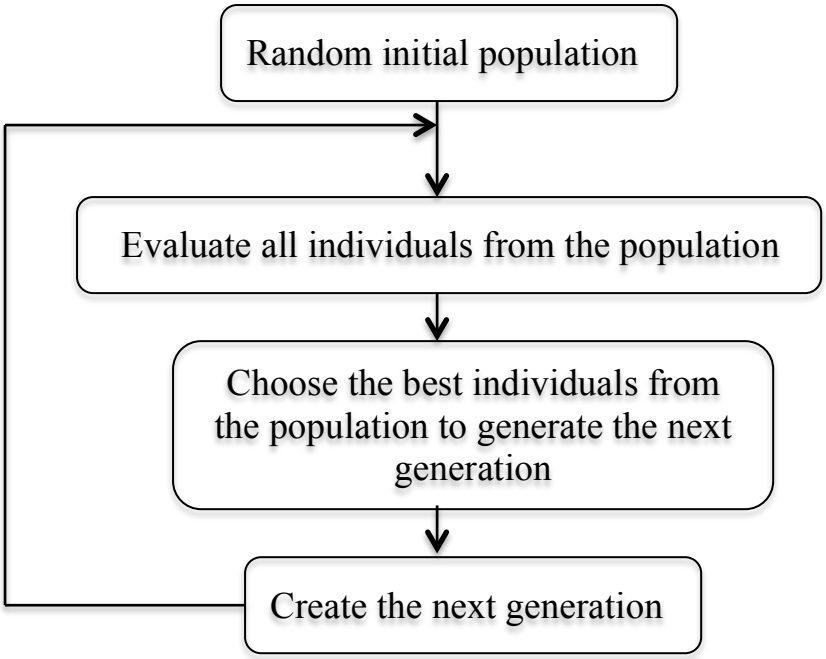


Figure 10: Evolutionary Algorithms

A genetic algorithm is a type of evolutionary algorithms. It consists of the following steps:

Initial population, evaluation, fitness assignment, selection, and reproduction.

Initial population: In this step the initial population is chosen. The initial population consists of individuals.

Evaluation: evaluates the current population according to an objective function.

Fitness assignment: determines the fitness of the population.

Selection: selects the fittest individuals for the reproduction process.

Reproduction: applies crossover and mutation to generate new individuals.

The principle is to reach an optimal solution. Reaching this solution is done by searching the design space to find an initial population. The individuals of this population are tested according to an objective function. New generations are then produced from the current generation by applying selection, crossover and mutation. An individual may be a number, set of integers, two dimensional or three dimensional variable, etc. Finding the initial population could be done randomely, by going through an algorithm, or by following other methods.

Learning classifiers systems

Learning classifiers systems achieve the optimization goal by reading the environment parameters in the form of messages coming from sensors. The messages are then processed by the classifiers. The result is encoded in a message, which represents new rules or is delivered to the effectors. Figure 11 points out the process. The effectors apply the received rules (if-then) to the system. The payoff represents the environment feedback. It measures the system performance. [11][36]

Learning classifier systems is a kind of rule-based machine learning. It depends on two componenets, a discovery componenet, and a learning component. The first one includes normally a genetic algorithm. The second may include supervised learning , unsupervised learning, or reinforcement learning. [83]

The strategy of the learning classifier system can be summarized by three steps [36]: Parallelism and coordination: When the system does not have a single rule to apply the received message, several rules can be applied in parallel.
Credit assignment: gives a feedback on the system behavior, and as a result decides which rules are giving the most successful behavior.
Rule discovery: refers to generating new rules using past experience.

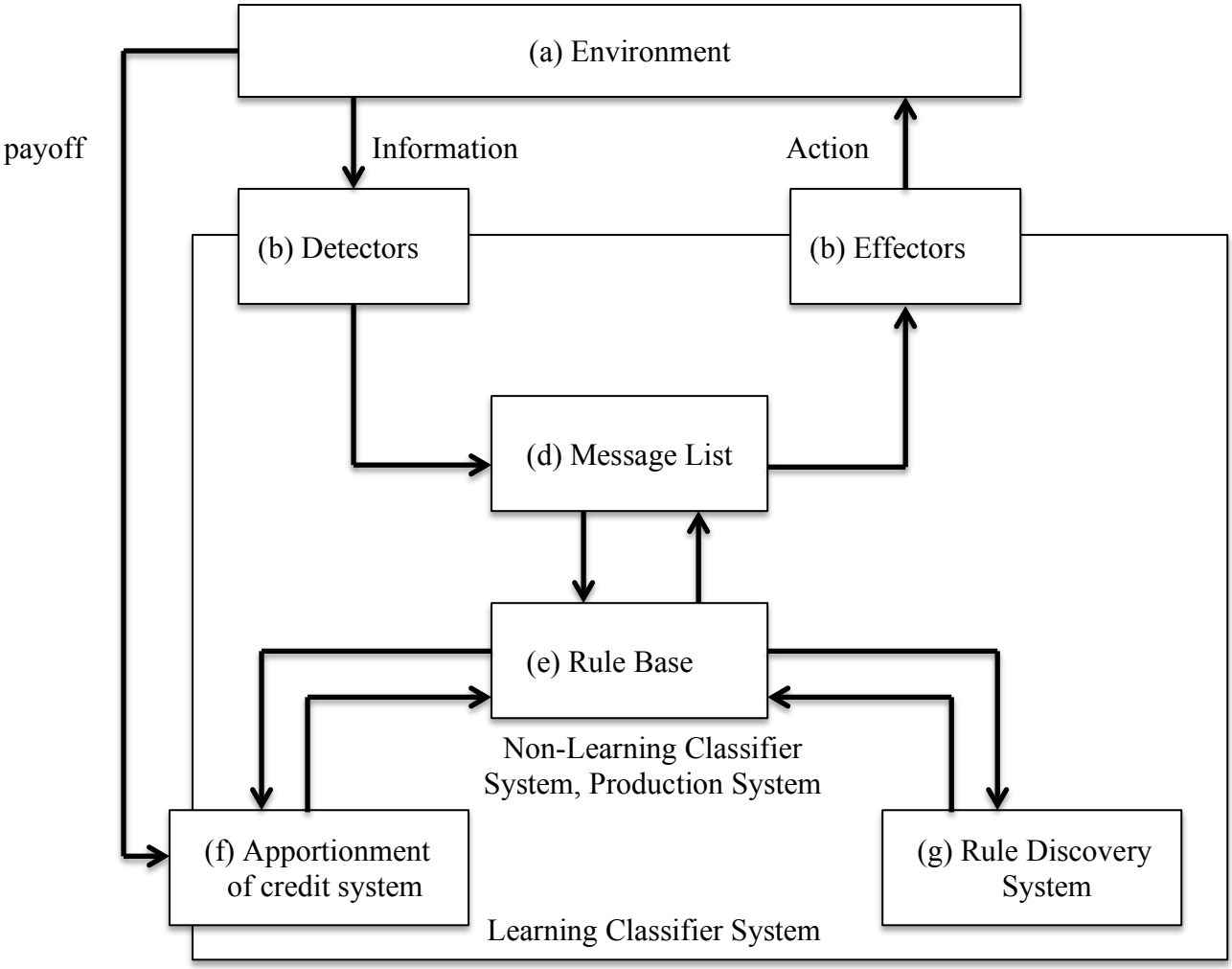


Figure 11: The structure of Michigan style Learning classifier system according to Geyer-Schulz [12]

Evolutionary strategy: Evolutionary strategies are heuristic optimization approaches. They construct a special kind of evolutionary algorithms. Here each individual is represented by a vector of real numbers. Most common operations are mutation and selection. Mutation uses the normal distribution to change a specific number in the individual (e.g., "in the Shrink mutation, this operator adds a random number taken from a Gaussian distribution with mean equal to the original value of each decision variable characterizing the entry parent vector" [97]). The parameters for the normal distribution are chosen by self-adaptation [13][14][15]. All other aspects remain the same as a traditional evolutionary algorithm.

"Mutation is performed by adding a normally distributed random value to each vector component.

Selection is deterministic and only based on the fitness rankings, not on the actual fitness values. The resulting algorithm is therefore invariant with respect to monotonic transformations of the objective function. The simplest evolution strategy operates on a population of size two: the current point (parent) and the

result of its mutation. Only if the mutant's fitness is at least as good as the parent one, it becomes the parent of the next generation. Otherwise the mutant is disregarded. This is a *(1 + 1)-ES*. More generally, λ mutants can be generated and compete with the parent, called *(1 + λ)-ES*. In (1 , λ)-ES the best mutant becomes the parent of the next generation while the current parent is always disregarded." [37]

"(1 + 1)-ES: The population only consists of a single individual which is reproduced. From the elder and the offspring, the better individual will survive and form the next population." [11]

"(μ + 1)-ES: Here, the population contains μ individuals from which one is drawn randomly. This individual is reproduced from the joint set of its offspring and the current population, the least fit individual is removed." [11]

"(μ + λ)-ES: Using the reproduction operations, from μ parent individuals $\lambda \geq \mu$ offspring are created. From the joint set of offspring and parents, only the μ fittest ones are kept." [11]

"(μ,λ)-ES: In (μ,λ) Evolution Strategies, introduced by Schwefel [38], again $\lambda \geq \mu$ children are created from μ parents. The parents are subsequently deleted and from the λ offspring individuals, only the μ fittest are retained. [38] [39]" [11]

"(μ/ρ,λ)-ES: Evolution Strategies named (μ/ρ, λ) are basically (μ, λ) strategies. The additional parameter ρ is added, denoting the number of parent individuals of one offspring. Normally, we only use mutation (ρ = 1). If recombination is also used as in other evolutionary algorithms, ρ = 2 holds. A special case of (μ/ρ,λ) algorithms is the (μ/μ,λ) Evolution Strategy [40]." [11]

"(μ/ρ + λ)-ES: Analogously to (μ/ρ, λ)-Evolution Strategies, the (μ/ρ + λ)-Evolution Strategies are (μ, λ) approaches where ρ denotes the number of parents of an offspring individual." [11]

"$(\mu',\lambda'(\mu,\lambda)^{\gamma})$-ES: Geyer et al. [41][42][43] have developed nested Evolution Strategies where λ' offspring are created and isolated for γ generations from a population of the size μ'. In each of the γ generations, λ children are created from which the fittest μ are passed on to the next generation. After the γ generations, the best individuals from each of the γ isolated solution candidates propagated back to the top-level population, i.e., selected. Then, the cycle starts again with $\lambda`$ new child individuals. This nested Evolution Strategy can be more efficient than the other approaches when applied to complex multimodal fitness environments [44][43]. " [11]

Evolutionary programming [16] [17] [18] [19] [20]: Here, recombination cannot be applied. The only operations that can be applied are mutation and

selection.

Genetic programming [11]: To apply genetic programming, programs and algorithms are represented as trees. In this case gemones are represented as tree data structures instead of chromosomes, that are used in the usual genetic algorithms [21][53]. A tree may stand for a mathematical expression or a decision tree [54] (very similar structure to algorithms and programs).

Genetic Programming directly evolves individuals (tree data structures) by applying genetic algorithms.

In [21][22] we find the basic concept of genetic programming studied by Koza. In this concept, each individual, or a chromosome may represent a different algorithmic strategy. Koza has stated many advanced stages of genetic programming including learning of Boolean functions [47][48], Artificial Ant problem [49][50][51], and symbolic regression [47][52].

Other metaheuristics of optimization problems: Combinations of combinatorial optimization algorithms and improving algorithms, in order to overcome local optima. This provides a general set of possible solutions. Metaheuristic approaches have proved their effectivity in solving complex problems especially those depending on combinatorial algorithms. [23]

Heuristics [56] are algorithms dedicated to solve specific problems faster than other ones that may produce non-exact solutions "approximate solutions" for the problems. In other words, the solution of a heuristic algorithm might not be the best one. It is, however, an acceptable one, because it is sufficient to fulfill criterion of time to find the solution and also quality of the solution. Metaheuristic is an advanced class of heuristics. It may provide solutions even with incomplete input or resources. [57] A metaheuristics is a universal principle that can be applied to a broad class of problems, in contrast to a dedicated heuristics which is bound to a specific problem.

Metaheuristics (as e.g., greedy search) have the tendency to be stuck in local optima. Advanced meta heuristics try to overcome this deficiency using various techniques.

In [24], several ways are introduced to combine exact and metaheuristic algorithms for combinatorial optimization, See Figure 12.

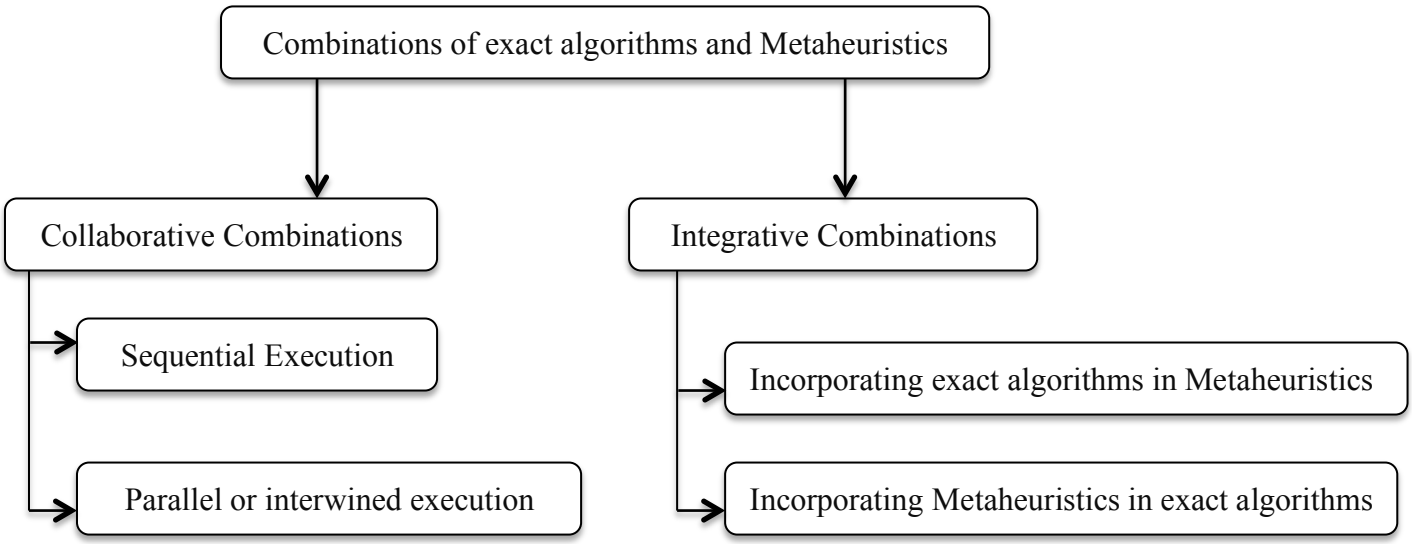


Figure 12: Major Classification of exact/metaheuristic combinations [24]

Figure 12 shows the ways of combining exact and metaheuristic algorithms. In collaborative combinations, two algorithms might exchange information, but they are applied separated from each other. In integrative combinations, one algorithm might become a part of another algorithm.

In collaborative combinations, the algorithms may execute in a sequential way, parallel or interwined [24]:

- Sequential Execution: The exact algorithm executed before the metaheuristic algorithm or vice versa. The processing of the former may be a precondition of processing the latter. The processing of the latter may, however, be a post condition of processing the former.

- Parallel Execution or interwined execution: The algorithms are executed in parallel or interwined way.

In integrative combinations, the algorithms might be integrated in two different ways [24]:

- Integrating exact algorithms with metaheuristics: This may include exactly solving relaxed problems, exactly searching large neighborhoods, merging algorithms, or exact algorithms as decoders.

- Integrating metaheuristics into exact algorithms: This may include metaheuristics for strategic guidance of exact search, metaheuristics for column and cut generation, etc.

Examples of metaheuristics include simulated Annealing [25], iterated local

search [26], evolutionary algorithms [27], etc.

Simulated Annealing:

is a method inspired from nature. The main concept of Simulated Annealing is that at "high temperatures" large changes are possible, allowing to escape from local minima. Later, good solutions are "frozen" at "lower temperatures". This concept is influenced from the annealing process of metals where at higher temperatures the molecules can move more freely while at lower ones they are fixed to crystal structures. The annealing process supports avoiding that molecules would take too early positions that later would disallow a perfect crystalline structure. In Simulated Annealing this is achieved by using a controlled cooling down process.

A well-known application is the traveling salesman problem. The traveling salesman problem aims at visiting a number of cities with the shortest time. This means with shortest possible path. Arranging the cities on the way of the traveller could be done by simulated Annealing. [28]

Iterated local search: is an algorithm that aims to enhance local search by iteration [29]. Local search tries to find an optimal solution. It starts with an initial solution (current solution). Then a transformation function is applied on the current solution. This transformation results in a new solution. If the new solution is better than the current one, it replaces the current one [58]. The iterated local search procedure is defined as follows:

"Procedure *Iterated Local Search*
s_0 = GenerateInitialSolution
s^* = LocalSearch (s_0)
repeat
s` = Perturbation (s*, history)
s*` = LocalSearch (s`)
s* = AcceptanceCriterion (s*, s*`,history)
until termination condition met
end" [29]

For an input s_0, the local search produces s*

The intermediate state s` results in from s* by applying Perturbation (change)

For an input s`, the local search produces s*`

If s*` passes the acceptance test, it is assigned to s*.

“The potential power of iterated local search lies in its biased sampling of the set of local optima. The efficiency of this sampling depends both on the kinds of perturbations and on the acceptance criteria.
The main drawback of local descent is that it gets trapped in local optima that are significantly worse than the global optimum. Much like simulated annealing, ILS escapes from local optima by applying perturbations to the current local minimum. Generally, the local search should not be able to undo the perturbation, otherwise one will fall back into the local optimum just visited. Surprisingly often, a random move in a neighborhood of higher order than the one used by the local search algorithm can achieve this and will lead to a satisfactory algorithm. Still better results can be obtained if the perturbations take into account properties of the problem and are well matched to the local search algorithm.
If the perturbation is too strong, ILS may behave like a random restart, so better solutions will only be found with a very low probability. On the other hand, if the perturbation is too small, the local search will often fall back into the local optimum just visited and the diversification of the search space will be very limited”. [29]

2.3.3 The Knapsack problem

The previous section provided an introduction to solve algorithmic problems. In this section, we go through the knapsack problem, and its probable solutions. Solving a knapsack problem constitutes an integral part of achieving the objectives covered in this thesis.

The knapsack problem is an NP-hard combinatorial problem. There is a set of items, where each item has a benefit and a weight. A subset of items should be selected, so that the sum of their benefits is maximized, and the capacity of the knapsack is not exceeded. There are several types of knapsack problems [30]:

- 0-1 Knapsack problem: There is a set of items. We want to put the items into a knapsack of capacity W. We should pick a set of mutually different items, so that the total value is maximized and the capacity of the knapsack is not exceeded.

- Bounded Knapsack problem: Same as 0-1 knapsack problem. However, we can select more than one instance from each item. The number of selected instances is limited by a certain bound specified for each item.

- Multiple knapsack problem: Same as 0-1 knapsack problem. However, here we have more than one knapsack. Each knapsack has a capacity. The knapsacks should be filled with the items, so that the total value is maximized, and the capacity of each knapsack is not exceeded.

- Multiple-choice knapsack problem: Same as 0-1 knapsack problem. However, the items should be chosen from different disjoint classes. Only one item is chosen from each class.

- Multi-dimensional multiple choice knapsack problem: Same as multiple-choice knapsack problem. However, the knapsack may have a vector of capacities. Each capacity represents the availability of different resources (dimensions) that the knapsack provides. The weight of each item is represented by a vector. Each weight in the vector reflects the weight of a unique resource. When a set of items is chosen by solving the knapsack problem, the sum of weights for a specific resource should not exceed the resource capacity provided by the knapsack.

In our approach, we use the multi-dimensional multiple choice knapsack problem to illustrate the optimization problem we are working on.

2.3.3.1 Solution techniques for the Knapsack problem

There are many approaches for solving the different knapsack problems. As we are considering in our approach the multi-dimensional multiple choice knapsack problem, an optimal solution can be found using Dynamic Programming, however in exponential time. Therefore some heuristics are needed. We describe here some of the most common solutions [31] [32]:

A modified GLS algorithm: is a metaheuristic algorithm that applies conditions on the local search process to overcome the local optima. GLS stands for guided local search. In the usual search process, all possibilities might be traversed to find a required solution. The traversing process has a minimum and maximum number of steps. A local optima is the optimal solution in a neighborhood set. “A neighborhood set N(x) returns a set of neighbouring solutions to x. the neighbourhood N(x) is simply the set of all solutions resulting from changing the value of one variable from true to false or vice-versa”. [98] In GLS, two phases are applied. In the first one, the elements with the greatest value in each class are chosen. If this results in a solution (the weight condition is true), then the algorithm has reached a solution. If, however, this did not result in a solution, then the second phase is applied. Here, the class of the element of the greatest weight in the unfeasible solution is selected. This element is replaced by another element from the class. If the solution is still unfeasible, then the element with the smallest weight in the class is selected. The second phase is applied for a fixed number of times. A feasible solution might also not be found after applying the second phase. [31]

> In case no feasible solution results from the first phase of the GLS algorithm, chosen items are swaped with other items from their classes, until a feasible solution is reached. This process, called a complementary

local search, stops according to a defined condition. [31]

- **A derived algorithm using penalties and normal transformations (Der-Alg):** The DER algorithm is derived from the GLS algorithm. If a feasible solution could not be found after a number of iterations, a penalty is perfomed to transform the benefits of the objective function. When a feasible solution is found, a normalization process is applied to get the original values of the benefits. [31]

- **An evolutionary algorithm:** The algorithm starts by initializing a population, according to some selected criteria. The population is then evaluated. Mutation is done to enhance the objective function. The process of mutation can be repeated until a certain condition is satisfied. [32]

" Begin
    t ⟵ 0
    initialize P(t)
    evaluate P(t)
    while (not termination-condition) do
        begin
            t ⟵ t + 1
            select P(t) from P(t-1)
            mutate P(t)
            evaluate P(t)
        end
end" [32]

P(t) is a generation, which consists of a set of chromosomes.

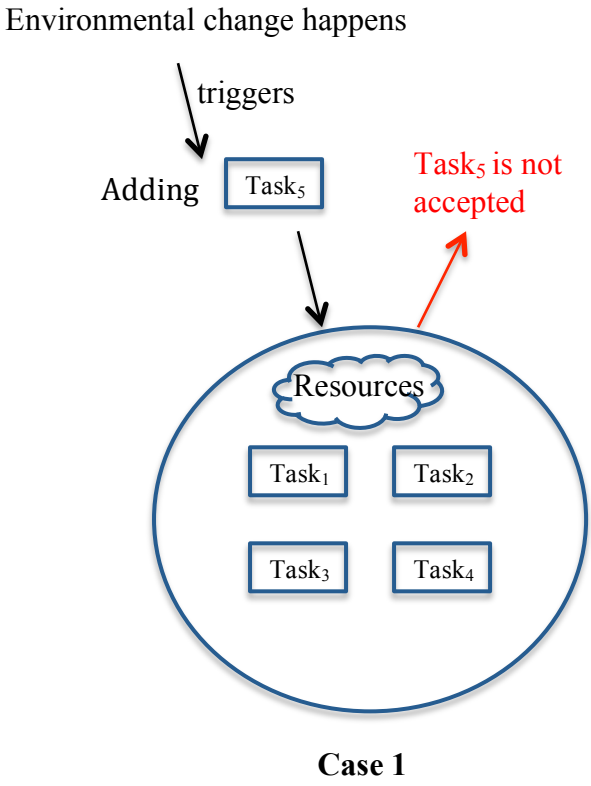
The complexity of the algorithm depend on the termination condition. If all states are to be traversed, then the complexity is exponential.

In our approach, we use a genetic algorithm inspired from [32] to solve the knapsack problem. The reason is that this approach can provide the whole solution (individual) at once if available. This allows to use required parameters of the individual elements in order to calculate the parameters of other elements.

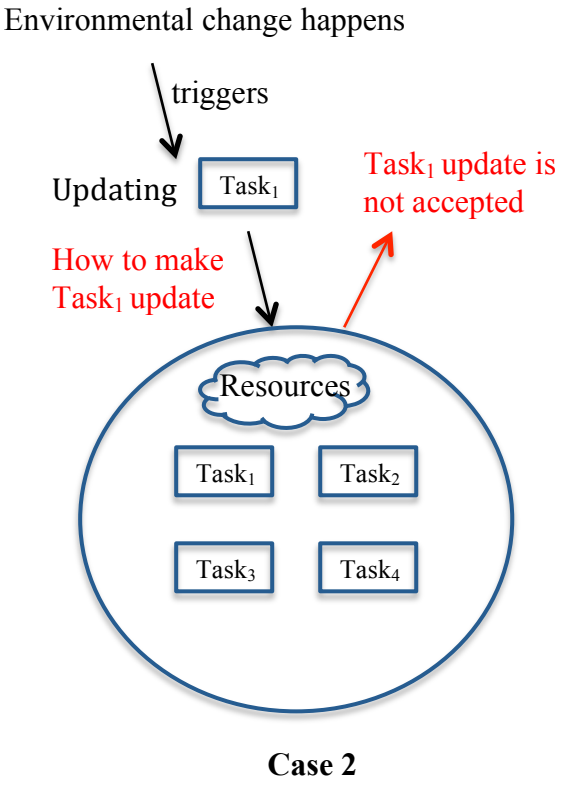
Most important for our application is the fact that it is an "Anytime Algorithm" in the sense that at any time the current valid solution of the algorithm can be used. This solution may be far away from an optimal one. However, if the initial population is a valid solution it is guaranteed that at any time a valid solution can be provided.

Chapter 3 Basic Concept

Let us assume a system consisting of a computing node. The node is running a real-time



Case 1



Case 2

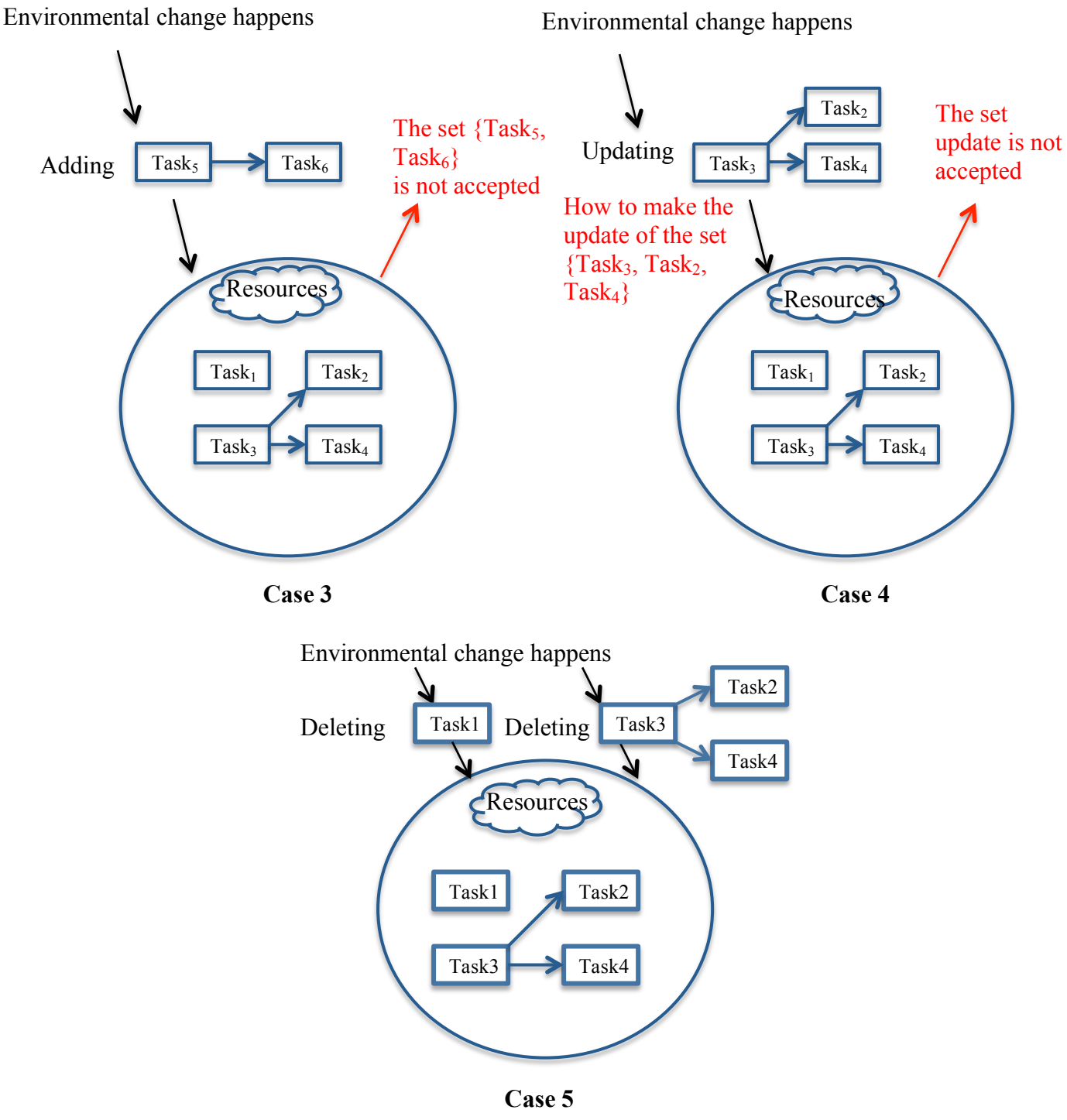


Figure 13: Problem Description

operating system. Requests that arrive at run time could be

e.g., adding new tasks. These tasks are either accepted or not accepted by the system. The challenge we are facing here is how to transform the system into a highly dynamic self-adaptive real-time system. This means, how to increase the

ability of the system to accept as much requests as possible, in order for the system to adapt itself to environmental changes. Here, several kinds of requests are defined to map the real life scenarios of adaptation. Figure 13 points out the challenge we are facing.

In case 1, a new task should be added to the system as a result of some environmental change. Adding it may cause a non-feasible schedule. As a consequence, the system cannot adapt itself to the current state of the environment. The problem we have to solve here is providing sufficient processor capacity to allow adding the new task using the minimum possible amount of resources.

In case 2, a running task should be updated as a result of some environmental change. Updating it may cause in a non-feasible schedule. As a consequence, the system cannot adapt itself to the current state of the environment. The problem we have to solve here is first to define the task update in terms of the properties of real-time tasks, and second to provide sufficient processor capacity to allow the new update using the minimum possible amount of resources.

In case 3, a new set of dependent tasks should be added to the system as a result of some environmental change. Adding it may cause a non-feasible schedule. As a consequence, the system cannot adapt itself to the current state of the environment. The problem we have to solve here is providing sufficient processor capacity to allow adding the new set using the minimum possible amount of resources.

In case 4, a running set of dependent tasks should be updated as a result of some environmental change. Updating it may cause a non-feasible schedule. As a consequence, the system cannot adapt itself to the current state of the environment. The problem we have to solve here is first to define the task update in terms of the properties of real-time tasks, and second to provide sufficient processor capacity to allow the new update using the minimum possible amount of resources.

In case 5, an executing task and/or a set of dependent tasks should be deleted from the system as a result of some environmental change.

In our approach, we even discuss the solution, when a mixture of the introduced requests happens at once, or when several requests of the same type arrive at once.

For solving any of the described problems, we have developed a new model for real-time tasks, called cells. A cell has the same structure of a real-time task, but with additional data. The additional data is used to classify tasks into classes. A class defines a set of tasks, which accomplish the same function, but with

different cost and time characteristics. Each task that belongs to a class is called a variant. Classes can be updated, when specific environmental circumstances change.
Whenever a set of requests arrive to the system, we select the best variants that belong to the classes of currently executing variants, and to the triggered requests. Before the selecting process, time characteristics of the participants in the acceptance test might be calculated according to the type of requests. The selection process is done by solving a multi-dimensional multiple choice knapsack problem. The goal is to meet the deadlines according to the specified schedulability test, and to minimize the costs. The optimization problem is formulated as follows:

Minimize $\sum_{i=1}^{Num} Cost_i$
Subject to: Us + Up $\leq$ 1
$\sum_{i=1}^{Num} Cost_i \leq Cost_{total}$

- Num is the number of participating classes in addition to the central cell "The Engine-Cell", which leads the adaptation process.
- $Cost_i$ is the cost of the i_{th} variant.
- Up is the periodic utilization.
- Us is the aperiodic utilization.

The basic strategy for solving the problem can be summarized as follows:
We assume an initial system consisting of a set of cells where for each cell, there may be a collection of variants. Initially, the system only includes system cells and Engine-cell. We assume that the initial system is optimal in the sense as describe above, i.e. precise solution of the knapsack problem is used as initial configuration. As this is decided offline, the knapsack problem can be solved precisely.
Whenever we have to handle an adaptation, we assume the present configuration to be optimal, or close to optimal.
To find a solution we use an evolutionary algorithm to solve the knapsack problem. This is an iterative approach, which needs an initial configuration. If we use a configuration which selects from each cell class the variant with the lowest demand on processor power then we have an easy test, whether a solution is possible. If this configuration satisfies the constraints (related to time and costs) then a solution is possible, otherwise not. To find out the initial configuration, we construct a generation of individuals. Each individual represents a configuration. If a solution is possible using one of these individuals, then we try to enhance it by finding a solution with better quality (lower costs) inside the same generation (It is not necessarily optimal as a precise solution of the knapsack problem over the entire task set after adaptation might result in even lower global costs).

If a solution is not found, we iterate over generations. Each generation results from the previous one by applying mutation.
We can install a time bound on the iterations so that the algorithm will respond in bounded time with a solution which might be different from the achievable optimum (This anytime algorithm is a heuristic).

Chapter 4 Related Work

In this chapter, we describe the most related works we found in the area of organic programming and self-adaptable systems.

In [2] a new programming model is introduced. This model is called organic programming. It aims to overcome limitations of the traditional programming models as e.g., the object oriented programming. By definition, “Organic programming is centered around the concept of a true thing. A thing behaves like objects do in our real world, or cells do in an organism”. The idea behind the approach is to have a system that is able to grow and evolve continuously. For the previously defined model, a specific software representation “Ercatons” was developed. An Ercaton is a cell with specified properties, and a cell is a thing that can be active and behave on its own. This model however was made for the traditional software systems, and did not discuss any real-time aspects. In our approach, we concentrate on having a system consisting of cells with defined properties that enable for self-adaptability in real-time. In other words, the tasks that should be fulfilled by these cells, should also meet specific deadlines.

In [66] we find a summarized description for the state of the art in terms of modeling dimensions, research challenges, and requirements of self-adaptive systems. A self-adapting system has the following dimensions: “Goals: Evolution, Flexibility, Duration, Multiplicity, Dependency, Change, Source, Type, Frequency, Anticipation. Mechanisms: Type, Autonomy, Organization, Scope, Duration, Timeliness, Triggering. Effects: Criticality, Predictability, Overhead, Resilience”. [66]
The goals of the system could be either static for the lifetime of the system, or changeable according to the different adaptability aspects. In our approach, the system may have different tasks that can change at run time.
- The evolution of a system is measured by the evolution of its goals. Our approach may have static goals, which can be updated over time, as well as goals, which may change according to the adaptation scenarios.
- Flexibility of a system is measured by the uncertainty of goals. A goal that must be fulfilled is an inevitable goal. But a goal that might be fulfilled is an evitable goal. Our approach has also rigid goals.
- Duration of a goal refers to the time over which a goal is valid. Our approach may or may not have continual (periodic) goals. The multiplicity dimension refers to the number of goals. In our approach, there might exist many goals.
- Dependency reflects how the goals are related to each other. In our approach, dependencies do not exist. We want to find a solution that minimizes the costs, and meets deadlines. Costs and execution time may influence each other.

Change is the event that causes the adaptation process. Types and frequencies of changes may differ. In our approach, the events depend on the system, where we apply the developed algorithm.
- The source dimension determines the origin of the change. It could come from inside the system or outside. In our case, there is no restriction on the events.
- Type of the change could be functional, non functional, or technological. A functional change results from changing the functions of the system. A non-functional change results from other perspectives, as for example, a change in the quality properties of the system. A technological change results from the system version, or similar technological aspects. In our approach, there is no restriction on this issue.
- The frequency, by which a change happens, can be high or low. Our approach defines this aspect according to the resources of the system. It, however, assumes an upper bound for the frequency of the changes that lead to requests arriving to the system.
- Anticipation determines if the change that causes the adaptation can be predicted. In our approach, it is foreseeable, but can change over time. This means that the actions, to which the system must adapt itself are dedicated to specific changes in the environment (foreseeable). These changes are normally measurable. Measuring the changes is not in the scope of the thesis. However, as the system grows, the expected changes may be updated by considering new parameters in the environment, and new actions may be developed to deal with the newly defined changes. This can happen at run time.

Mechanisms of adaptability summarize how the system can react to changes, in terms of space and time required. Under this category of dimensions, we have several aspects:
- the type of an adaptation can be structural or parameterized.
In our approach, there is no such restriction. A structural adaptation might take place, as we allow for structure and behavior changes of tasks, which means that components and resources of the system might also differ over time.
- Autonomy identifies the degree of getting outside actions in applying the adaptation. The approach we define, may act by decisions taken automatically or by other parties as humans, system specialists or system administrators.
- The organization dimension captures whether the adaptation is done by a centralized or distributed components. In our approach, the adaptation is done by a central component.
- The scope of adaptation could be local or global, in reflecting the results and effects of adaptation. The scope of our approach can be local or global according to the system.
- Duration of adaptation points out the time over which the adaptation process takes place. In our approach, the lifetime of the system differs from one system to another. The duration of adaptation is influenced by execution time of the Engine-cell.

- Timeliness captures if the time required for self-adapting is guaranteed (deadlines are met). In our approach each task has a hard deadline. We also determine the execution time of EC in our calculation. This results in a guaranteed finishing time of the adaptation.
- The triggering dimension identifies the adaptation if it is time-triggered, or event-triggered. In our approach, there are no restrictions on this aspect, as we do not discuss the causes of the triggers.

The set of dimensions, effects deals with the results of adaptation, as e.g., the overhead. In our approach, missing a deadline may confirm the failure of the system.
- Criticality of a system captures the impact of a failure for the self-adapting procedure. This dimension can range from harmless, mission-critical, to safety critical. In our approach, we assume a safety critical system, as we have hard deadline tasks. We also determine the execution time of EC in our calculation. This results in a guaranteed finishing time of the adaptation.
- Predictability identifies whether the consequences of self-adaptability could be predictable in value and time. This property could be deterministic, or non deterministic. In our approach, it is deterministic.
- Overhead defines the negative influence on system performance, when self-adaptation is applied. In our approach, the execution time of the EC (adaptation time) influences system performance.
- Resilience is the property related to persistence of service delivery, when facing challenges [68]. Our approach achieves resilience.

There are many research challenges of modeling dimensions. The challenges are concerned mainly with balancing the different aspects for achieving an optimal self-adaptable system. Optimization techniques, monitoring techniques, systematic engineering may play here a significant role to overcome these challenges. In our approach, we are concerned mainly with optimizing the solution, in terms of cost (quality).

Engineering of self-adaptive systems concentrates on the control loops, where data is collected and analyzed. Based on it, decisions are taken, and actions take place. In our approach, we solve the acting part. The three other aspects are open to follow different methods and techniques, according to the type of the system. The acting part in our solution includes scheduling the real-time requests, and reconfigure the system dynamically, to ensure high acceptance percentage for the requests.
Many challenges are related to the engineering of self-adaptive systems. E.g., verification and assurance of system requirements. In our approach, we are concerned only with real-time requirements, as e.g., fulfilling dependencies, meeting deadlines, and act within the cost limitations. Fault tolerance is another challenge, which we can handle in our approach as a specific context or event,

that may trigger the self-adaptation algorithm.

Transferring the system from a stable state to another stable state when applying the self-adaptation is also a challenge. The correctness of this requirement depends on number of succeeded request adaptations. In our approach, we do not provide a method for measuring this aspect. As the aspect may differ from one system to another, and also from one scenario in the same system to another scenario (request frequency, system resources, and how they change over time plays a basic role), providing a method becomes a statistical issue.

In [67] a second roadmap for state of the art is presented. Challenges of a self-adaptive system described here are the design space, processes, decentralization of control loops, and run-time verification and validation.

- The first challenge is to understand the different alternatives that may represent designer or developer decisions. It is also to refine different dimensions contributing in establishing a design space. These dimensions are, observation, representation, control, identification, and enabling adaptation.
Observation is done for the domain knowledge, information, environment and goals. Observation can be triggered by an event, timer, or previous observation. Representation refers to the information that should be available at runtime, and how it is represented. Control is concerned with the events that trigger adaptation, and feedback loops, which control the flow of actions that may lead to a self-adaptive procedure. Identification identifies the system status at run time. Enabling adaptation refers to the methods by which an adaptation can be triggered, and to the means by which the adaptation is supported.
Discussing the details of each dimension needs normally to be done for a specific system, where the adaptation approach is applied. In our approach, we have developed a general strategy that applies for different kinds of real-time systems. However, the components by which an adaptation is supported are defined by a dedicated Engine-cell and an array of different configurations for each task in the system, in addition to the external components that provide a source for updating and developing the different configurations.
The possible challenges regarding the design space construct a base for the different design decisions, bridging the gap between design and implementation. In our case, we have an abstract implementing component, which fits as a reusable component for different systems. The interaction between control loops and self-adaptation mechanisms is also a challenge. In our case, the self-adapting mechanism concentrates on acting, and leaves the issues of collecting data, analyzing and taking decision to be determined according to the chosen system, where the approach is applied.

- The second challenge is concerned with understanding the nature, goals, and lifecycle of the system. How the processes evolve, and what are the factors

influencing these processes, also belong to this challenge. In addition, it is important to formalize the process in a way, which identifies the general and reusable components. In [67], a comparison between the basics for traditional software processes, and self-adaptive processes is described. The first one is illustrated in [69] by the traditional approach to corrective maintenance, and the second in [70,71] by the automatic workaround approach.
The traditional approach to corrective maintenance reports the problem to the developers, who will fix it, after analyzing the causes of the failure. The automatic workaround approach moves the correctness to run time by applying alternative procedures when a failure happens. In our approach, the alternative procedure might be a new request or an update request. If the causes of the failure should be analyzed, this is done by a component subsystem. Our approach may assign the analyzing task to a human (system administrator), or to a subsystem. This depends on the sort of system, where the approach is applied. These issues, however, are not our concern in the presented thesis.
In the workaround approach, the recovering methods are developed at the design phase. In our approach, this can be done at runtime, as new solutions can always be added online. In the workaround approach, if a recovering method does not exist, a report is sent to the developers to analyze the failure according to the traditional systems. In our approach, we do not assume a fault tolerant system. If, however, a fault happened in the system, then we assume that the fault will trigger a request, and a solution how to handle the request has already been developed. If a recovering method does not exist, a report can be sent to the developers to analyze the failure according to the traditional systems, the same way this is treated by the workaround approach. Recovering methods, however, is not our concern in the presented thesis.
Representing the life cycle of a self-adaptive system constructs also a challenge. In [72], a framework for describing self-adaptive systems is presented. The approach is called SPEM, stands for “Software & Systems Process Engineering Metamodel Specification”. SPEM includes elements for reusable components, process models, activities and artifacts. It is used, however, for concrete systems.

- The third challenge is concerned with decentralization of control loops. Control loops include normally four activities: monitoring, analyzing, planning, and executing. We are mainly concentrating on the fourth activity in our approach. The other activities are planned according to the type of the system. In general, controlling a system could be by a centralized or decentralized manner. There are many approaches for centralized fashion in controlling a system, as e.g., [73, 74, 75]. The trends for decentralizing the control of self-adaptive systems are included in several other approaches, as e.g., [76,77,78,79,80]. Scalability in a distributed system is better than in a centralized system, because in the latter, information has to be collected on one node. The self-adapting component is central in our approach, as network

reliability concerning time and trustworthy is a main concern in real-time systems. If the algorithm, however, is distributed on a multi-processor system, this might be a promising approach. We still, however, need to collect the information on one node, and the solution would cost more resources. For this reason, we got a better trade-off by assuming to apply our approach on a uni-processor system.

- The forth challenge is the verification and validation of the system: usual issues that should be discussed here are the aspects that should be validated, the point of time to apply the validation, and the system requirements and properties that should be verified and validated. In our approach, verification is done for requirements of real-time systems, apart from the context of the system.

Chapter 5 Problem Description and Solution

Here we present for each different case an assumption, problem, and solution. We strictly follow the concept of transactions. If a solution is found after applying specific elementary operations, the system state is updated. If a solution is not found, the system goes back to the previous state.

In each case, we put assumptions on the:

- Type of tasks: periodic, aperiodic or both together.
- Dependability between tasks
- type of the requests:

    - Adding a task: means adding a new mission that has a deadline.
    - Updating a task: means updating the structure and behavior of a task. The updated task has a deadline.
    - Adding a set of dependent tasks: means adding several new missions that should be achieved by the system in real time. Executing one or more missions depend on the results achieved by one or more other missions.
    - Updating a set of dependent tasks: means updating several new missions that should be achieved by the system in real time. Executing one or more missions depend on the results achieved by one or more other missions.

The mentioned types of requests may result from a change in the internal or external parameters of the system. Internal parameters might be e.g., the clock of the system, the temperature of the system, the resources used by the system, etc. The external parameters might be environmental parameters, e.g., the wind, the weather, the influence of other contiguous systems, etc.

The cases are described from the simplest case to the most complex one. In 5.1, we assume a periodic task environment. The requests can add or delete tasks. There are no dependencies between tasks.
In 5.2, we assume a mixed periodic and aperiodic task environment. The requests can add or delete tasks. There are no dependencies between tasks.
In 5.3, we assume a mixed periodic and aperiodic task environment. The requests can add, delete or update tasks. There are no dependencies between tasks.
In 5.4, we assume a mixed periodic and aperiodic task environment. The requests can add or delete tasks or sets of dependent tasks. The requests can also update tasks in the system. The dependencies may only exist between aperiodic tasks.

In 5.5, we assume a mixed periodic and aperiodic task environment. The requests can add or delete tasks or sets of dependent tasks. The requests can also update tasks or sets of dependent tasks in the system. The dependencies may only exist between aperiodic tasks.

In each of the defined cases, we follow the same adaptation strategy. We assume that we have a local node. The Engine-Cell resides on the local node.
The system tasks, and tasks that are triggered, have to be executed on the local node. In order to be able to run the adaptation algorithm, additional variables are added to each task. This comes up with the concept of real-time cells. A cell is a task that is able to change its structure and behavior at run time, to allow adaptations in real-time. The change is decided by the Engine-Cell. The Engine-Cell is a central component. It runs the adaptation algorithm on a two dimensional array. Each column stands for a class of cells which all share the same principal functionality. Each cell in the column is a variant, which accomplish the same task, but with different cost and time characteristics. The Engine-Cell runs an adaptation algorithm to choose a best combination of variants that allow to accept the newly arrived requests.

We also assume a remote node that is dedicated for installing variant updates, and newly deployed cells. An update updates a cell according to a predefined change in the internal or external parameters. A newly deployed cell may define a totally new task (adds a column to the array), or a variant for a predefined cell in the array.
In each execution of the Engine-Cell, the same order of steps is followed. First, newly deployed cells are imported from the remote node. They are added to the above mentioned array using a filter procedure to keep the upper bounds of the array dimensions preserved. Then, the newly arrived requests are stored in a buffer. Their types differ from one assumption to another. After that, the quality cost that the system provides is calculated. This cost function is defined to be inversely proportional to quality. Besides the feasibilty of the obtained solution, the quality cost is an important factor in deciding to accept the requests or not. Then an intermediate array is constructed. It includes executing variants, the Engine-Cell, and the newly arrived requests. New time characteristics are calculated for each variant in the array, according to its type and the type of the requests. After that, a knapsack problem on the constructed array is solved. The goal is to select one variant from each column of the array, so that a schedulability test becomes successful for the selected variants, and resulting cost is minimized. In case a solution is found the selected variants are activated (installed in the memory) if they are new. The properties of the Engine-Cell are updated. In case no solution could be found, a notification is sent to the system administrator, and the system stays in its initial state (the entire set of transactions handled by one activation of the EC).

Of course the history of versioning (the log) have to be maintained. But this is out of scope of the objective of the thesis. Management of versions, recovery, and related questions are solved by dedicated versioning systems. Such systems run more or less orthogonal to what we are considering. The only connection is that a versioning system may deliver a new version by means of an update request to the system and the system has to report to the versioning system the success or failure of the update process. We can assume that such a system exists.

Underlying Data Structure and Initialization

The solution we provide applies the organic programming concept that is defined in [2] to real-time applications. "The organic programming is centered around the concept of a true thing. A thing behaves like objects do in our real world or cells do in an organism" [2]. In this sense, our solution follows the organic behavior of any real world entity. The behavior states the ability to change the structure and/or behavior of the object in order to adapt itself to environmental changes. For this reason, we define here a new concept in real-time systems, the cell. A real-time cell is a system component dedicated to fulfill a task in real time and has the ability to change its structure and behavior at run time to adapt itself to the environmental changes. In other words, the real-time cell inherits the properties of a task, but is characterized with additional properties that enable the organic behavior of the system.
We consider two types of real-time cells in our approach, the controlling real-time cells, and the controlled real-time cells. The first one should be able to have control over the second one. We define the Engine-cell to be a cell of the first type. It exists in the system before it starts to run. This cell is abbreviated as EC. Any other cell that arrives to the system while it is running belongs to the second type, and abbreviated as RTC.
A real-time cell becomes active when it is accepted by the system for execution. Thus, the Engine-cell is called an Active Engine-cell (AEC) once it is activated, and a Non-active Engine-cell (NEC) before it is activated. Also, any RTC is called a non-active real-time cell (NRTC) before it is accepted for execution and an active real-time cell (ARTC) once it´s accepted for execution. Figure 14 shows the different shapes which are used in this thesis to illustrate the different states of the cells.

Figure 14: The Different States of Cells

The Engine-cell is first activated with a relative deadline equal to its predefined initial period. It is treated here as a periodic cell. The Engine-cell is responsible for the adaptation process. To achieve this purpose, each of the Engine-cell, and the RTCs should have a set of properties, in addition to the properties they inherit from the normal real-time tasks. These properties are defined for the Engine-cell as follows:

1- EC-ID: is a unique ID that differentiates the Engine-cell from the other cells in the system.

2- WorstCaseExecutionTime ($WCET_{EC}$): is the execution time of the AEC. This parameter is recalculated each time the AEC executes, as it depends on other parameters that may change by the execution of the AEC, as e.g., the number of ARTCs in the system (NumOfARTCs).

3- WorstCasePeriod (WCT_{EC}): is the period of the AEC.

4- Hyperperiod: is the hyperperiod of the currently accepted periodic ARTCs. The next point in time where a hyperperiod completes execution is abbreviated as NHP.

   At the start of the system, the hyperperiod is calculated as the least common multiple of the periods of periodic ARTCs that initially might exist at the system startup. The resulting value is set as initial value for the AEC's period. We examine the total utilization (AEC and ARTCs). If it is smaller or equal to 1, we have found the shortest possible period for AEC (which at the same time by definition is the hyperperiod). If the total utilization is beyond 1 then the hyperperiod has to be extended by a harmonic multiple until the total utilization is no longer beyond 1.
   If the resulting utilization of the ARTCs is 1, the system reports that the AEC should start at some later point of time. In this case, no adaptations are possible until the overall RTC utilization goes below 1 or equal to 1 due to ARTCs ending their execution.

5- NumOfARTCs: is the number of the current ARTCs in the system. Increasing this number is done by the AEC when an RTC is accepted for executing. Decreasing this number is done by the kernel as part of the deletion process, whenever an RTC is deleted or ends its execution.

6- Cost: is the cost that the active Engine-cell is assumed to consume. The cost is seen as a function of quality factors.

7- Active: is a Boolean variable set to true when the system starts. Whenever the system stops executing, the Engine-cell becomes not active, and the variable is set to false. We assume that whenever this property is turned into true, this is done by the system administrator, and whenever it is turned into false, then this is done by the kernel as part of the deletion process.
The Active property of the Engine-cell might be used for statistical purposes, e.g., to know how many active cells we have in the system. In the current version of the algorithm, this property is not used.

8- RTCArray: is the data structure that holds the different RTCs in the system. Figure 15 shows the RTCArray consisting of different RTCs. Each column is called an RTClass. Each RTClass holds a number of variants, which are RTCs dedicated to fulfill the same task, with different cost and time characteristics. Time characteristics refer here to execution time of variants. All periodic variants, that belong to the same class, have the same period.
The RTCArray keeps the values of its elements even when the Engine-Cell becomes not active. The upper bounds of its dimensions can grow online, when adding new resources to the system.

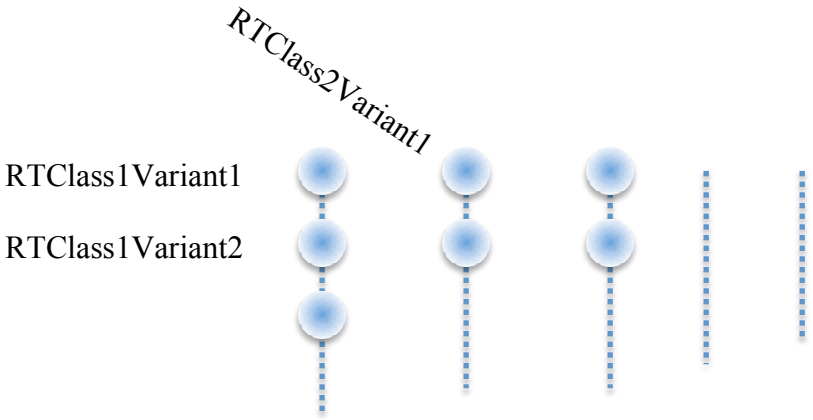


Figure 15: RTCArray

Another set of properties is defined for the RTCs:

1- RTClassID/VariantID: is a combined unique ID that differentiates an RTC from the other RTCs in the system. Here, RTClassID is the ID of a class of RTCs, which can accomplish the same task in different ways (with different time and cost characteristics). In the RTCArray, a different RTClassID is assigned to each column. The VariantID differentiates the different RTCs in the same column.

2- VariantsAllowed: is a Boolean property that expresses if the RTC, i.e., the specified variant is mandatory or not. When it is equal to true, all variants that belong to the class of respective existing or arriving RTC should be examined to select the best variant in the adaptation algorithm. If the property, however, is equal to false, the RTC is considered mandatory to be processed by the adaptation algorithm without examining additional variants of its class.

3- UpdatingPoints (UP): is a set of points in the code of the RTC routine. At these points, the RTC could be substituted by another variant from the RTCArray. This substitution has no influence on the functionality of the RTC. All variants, which have the same RTClassID, have a set of updating points with the same number of points, where each point in a specific set has a counter part point in all the other considered sets. Here, we define the natural updating point to be the release time of the next instance of a periodic task [9]. The first updating point of an aperiodic cell is its arrival time. The end of the execution of an RTC does not represent an updating point.
An x_{th} updating point is represented as UP$[x, y]$: $x \in \{0,1,2,..\}$, y is the consumed computation time up to the updating point.
When introducing the concept of updating points, we assume that an updating point always has a context switch operation. In this context switch the AEC has to replace the address of old variant to the address of the respective location of the new variant. In case of aperiodic variants, we assume the current aperiodic variant just disapears after execution if not explicitly reactivated at some later time. Under this assumption the old variant of the aperiodic RTC just disappears automatically and a potential reactivation automatically relates to the new variant.

4- $ET_{executed}$: is the time that has been spent in executing an aperiodic RTC before the last NHP. We assume that this value is always provided by the underlaying RTOS. Storing this amount could be done by the scheduler. $ET_{executed}$ is set initially to 0.

5- NextUpdatingPoint: a variable that saves the next updating point which has not been yet reached by the executed code of the RTC.

6- Triggered: is a Boolean property that reflects the status of an RTC. If it is equal to true, this means that the RTC is triggered for execution (it is not active but it has to become active at the next appropriate time). Otherwise, it is not triggered. Whenever a decision is taken about a task to be accepted or not accepted, this property turns to be false. In this case, we assume that the property is turned to false by the AEC.

7- TriggeringTime: is the time at which an RTC is triggered (chosen from the RTCArray). Here, we differentiate the arrival time from the TriggeringTime, by defining the arrival time as the time at which the cell becomes ready for execution.

8- TriggeringRange: is the range of time within which the arrival time of an RTC could be set. It starts at the triggering time.

TriggeringRange provides flexibility in choosing arrival times of requests. It is used, in case arrival times are not identical with the next point, at which the hyperperiod of periodic cells is completed (NHP).
Our goal is to set the arrival time of request equal to NHP, because at this point, we assume that all accepted periodic requests are simultaneously activated (i.e, we assume that all phases to be 0).

9- Deletion: a Boolean property, set to true if the request should be deleted. It is set to false, otherwise.

10- DeletionTime: is the time at which the RTC is to be deleted, if its Deletion property is equal to true.

11- Active: is a Boolean variable set to true when the cell is accepted for execution. Whenever a task ends execution or is deleted, this property turns to be false. We assume that whenever this property is turned into true, then this is done by the AEC, and whenever it is turned into false, then this is done by the kernel as part of the deletion process.

12- ImportanceFactor: is a number, which represents the expected importance of the RTC, regarding its use in the system. The importance increases by increasing the number. All variants that belong to a specific RTClass have the same ImportanceFactor. For simplicity reasons, we assume the ImportanceFactor of a specific RTClass could be reassigned in the idle time of the processor according to the expectations of the system administrator, or according to the changing percent by which the RTCs of the specific RTClass are used. The ability to change the ImportanceFactor enables the system to be more dynamic.
The ImportanceFactor is used to filter the RTCArray when a newly deployed RTC adds a new RTClass. The filtering process ensures that the upper bound on the number of RTClasses is not exceeded. Only most important RTClasses are kept.

13- Essential: is a Boolean property set to true, when the process of the RTC is essential for the system to operate. If, however, removing the RTC

does not have an influence on the basic operations of the system, then Essential is set to false. Changing the value of this property normally takes place in the idle time of the processor.
All variants that belong to the same class have the same Essential value. The number of essential classes is assumed to be smaller than the upper bound of the RTC classes number that can be stored in the RTCArray.

This property is necessary to differentiate the RTClasses that are not allowed to be removed from the RTCArray, during the filtering process.

14- Cost: is an abstract concept, which includes a variety of possible constituents, e.g. memory demand or provided quality like precision of computation. In the latter case cost = 0 is assigned to the highest achievable quality while lower levels get assigned higher figures.

For simplicity reasons, we assume that the cost of the various constituents in a system, which is consumed by any cell is represented by one factor "Cost". This factor is a function of several system parameters. Each parameter represents a constituent. The following cost function may serve as an example:

$$\text{Cost} = \text{f}\,(P_1, P_2, .., P_r) = w_1 * P_1 + w_2 * P_2 + .. + w_r * P_r$$

w represents the weight, and P represents the amount.

Note that processor utilization demand (i.e. time) is not included in the cost. Time is treated as constraint (deadline) in the context of real-time systems.

15- StaticParameters: is a list of static parameters used in calculating the cost of the RTC. Each parameter has a name, amount, and a weight.

Figure 16 points out the different shapes as used in this thesis to illustrate the cells according to their different status.

Figure 16: The Different States of Cells

In the following sections:

- We use the term ExpARTCs to refer to the set of current ARTCs excluding the RTCs, which belong to the deletion requests. ExpARTCs is updated whenever the adaptation algorithm takes place.

- We use the term ExpPARTCs to refer to the set of current periodic ARTCs excluding the RTCs, which belong to the deletion requests. ExpPARTCs is updated whenever the adaptation algorithm takes place.

- We use the term ExpAARTCs to refer to the set of current aperiodic ARTCs excluding the RTCs, which belong to the deletion requests. ExpAARTCs is updated whenever the adaptation algorithm takes place.

The following calculation takes part in each assumption. It can be carried out either offline or as part of the initialization when starting the system. In any case no iterative calculation is needed as the needed factor for the hyperperiod can be calculated directly by solving a simple equation.

Calculating the initial NHP:

This procedure takes place offline before system start. It will not be repeated after each period of the Engine-cell.
The period of the Engine-cell WCT_{EC} is set to the least common multiple of periods of system periodic cells. Let sum1 denote the sum of initial periodic RTCs utilizations. Initial periodic RTCs are RTCs which initially are in the system and let lcm_initial denote the least commen multiple of the respective periods. Let Us_Initial denote the server utilization to handle the aperiodic RTCs which initially are in the system with respect to their deadlines. Then, the utilization which can be spent for EC can be calculated by $WCET_{EC}/(\text{lcm\_initial} * \text{factor}) = 1 - \text{Sum1} - \text{Us\_initial}$. By resolving this equation for factor we obtain factor = $\lceil WCET_{EC}/\text{lcm\_initial} * (1 - \text{Sum1} - \text{Us\_initial}))\rceil$. Rounding up is neccessary as factor has to be a positive integer. The initial NHP is equal to lcm_initial $*$ factor.

This calculation shows a fundamental trade-off. First of all, a system which is highly utilized by its "normal" load suffers from low adaptability as only a small part of the processing power can be spent for the EC. Secondly, $WCET_{EC}$ is dependent on the parameters b and QB. This implies a trade-off between short reaction time on adaptation requests, i.e. small value of factor versus high amount of possible adaptation actions per activation of the EC , i.e. a high value of $WCET_{EC}$ due to high values of b and/or QB. In the mean the reaction time on an adaptation

request is $WCT_{EC}/2 + WCT_{EC}$ as a request is pending until NHP and then it takes one WCT_{EC} until the adaptation becomes active. It is up to the system administarator to balance offline the system by properly setting the parameters b and QB.

5.1 Pure periodic task environment, only independent tasks, potentially adding new tasks

Assumption1: Let us assume a set of hard-deadline periodic tasks that have to be scheduled with the earliest deadline first algorithm (EDF) on a uni-processor system. The system might receive new requests at run time as a result of some environmental changes. In the simplest case, the new request is adding a new task to the system, or deleting a task from the system.

Problem1: If the new requests are not going to result in a feasible schedule, then a new technique should be developed to reconfigure the system so that it can adapt the new requests at run time preserving all real-time constraints.

Solution1: Assumption1 is applied on real-time cells rather than tasks. The active Engine-cell in this case is going to run an adaptation algorithm. The algorithm is dedicated to solve Problem1, in which a new request is represented by the arrival of a new RTC, or the deletion of an RTC.

In the following, we introduce the steps of the algorithm:

1- **Gathering and filtering the newly deployed RTCs:** As newly deployed RTCs could be added at run time, the first step of the AEC is to collect the RTCs, and store them in the a WorkingRTCArray (a copy of RTCArray) following a procedure that ensures to keep the upper bound of the WorkingRTCArray dimensions preserved. The upper bound of the WorkingRTCArray dimensions is equal to the upper bound of RTCArray dimensions.
The newly deployed RTCs remains located in the WorkingRTCArray. They might be activated later, only if they become part of a request, and this request is accepted. Newly deployed RTCs enlarge the solution space when applying the adaptation algorithm, because they represent new cells or new variants of cells that can raise the adaptation possibilities according to the events that trigger adaptations or according to time and cost characteristics. Newly deployed RTCs become part of the WorkingRTCArray. Requests, however may include a set of Newly deployed RTCs or not. Newly deployed RTCs are deployed offline by parties different from those who select requests. Requests are selected online according to the environmental changes.

Let b be the upper bound of newly deployed RTCs that can arrive at this step.

Here is a description for the procedure:
For the purpose of providing predictability we restrict ourselves to fixed upper bounds in both dimensions of the WorkingRTCArray.
Let us assume that f is the upper bound of the different variants of each class in the WorkingRTCArray, and h is the upper bound of the different RTClasses that can be stored in the WorkingRTCArray, as pointed out by Figure 17. If the newly imported RTCs may cause the exceed of the upper bound of variants in a column, or the upper bound of columns, the Engine-cell preserves the upper bounds by applying a filter procedure.

Each time a set of newly deployed RTCs should be added to the WorkingRTCArray, the filter procedure classifies the arrived RTCs according to their IDs. Each RTC is treated alone. If the respective RTClassID does not exist in the array, this means that we are adding a new column. If the RTClassID exists and the VariantID does not exist, then we are adding a new variant to an existing column.
If the upper bound of any of the influenced dimensions would not be exceeded, then the RTC is added to the WorkingRTCArray.
The RTCs that are adding new columns are processed before the RTCs that are adding new variants to the columns.The reason is that adding a new column might cause the deletion of another column from the WorkingRTCArray. In this case no new variants can be added to the deleted column.

As one possible heuristic for removal of an RTClass an ImortanceFactor-based approach may be applied, as described below.
If the newly deployed RTC is adding a new RTClass and the influenced dimension is exceeded, then the RTClasses that could be chosen to be replaced by the newly arrived one are the classes that are not essential or activated. We then make a list of these classes in addition to the added one. We exclude the class with the smallest ImportanceFactor.
(Other strategies could also be used to perform removal of a class).

Note: when all classes in the array are activated or essential and their number is equal to the upper bound of classes in the WorkingRTCArray, then if the system administrator sees that the arrived RTC is more urgent than one of the active cells, the active cell can be scheduled to be deleted on the next possible point (the natural updating point for a periodic cell).

The RTClass of the newly arrived RTC takes the place of the column, which has included the active RTC.

If the upper bound will be exceeded and the RTC adds a new variant to an existing column, then a decision should be taken, which RTC to drop. The dropped RTC could be the new one as well. To make a decision, we copy the elements of the column in addition to the new RTC and put it into a vector.

There might be many heuristics for dropping variants and classes, as for example LRU [93][94], Pseudo LRU[95], or RR[96].
Arbitrary dropping can be executed in constant time while any of the mentioned heuristics may be implemented in a way that it demands not more than linear time.

Assuming that the periodic cells in each column are ordered according to utilization factor (C_i/T_i), from the smallest to the highest value, the new RTC is added to the vector, so that it preserves the order of it.

After that we examine the RTC in the vector with the highest C_i/T_i. If it does not result in a successful schedulability test together with the AEC (the AEC is considered in the test with its execution time and period that were assigned to it before it is activated), then we exclude it. However, if it results in a successful schedulability test, then we exclude the RTC with the highest cost.

The exclusion is applied on the WorkingRTCArray, if the excluded RTC belongs to the array, i.e., it is not a newly deployed RTC.

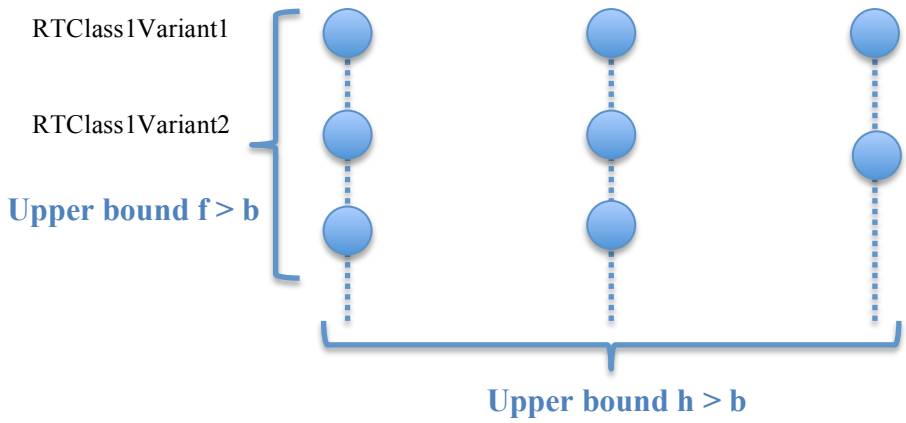


Figure 17 : RTCArray Dimensions

We assume that the b is smaller or equal to the upper bound QB of the number of requests that is allowed to arrive in each execution of the Engine-cell. The reason is that in case of all arrived newly deployed RTCs are urgent, and all classes which fill the upper bound of number of classes in the WorkingRTCArray are active, we will have a number of requests equal to b. These requests have to be stored in a buffer of an upper bound size QB.

Any other circumstances (RTCs with equal costs, or all essential classes do already exist in the WorkingRTCArray) can let the newly deployed RTCs be a part of the comparison process. The reason is that:

- In case of adding a variant: we perform arbitrary removal, when costs are equal.
- In case of adding a class: the number of essential classes is always smaller than the upper bound permitted for the classes number in the array. We assume that determining the number of essential classes belongs to the system design. Determining the number of permitted classes in the array depends on the system resources. We assume initially that system design and array dimensions fit together. Both numbers can change at run time. When system resources change, this may allow for a change in the number of essential classes.

The upper bounds of the WorkingRTCArray might change over time according to the available resources. This enables to add additional classes.

Obviously when applying the previous heuristic, it restricts the search space of the subsequent activities of the algorithm. E.g. a variant may be dropped from the WorkingRTCArray due to the fact that at the moment there is no sufficient processor capacity available. At other points of time exactly this variant might be preferred because it might provide the highest quality (i.e. lowest cost) among all respective variants. Similarly a variant being dropped due to its high costs may be the preferred one at another point of time because it consumes sufficiently low processor capacity (due to its low utilization) to enable an overall balance of the system to be executed. This variation in decision to refuse or add a variant could refer to changes in the system resources, e.g., the processor capacity.

2- **Triggering and Handling the newly arrived requests:** We assume that the requests arrival frequency is substantially smaller than the frequency of the Engine-cell ($1/WCT_{EC}$). The requests could be triggered (chosen from the WorkingRTCArray), when they are adding RTCs to the system.

The requests may also set deletions for specific ARTCs in the idle time of the processor or at the start of this step. The requests will be stored in a buffer. Triggering a request from the WorkingRTCArray turns the Triggered Boolean property into true.[1] The buffer size is constant.[2]
The arrival time of a request, which is assigned when triggering the request, depends on the next time point where the current hyperperiod is completed (NHP). This point is a point where all accepted periodic requests are simultaneously activated. I.e. all periodic tasks are assumed to have a phase of 0.

More than one request can be handled by the AEC during one execution. If the arrival time of the requests that have to be processed is not equal to the current NHP, then their TriggeringRange is examined. If TriggeringTime ≤ NHP ≤ TriggeringTime + TriggeringRange, then the arrival time is set to the NHP. Otherwise the requests, which do not satisfy the previous condition, are not accepted and deleted from the buffer. After that a notification is sent to the system administrator.

The DeletionTime of the requests that have to be deleted is set to the next natural updating point.

The requests, which satisfy the condition, proceed to the adaptation algorithm. When these requests are considered by the AEC, the buffer becomes empty again.

We assume that the number of requests that can be handled in this step is equal to the buffer size.

3- **Calculating the cost of quality factors for the system :** The parameters of the system (could be cost parameters or other parameters that play a role in creating a new request) have to be read. A part or the whole set of local parameters might represent the quality factors available by the local node. Parameters of the system should be read in each execution of the Engine-cell because they might change. This change affects the result of the adaptation. E.g., adding new resources may allow accepting a set of requests, that is not accepted with less resources.

---

[1] As the triggered requests are supposed to be a subset of the RTCArray, they are loaded by saving a reference of their IDs in the buffer. Here, each request represents a newly triggered RTC.

[2] The buffer size can be changed in the idle time of the processor, e.g., when resources change in the system, or if the system with its current resources allows for this change.

Quality factors available by a certain node are those, which are normally considered by cells (it means, play a role when calculating cost of cells). The total cost of factors available by a node is called $Cost_{total}$.

4- Adaptation algorithm:

In this step, we calculate the lowest-cost feasible solution over the entire set of RTClasses stored in AdaptationRTCArray[3] introduced in the following (This is under the assumption that the chosen set of variants will substitute the current ExpARTCs[4] on the NHP):

- We copy the variants of the WorkingRTCArray into a temporary array AdaptationRTCArray.
- We then reduce AdaptationRTCArray to contain only the RTCs variants, which RTClassID exist in the currently executing periodic ExpARTCs. For each ARTC, that has the property VariantsAllowed set to false, we don´t consider variants that hold the same RTClassID in AdaptationRTCArray, other than the ARTC itself.
- We then add a column that include the AEC as a periodic cell.
- We also add the newly triggered requests. If their properties VariantsAllowed are set to true, we add columns that represent RTClasses equal to the RTClasses of the newly triggered variants. If, however, their properties VariantsAllowed is set to false, we add only such columns that contain the newly triggered variants (a column for each variant). The value of VariantsAllowed might be different among the different requests.
- In this way, we can use the reduced array for the adaptation algorithm as each column represents a participant in the test.

These are the chosen alternatives for ExpARTCs, the AEC, and the newly triggered variants. See Figure 18.

[3] AdaptationRTCArray: is an array of RTCs, on which we solve a knapsack problem, so that we choose an RTC from each column. The chosen RTCs are the ones that will execute in the next hyperperiod.

[4] ExpARTCs is the set of current ARTCs excluding the RTCs, which belong to the deletion requests, and are stored in the buffer. ExpARTCs is updated whenever the adaptation algorithm takes place.

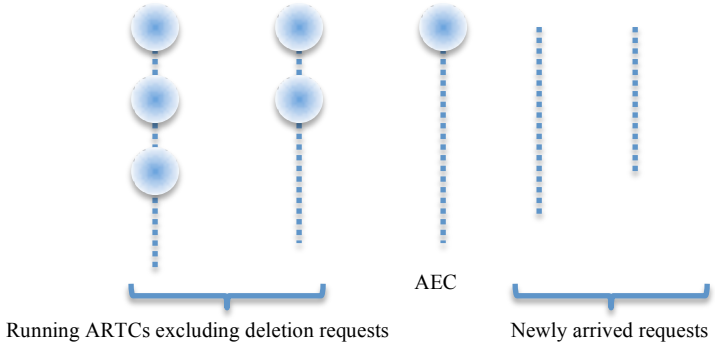


Figure 18: AdaptationRTCArray

To find the solution, we solve the following multiple choice multi dimensional knapsack problem [81] on AdaptationRTCArray:

$$\text{Max } \sum_{i=1}^{m} \sum_{j=1}^{n_i} -Cost_{ij} x_{ij}$$

$$\text{Subject to: } \sum_{i=1}^{m} \sum_{j=1}^{n_i} W_{ij}^{k} \, x_{ij} \leq R^{k}$$

$$\text{Where: } \sum_{j=1}^{n_i} x_{ij} = 1; i = 1..m \; \& x_{ij} \in \{0,1\}; \text{ i = 1..m and j = 1..} n_i$$

m = NumOfColumns, k = 1 : 2

NumOfColumns is the number of columns in AdaptationRTCArray

$$W_{ij}^{1} = Factor_1 / \, Factor_2$$

For any of the ExpARTCs and the newly triggered variants, $Factor_1 = C_{ij}$, $Factor_2 = T_{ij}$

For the AEC, $Factor_1 = WCET_{EC}$, $Factor_2 = WCT_{ECTemp}$[5]

$$W_{ij}^{2} = \text{Cost}$$

R^1 = 1 (i.e. schedulability)

$R^2 = Cost_{total}$ (i.e. no excess of cost budget)

The limit $Cost_{total}$ is optional. If it is set to infinity, then the optimization process tries just to find the lowest-cost solution. If the limit is set to a finite value, then the solution space is further limited.

Note:

$Cost_{AEC}$ stays constant as the only property that might change in the AEC properties by the acceptance of the new request is the number of ARTCs and as the precision of numbers is static, then its related cost is static.

[5] WCT_{ECTemp}: is the expected period of the Engine-cell, in case the requests can be accepted.

Note:

The expected hyperperiod is calculated as the least common multiple of the periods of the periodic ExpARTCs, and the periods of the newly triggered requests. The resulting value is set as initial value for the AEC's expected period WCT_{ECTemp}. If the resulting utilization of the RTCs is below 1 then, we examine the total utilization (AEC and RTCs). If it is smaller or equal to 1, we have found the shortest possible expected period for AEC (which at the same time by definition is the hyperperiod). If the total utilization is beyond 1 then the expected hyperperiod has to be extended by a harmonic multiple until the total utilization is no longer beyond 1.
If the resulting utilization of the RTCs is 1, the set of chosen RTCs results in a non-feasible solution.

In each hyperperiod, only one execution of the AEC is assumed.

- If VariantsAllowed of the ith ExpARTC or any of the newly triggered variants = true, then{

  n_i = the number of variants in the AdaptationRTCArray that hold the same RTClassID of the ith ExpARTC, or the newly triggered variants.

  (In this way, we can choose the best variants in AdaptationRTCArray among all variants that hold the same RTClassIDs of the newly triggered RTCs. The best variants are defined as the variants that consume the minimum cost among those which need time capacity that is sufficient to meet the deadlines of all active cells according to EDF).
  }

Else{

  n_i = 1, and the only variants to be considered in this case is the ith ExpARTC, and the newly triggered variants (which are considered here as the best chosen variants).
  }

  If a solution is found, the newly arrived requests are accepted.

- If the newly arrived requests are accepted by the system, the ExpARTCs set is set to be modified and to be substituted by the chosen alternatives at NHP (delete the ones that should be substituted and

load the alternatives at NHP). The chosen alternatives are put into a ready queue.
The deletion requests are set to take place at NHP. At this time, the Active property of the alternatives and for the newly triggered requests is set to true. After that, step 5 is applied.
The Active property of the alternated cells is set to false by the scheduler once they are replaced (deleted).

The AEC takes care also that the deletion of ARTCs takes place before any triggered request that happens at the same time.

5- **Activate the accepted requests, and update the AEC:** If the newly triggered requests are accepted, the Active property of their RTCs becomes true. They are put into the ready queue, and the AEC schedules the first arrival of each request to be at NHP. This is done by loading the accepted RTCs into the memory (transforming them into ARTCs). The scheduler is responsible for loading the accepted RTCs at NHP.
The AEC updates then its properties according to the changes that will take place. Here, NumOfARTCs is modified according to the accepted requests.
WCT_{EC} get assigned the temporary value which is calculated in step 4, as follows:
- $WCT_{EC} = WCT_{ECtemp}$
The hyperperiod is updated according to step 4. Hyperperiod = WCT_{EC}.
After that, AdaptationRTCArray is set to empty.

6- **Turning the triggered requests into non-triggered:** When a decision is taken about the requests to be accepted or not accepted, the Triggered Property of their RTCs is turned into false.
WorkingRTCArray is copied back to RTCArray, and WorkingRTCArray is reset to empty.

7- **Notify the system, in case the requests are not accepted:** If the set of proceeded requests cannot be accepted, then a notification is sent by the AEC to the system for substituting the proceeded set by another set of requests. $Cost_{total}$, The expected hyperperiod, AdaptationRTCArray, WCT_{ECTemp}, and the ExpARTCs are reset to their initial values.

5.2 Mixed periodic/aperiodic task environments, only independent tasks, potentially adding new tasks

Assumption2: As a combination of periodic and aperiodic is mostly used in control systems, and control systems represent a large field of applications for

the problem described in the thesis, we upgrade the assumption to include hard-deadline periodic and aperiodic tasks.
Let us assume a set of hard-deadline periodic and aperiodic tasks that have to be scheduled on a uni-processor system. The system might receive new requests at run time as a result of some environmental changes. In the simplest case, the new request is adding a new task to the system, or deleting a task from the system.

Problem2: If the new requests are not going to result in a feasible schedule, then a new technique should be developed to reconfigure the system so that it can adapt the new requests at run time preserving all real-time constraints.

Solution2: Here we follow the same principles of solution1. *However, we add some additional properties to the EC, the NumOfPARTCs, NumOfAARTCs, and NumOfANHP. We add also additional properties to the RTC: Type, and Cost_Update.*

We include two additional properties in the EC instead of the NumOfARTCs:

- NumOfPARTCs: is the number of the current periodic ARTCs in the system. This number is calculated the same way NumOfARTCs is calculated, but here we consider the type of RTC
- NumOfAARTCs: is the number of the current aperiodic ARTCs in the system. This number is calculated the same way NumOfARTCs is calculated, but here we consider the type of RTC
- NumOfANHP: refers to the number of aperiodic ARTCs which deadlines exceed the NHP. This parameter is important, as the cells which deadlines exceed the next hyperperiod have a special consideration in recalculating their time characteristics when a new cell is added to the system, or when an update of an RTC happens. The reason for this is that the arrival time of periodic cells happen to be at the NHP, and arrival time of aperiodic cells happen at or after the NHP, and as a result the periodic utilization may change, as well as the aperiodic utilization.

We also include two additional properties in the RTCs, the type of an RTC, and the updated cost:

- Type: the type of an RTC could be periodic or aperiodic.
  All variants, which have the same RTClassID have the same value of property Type.
- Cost_Update: the updated cost, which should be calculated for an RTC, when it replaces another executing RTC. Cost_Update is set to 0 in case of periodic RTCs, because the cost of a periodic cell stays the same, when it replaces an active cell, as the updating takes place on NHP.

We also assume in this assumption, that we handle only one present instance of an aperiodic cell, and we have no information whether there will be subsequent instances, when the cell is released.
We assume that all aperiodic variants, that belong to the same class, have the same deadline.

The Engine-cell algorithm turns to be as follows:

1- **Gathering and filtering the newly deployed RTCs:** same as Assumption 5.1. The procedure that ensures to keep the upper bound of the RTCArray dimensions preserved is also the same. *However, we discuss here the existence of columns that include aperiodic cells.*

   In case the newly deployed RTC adds a new variant to an existing column, the upper bound will be exceeded, and the column includes aperiodic cells, then a decision should be taken, which RTC to drop. The dropped RTC could be the new one as well. To make a decision, we copy the elements of the column in addition to the new RTC and put it into a vector (excluding any variant that could be activated).
   Assuming that the aperiodic cells in each column are ordered according to their costs, from the smallest to the highest value, the new RTC is added to the vector, so that it preserves the order of it.

   Many heuristics for dropping variants are possible. One option may be that we exclude the RTC that consumes the highest cost. An arbitrary exclusion can also take place.
   The exclusion is done on the WorkingRTCArray, if the excluded RTC belongs to the array (i.e. it is not the new RTC).

   Note: If the newly deployed RTC is adding a new RTClass, and all classes in the array are activated, then if the upper bound of classes number in WorkingRTCArray could be exceeded, and the system administrator sees that the arrived RTC is more urgent than one of the active cells, active cell can be scheduled to be deleted on the next possible point (the natural updating point for a periodic cell, and the next possible updating point for an aperiodic cell. In this case, the execution time of the cell is substituted by $y_{UpdatingPoint}$*. If the absolute deadline of the aperiodic ARTC exceeds NHP, then a deletion request for this ARTC is added to the requests buffer). As there is an executable copy of the active RTC in the memory, we can delete the RTClass at this step and replace it with the newly arrived (newly deployed) RTC. The reason is that any alternatives for the deleted RTC will not be needed later. As a result, the RTClass, which includes the deleted RTC, will not be needed.*

2- **Triggering and handling the newly arrived requests:** the arrival times of periodic requests are treated the same as step 2, Assumption 5.1. *However, if the arrival times of aperiodic requests are greater than NHP, they stay the same. If they are smaller than NHP, we set their arrival times the same way as for periodic requests in Assumption 5.1. This is because the AEC can be preempted during its execution, and as a result a decision about the newly arrived requests might not take place before NHP.*

3- **Calculating the cost of quality factors for the system:** Calculating $Cost_{total}$ is done the same way as in Assumption 5.1. The cost of the AEC is a constant value, for the same argument mentioned in Assumption 5.1.

4- **Adaptation algorithm:**

In this step, we calculate the lowest-cost feasible solution over the entire set of RTClasses stored in AdaptationRTCArray introduced in the following (This is under the assumption, that the periodic alternatives will substitute the current periodic ARTCs on the NHP, and the aperiodic alternatives will substitute the current aperiodic ARTCs on the next possible updating point):

- We copy the variants of the WorkingRTCArray into a temporary array AdaptationRTCArray.
- We then reduce AdaptationRTCArray to contain only the RTCs variants, which RTClassID exists in the ExpPARTCs[6] and ExpAARTCs[7] with absolute deadlines exceeding NHP. For each ExpPARTC or ExpAARTC, that has the property VariantsAllowed set to false, we don´t consider variants that hold the same RTClassID in AdaptationRTCArray, other than the ARTC itself.
- For each aperiodic ARTC that should be deleted (stored in the requests buffer), and has absolute deadlines exceeding NHP, we add a column including the ARTC as the only variant. If a next possible updating point exists, its execution time is set to $y_{UpdatingPoint}$. The mentioned updating

[6] ExpPARTCs ist the set of current periodic ARTCs excluding the RTCs, which belong to the deletion requests, and are stored in the buffer. ExpPARTCs is updated whenever the adaptation algorithm takes place.

[7] ExpAARTCs ist the set of current aperiodic ARTCs excluding the RTCs, which belong to the deletion requests, and are stored in the buffer. ExpAARTCs is updated whenever the adaptation algorithm takes place.

point is the next possible updating point. The reason is that updating points are the most suitable to apply deletion, as partial results are delivered on these points. Deleting a cell suddenly on an arbitrary point may cause errors.
- We then add a column that includes the AEC as a periodic cell.
- We also add the newly triggered requests. If their properties VariantsAllowed are set to true, we add columns that represent RTClasses equal to the RTClasses of the newly triggered variants. If, however, their properties VariantsAllowed are set to false, we add only columns containing the newly triggered variants (a column for each RTC). The value of VariantsAllowed might be different among the different requests.
- In this way, we can use the reduced array in the next step for the adaptation algorithm as each column represents a participant in the test. These are the chosen alternatives, the AEC, and the newly triggered variants. See Figure 18.
We arrange the array so that the columns that represent periodic cells are stored first. Afterwards, the AEC is stored, and finally, the columns that represent aperiodic cells.

Here the initial expected hyperperiod is set to the least common multiple of the periods of the periodic ARTCs).
As described in Section 5.1/ step 4, the expected hyperperiod considers the newly arrived requests if they are periodic.

Let us assume that:
- The number of columns in AdaptationRTCArray = Num.
- N` is the number of columns, which represent the newly triggered aperiodic requests.

If (NumOfANHP > 0) then {

- We calculate the arrival times, execution times, and Cost_Update for the running aperiodic ARTCs that are stored in AdaptationRTCArray, and deadlines exceed the NHP as follows:

  - New execution time is assigned to the variants:

    Amount = Hyperperiod – $[(\sum_{i=1}^{NumOfPARTCs}((\text{Hyperperiod } /T_i) * C_i)) + \text{WCET}_{EC} + \sum_{i=1}^{NumOfAARTCs-NumofANHP}(C_i - ET_{executed\,i})]$

Amount is the time left in the current hyperperiod, after excluding the time that should be spent in executing the periodic ARTCs, and aperiodic ARTCs, which deadlines do not exceed NHP.

If (Amount > 0) then {

- We construct a vector V of the running aperiodic ARTCs, which deadlines exceed NHP.

- We order the elements of this vector according to the increasing absolute deadlines.

- For (i = 1 to NumOfANHP){

  If ($C_i - ET_{executed_i} \leq$ Amount) then {
  Amount = Amount – ($C_i - ET_{executed_i}$)
  Exclude the column of the i_{th} aperiodic variant from AdaptationRTCArray.
  Decrease Num by 1.
  }
  else{
  $C_{i,new}$ = ($C_i - ET_{executed_i}$) – Amount.

  Set the execution time of the variant in AdaptationRTCArray that is equal to the i_{th} variant to $C_{i,new}$.

  Set the arrival time of the variant in AdaptationRTCArray that is equal to the i_{th} variant to Arrival time = NHP, if Arrival time < NHP
  Amount = 0.
  }
  }
}

else{

For (i = 1 to NumofANHP){
$C_{i,new} = C_i - ET_{executed_i}$

Set the execution time of the variant in AdaptationRTCArray that is equal to the i_{th} variant to $C_{i,new}$.

Set the arrival time of the variant in AdaptationRTCArray that is equal to the i_{th} variant to Arrival time = NHP, if Arrival time < NHP

}}

- Cost_Update = the cost of the RTC

- The following iteration is done over the aperiodic RTCs in AdaptationRTCArray, which do not belong to the newly triggered requests:

  For ($k = Number\ of\ periodic\ columns + 1..Num - N$`){

  If (VariantsAllowed = true) && (there exists an updating point in the part of the running variant dedicated for $C_{i,new}$ (after NHP)) then{

  The following calculations are done to include the possible alternatives for the active aperiodic cells in the knapsack problem:

  - a new arrival time, execution time, cost and absolute deadline is calculated for the variants of the k_{th} column in AdaptationRTCArray, excluding the running variant in the k_{th} column:

  - Arrival time = Arrival time of the active variant in the k_{th} column.

  - New execution time is assigned to each variant in the k_{th} Column, excluding the running variant:

    $C`_{k,new} = (C` - y`)$[8] $+ (C_{k,new} - (C_{running,variant} - y)$[9]$)$

    C` is the execution time of the j_{th} variant, for which we are calculating the attributes, in the k_{th} column.

[8] (C` – y): This denotes the remaining computation time for the k_{th} variant after updating point y.

[9] $C_{running,variant} - y$: This denotes the remaining computation time of the running variant after update point y.

$C_{k,new}$ is the calculated execution time of the running variant in the k_{th} column.
$C_{running,variant}$: is the original execution time of the running variant in the k_{th} column.
y is the relative updating point time of the next updating point in the running variant.
y` is the relative updating point time of the counterpart updating point in the j_{th} variant.

- $Cost\_Update_j$ = Maximum of (Cost of the running variant, cost of the j_{th} variant).

- Specified absolute deadline = max (Specified absolute deadline for the running variant, NHP + ($C_{k,new}$ − ($C_{running,variant}$ − y))[10] + specified relative deadline)[11].

}
else
We choose the running variant in the k_{th} column.
}
}

To find the solution, we solve the following multiple choice multi dimensional knapsack problem:

$$\text{Max } \sum_{i=1}^{Num} \sum_{j=1}^{n_i} -Cost_{ij} x_{ij}$$

$$\text{Subject to: } \sum_{i=1}^{Num} \sum_{j=1}^{n_i} W_{ij}^{k} x_{ij} \leq R^{k}$$

$$\text{Where: } \sum_{j=1}^{n_i} x_{ij} = 1; i = 1..m \ \& x_{ij} \in \{0,1\}; \text{ i = 1..m and j = 1..}n_i, \text{ k = 1 : 3}$$

- $W_{ij}^{1} = Factor_1 / Factor_2$

For any of the periodic RTCs: $Factor_1 = C_{ij}$, $Factor_2 = T_{ij}$
For the AEC, $Factor_1 = WCET_{EC}$, $Factor_2 = WCT_{ECTemp}$

Here the initial value for the AEC's expected period WCT_{ECTemp} is equal to the least common multiple of the periods of the periodic ExpPARTCs in

[10] It might be that at the selected update point, the old version would need little remaining time while the updating request much more and the deadline might be missed.
[11] All variants that belong to a class share the same deadlines.

AdaptationRTCArray, and the periods of the newly triggered periodic requests in AdaptationRTCArray.

For any of the aperiodic RTCs: $Factor_1 = 0$, $Factor_2 = 1$[12]

- $W_{ij}^2 = Factor_1 - Factor_2$

For any of the periodic RTCs and the AEC: $Factor_1 = 0$, $Factor_2 = 0$
For any of the aperiodic RTCs: $Factor_1 = d_{Specified,ij}$, $Factor_2 = d_{Calculated,ij}$

Where:

$d_{Specified,ij}$: The specified absolute deadline for any aperiodic variant, which belongs to an aperiodic variant in AdaptationRTCArray is equal to its arrival time + relative deadline of the variant.

$d_{Calculated,ij} = \max\{d_{Calculated(i-1)j_{i-1}}, ArrivalTime_{ij}\} + C_{ij,new}/Us$.
$d_{Calculated(Number\ of\ priodic\ columns\ in\ Array1+1)j_{Number\ of\ periodic\ columns\ in\ Array1+1}} = 0$.
$U_s = 1-U_p$.

Depending on the different kinds of RTCs to be considered in solving the Knapsack problem, W_{ij} is defined as follows:

- W_{ij}^3 = Cost for periodic RTCs stored in AdaptationRTCArray

W_{ij}^3 = Cost_Update for running aperiodic RTCs that are stored in AdaptationRTCArray

W_{ij}^3 = Cost for added aperiodic RTCs stored in AdaptationRTCArray

$$R^1 = 1$$
$$R^2 = 0$$
$$R^3 = Cost_{total}$$

The limit $Cost_{total}$ is optional. If it is set to infinity, then the optimization process tries just to find the lowest-cost solution. If the limit is set to a finite value, then the solution space is further limited.

If a solution is found, the newly arrived requests are accepted.

If the newly arrived requests are accepted by the system, the ARTCs

[12] This assignment garantees that the aperiodic RTCs are not considered in the utilization test.

set or subset (which is represented in AdaptationRTCArray) is set to be modified and to be substituted by the chosen alternatives (Replacing a periodic ARTC means deleting the periodic ones that should be substituted and loading the periodic alternatives at NHP. Replacing an aperiodic ARTC means, the replaced RTC can be treated as a deletion request. When the deletion takes place, the information necessary for replacing the ARTC (transferred from replaced RTC to the replacing one to) should be stored. This can be differentiated from a normal deletion request by comparing the RTClassID of the RTC to be deleted with other RTClassIDs in the ready queue. If there exists an identical RTClassID, then it is a replacement process. The comparison is done by the scheduler, and the replacement is done at the next updating point that happens after the NHP).
The chosen alternatives are stored in a ready queue. At the NHP, the Active property of the alternatives and for the newly triggered requests is set to true. The Active property of the alternated cells is set to false once they are replaced (deleted).
This fact is to be applied also on the next assumptions, whenever aperiodic alternatives take place.
The Active property becomes true for the alternatives. After that, step 5 is applied.

The AEC takes care that the deletion requests take place before the triggered requests.

5- Activate the accepted request, and update the AEC: same as in Assumption 5.1. *NumOfAARTCs and NumOfPARTCs are calculated according to the type of requests. NumOfAARTCs is increased by number of accepted aperiodic RTCs, if the newly triggered RTCs are aperiodic, or the NumOfPARTCs is increased by number of accepted periodic RTCs, if the newly triggered RTCs are periodic. NumOfAARTCs is descreased by number of aperiodic RTCs that are to be deleted. NumOfPARTCs is decreased by number of periodic RTCs that are to be deleted.*
The AEC schedules the first arrival of periodic requests at the NHP. The scheduler is responsible for loading the accepted RTCs (periodic and aperiodic) at NHP.

6- Turning the triggered requests into non-triggered: same as in Assumption 5.1.

7- Notify the system, in case the requests are not accepted: same as in Assumption 5.1. ExpPARTCs, and ExpAARTCs are reset to their initial values.

5.3 Mixed periodic/aperiodic task environments, only independent tasks, adding new RTCs or updating existing ones

Assumption3: Let us assume a set of hard-deadline periodic and aperiodic tasks that have to be scheduled on a uni-processor system. The system might receive new requests at run time as a result of some environmental changes. These requests might include adding a new task to the system, updating a task, which already exists in the system, or deleting a task from the system.

Problem3: If the new requests are not going to result in a feasible schedule, then a new technique should be developed to reconfigure the system so that it can adapt the new requests at run time preserving all real-time constraints.

Solution3: Here we follow the same principles of solution2.

However, here we introduce how an update request is defined. We also explain how the Active property of the update and updated cells changes.

Any update of an RTC is also an RTC, which is dedicated to update the functionality of that RTC, according to some new environmental changes. Each update is represented by an RTClass, which is going to substitute an RTClass in the RTCArray. Each class holds a number of variants, which are also RTCs, but they are dedicated to fulfill the same task of the update, with different cost and time characteristics.
All variants that belong to a specific update have the same Essential value.
The Essential value might, however, be different from one update to another.
The number of essential classes has to be smaller than the upper bound of the RTClasses number that can be stored in the RTCArray.

In case of an update for a periodic or aperiodic cell, the Active property of the updated cell becomes false once it is replaced. We assume, in case of periodic or aperiodic update that turning the property into false is done by the kernel.

Updating a periodic ARTC is interpreted as deleting the updated RTC, and adding the updating one at NHP. Updating an aperiodic ARTC is interpreted as replacing the updated ARTC at the updating point next to NHP. The aperiodic updating request is considered only for the aperiodic ARTCs, with absolute deadlines exceeding NHP. Otherwise, the update request is deleted.

When applying the changes in the activation step for accepted requests, and the deletion takes place as part of the updating process of periodic ARTCs, the information necessary for the updating RTC should be stored. We can differentiate the deletion as part of the updating process from normal deletion, by comparing the RTClassID of the RTC to be deleted with other RTClassIDs in

the ready queue. If there exists an identical RTClassID, then it is an update process. This is because the update request is split into adding an RTC and deleting an RTC. If both are accepted, they occupy two places in the ready queue.

And the Engine-cell algorithm turns to be as follows:

1- **Gathering and filtering the newly deployed RTCs:** same as Assumption 5.2.
Here, we discuss two different cases:
- In the first case, a newly deployed RTC has to be added same as Assumption 5.2.
- In the second case, a newly deployed RTClass that has a ClassID, which exists in the WorkingRTCArray has to be added. This means that we are updating an existing column. If no variant is active in the existing column, we substitute it by the newly arrived one. If there is an active variant in the existing column, we add the newly arrived RTClass to a queue. We call it the UpdateQueue.

2- **Triggering and handling the newly arrived requests:**
The same as Assumption 5.1. *However, here, each request represents either a newly triggered RTC (newly triggered variant) or a newly arrived update.*

In case of an update, the request is represented by a list of RTCs that constructs the RTClass of the update. The triggered RTC in the list is the one that is going to replace the running RTC, in case VariantsAllowed = false, otherwise a best variant should be chosen from the list.

a similar queue to the UpdateQueue (constructed in the previous step) of triggered requests (add/delete RTCs) will be constructed. Then, a first iteration is done in parallel over each item in both queues. According to the priority, either an update is chosen or a triggered request (add/delete). The priority might be decided by the administrator of the system. For simplicity, the decision can be made arbitrarily. If the number of chosen requests is less than upper bound of requests, then another iteration is done on the UpdateQueue and the TriggeredQueue parallelly to choose requests, which were not chosen in the first iteration. The second iteration continues until number of requests is equal to upper bound or until no further requests exist.

Selected requests are stored in a queue called the RequestQueue. When these requests are considered by the AEC, the UpdateQueue, TriggeredQueue, and RequestQueue become empty again.

3- **Calculating the cost of quality factors for the system:** same as Assumption 5.2.

4- **Adaptation algorithm:**

Same as in Assumption 5.2. *However,*

- *when constructing AdaptationRTCArray, adding an aperiodic update is done (only if there exists an updating point after NHP in the aperiodic variant that is running) by adding the arrived RTClass which includes the triggered updating variant. If VariantsAllowed is equal to true, the class contains all variants which belong to this update. Otherwise, it contains only the triggered updating variant. The replacement is done at the next updating point (after NHP) of the running updated variant. In the following, we summarize how to check the existence of an updating point after NHP (Only in this case, the updated variant should be excluded when constructing ExpAARTCs), and how to set the time characteristics for the variants in the updating column:*

First (Determining the set of ARTCs that can be updated):

Amount1 = Hyperperiod – $[(\sum_{i=1}^{NumOfPARTCs}((\text{Hyperperiod } /T_i) * C_i)) + \text{WCET}_{EC} + \sum_{i=1}^{NumOfAARTCs-ANHP}(C_i - ET_{executed_i})]$

Amount1 is the time left in the current hyperperiod, after excluding the time that should be spent in executing the periodic ARTCs, and aperiodic ARTCs, which deadlines do not exceed NHP.

If (Amount1 > 0) then {

- For (i = 1 to NumofANHP){

If ($C_i - ET_{executed_i} \leq$ Amount1) then{
Amount1 = Amount1 – ($C_i - ET_{executed_i}$)

}
else{
$C_{i,new} = (C_i - ET_{executed_i})$ – Amount1.

If there exist an updating point in $C_{i,new}$
Add this ARTC to the set of variants that can be be updated
}
}}

else{
For (i = 1 to NumofANHP){
$C_{i,new} = C_i - ET_{executed\,i}$
If there exists an updating point in $C_{i,new}$
Add this ARTC to the set of variants that can be updated

}}

Second (Calculation of time characteristics for the updates):

If the found updating point is UP[x,y], the arrival time of the j_{th}[13] variant is set to the arrival time of the updated variant. The execution time for the j_{th} variant is set to $(y + C_j - y`)$, where y` is the relative updating point time for the counterpart updating point.
The specified absolute deadline for the j_{th} variant is set to max[$((D_j - y`) +$ (Arrival time of the running variant + y)), AbsoluteDeadline of the running variant that should be updated]

- Adding a periodic update is done by adding the arrived RTClass, which includes the triggered updating variant to AdaptationRTCArray. If VariantsAllowed is equal to false, only the triggered updating variant exists in the column.
Otherwise, all variants which belong to the update exist in the column. The updated variant has to be excluded when constructing ExpPARTCs, because executions of periodic instances are completed in each hyperperiod. This means, that when periodic update is applied in the next hyperperiod, we do not assume any execution of the updated variant.

If a solution is found, the newly arrived requests are accepted.

If the newly arrived requests are accepted by the system, the ARTCs set or subset (which is represented in AdaptationRTCArray) is set to be modified and to be substituted by the chosen alternatives. At the NHP, the Active property of the alternatives and for the newly triggered requests is

[13] j_{th}: j indicates the index of the variant in the updating column.

set to true. The Active property of the alternated cells or updated cells is set to false once they are replaced (In the aperiodic case, the part of the updated cell that follows the first updating point after NHP is to be replaced). At the replacement point for aperiodic cells, any data of the altered cells or updated cells that might be necessary for the alternatives or updating variants is stored. This fact is to be applied also on the next assumptions, whenever aperiodic alternatives take place, or aperiodic updates take place.

After that, step 5 is applied.

5 - Activate the accepted requests, and update the AEC: same as in Assumption 5.2.

Here, in case the request or any of the requests is an update for an active RTC, we replace the RTClass of the variant that should be updated by the elements of the newly arrived request (the newly arrived RTClass). We set the Active property of the triggered element in the newly arrived RTClass to true.

6 - Turning the triggered requests into non-triggered: same as in Assumption 5.2.

7 - Notify the system, in case the requests are not accepted: same as in Assumption 5.2.

5.4: Mixed periodic/aperiodic task environment, potential dependencies, no updates of dependent tasks allowed

Assumption4: Let us assume a set of hard-deadline periodic and aperiodic tasks. Aperiodic tasks might have dependencies between each other. These tasks have to be scheduled with the modified earliest deadline first algorithm (EDF*) on a uni-processor system. The system might receive new requests at run time as a result of some environmental changes. These requests might include adding a new task or a set of dependent new tasks to the system (the dependent set could only contain aperiodic tasks). A request could also be updating a task with no dependencies that already exists in the system. It could also be deleting a task or a set of dependent tasks.

Problem4: If the new requests are not going to result in a feasible schedule, then a new technique should be developed to reconfigure the system so that it can adapt the new requests at run time preserving all real-time constraints.

Solution4: Here we follow the same principles of solution3.

However, we add a new property to the properties of real-time cells. This property is called RelatedCells. We also make assumptions on the dependencies between related cells.

- RelatedCells (RC): includes the set of cells that construct the precedence graph.

We assume that all variants, which belong to any column in the RTCArray have the same dependencies. The dependencies here mean any state variable that might change in the system as a result of these dependencies (any state variable, which value is transmitted from one RTC to another, if there exists a dependency between the two RTCs).

And the Engine-cell algorithm turns to be as follows:

1- **Gathering and Filtering the newly deployed RTCs:** same as Assumption 5.3.

2- **Triggering and handling the newly arrived requests:** same as Assumption 5.3.

   However, a request could be here a newly triggered set of dependent RTCs, or deleting a set of dependent ARTCs. In such a case, a request can be accepted only if the entire set can be accepted.

   When the request is adding a set of dependent RTCs triggered from the WorkingRTCArray, the request is loaded by saving a reference for the RTCs IDs in the buffer.

   If the request includes a set of dependent cells, we assume that their modified arrival times and deadlines are calculated offline by EDF* [7]. When they arrive to the system, they are independent.
   a_i = current time + $a_{i,modified}$, and d_i = current time + $d_{i,modified}$ for any of the dependent variants; i = 1..DEP_n. DEP_n is the number of dependent cells.
   If $a_{modified}$ is smaller than the NHP, then $a_i = a_{i,modified}$ + FixedAmount, and $d_i = d_{i,modified}$ + FixedAmount; where FixedAmount = NHP - $a_{i,modified}$; i = 1..DEP_n. This offset is necessary.

3- **Calculating the cost of quality factors for the system:** same as Assumption 5.2.

4- Adaptation algorithm:

The adaptation algorithm is done as in Assumption 5.3.
When calculations are done to include the possible alternatives for the active aperiodic cells in the knapsack problem, we also assume that $a_{modified}$ *and* $d_{modified}$ *are calculated offline.*

If a solution could be found, then the substitution process is done as in Assumption 5.3.
After that, step 5 is applied.

Note: The Active property for the alternatives and the alternated cells is set as in Assumption 5.3.

5- Activate the accepted requests, and update the AEC: Same as in Assumption 5.3.
WCT_{EC} is assigned the temporary value, which is calculated in step 4.
The hyperperiod property is updated according to step 4.
In case the request is an update, the RTClass of the newly arrived update replaces the respective RTClass in the WorkingRTCArray.

6- Turning the triggered requests into non-triggered: same as in Assumption 5.3.

7- Notify the system, in case the requests are not accepted: same as in Assumption 5.3.

5.5 Mixed periodic/aperiodic task environment, potential dependencies, updates of dependent and independent tasks allowed

Assumption5: Let us assume a set of hard-deadline periodic and aperiodic tasks. Aperiodic tasks might have dependencies between each other. These tasks have to be scheduled with the modified earliest deadline first algorithm (EDF*) on a uni-processor system. The system might receive new requests at run time as a result of some environmental changes. These requests might include adding a new task or a set of dependent new tasks to the system. It might also include updating a task or a set of dependent tasks, which already exists in the system. The requests might be also deleting a task or a set of dependent tasks.
We assume that by updating dependent tasks, no new tasks are added to the graph, and no existing tasks are deleted.

Problem5: If the new requests are not going to result in a feasible schedule,

then a new technique should be developed to reconfigure the system so that it can adapt the new requests at run time preserving all real-time constraints.

Solution5: Here we follow the same principles of solution4.

And the Engine-cell algorithm turns to be as follows:

1- Gathering and Filtering the newly deployed RTCs: same as in Assumption 5.3.
In case there is no running RTCs with the same RTClassIDs as the arriving update, then arriving lists replaces directly RTClasses with the same RTClassIDs in the WorkingRTCArray.

2- Triggering and handling the newly arrived requests: same as in Assumption 5.3.

However, in this Assumption, each request is either a newly triggered RTC (newly triggered variant), a newly triggered set of dependent RTCs or a newly arrived update (a new variant which should substitute an existing variant, or a set of dependent aperiodic RTCs which is going to substitute a set of dependent aperiodic ARTCs). It could also be deleting a variant, or a set of dependent variants.

In case the request consists of more than one RTC, a request can be accepted only if the entire set can be accepted.

When the request is adding a set of dependent RTCs triggered from the WorkingRTCArray, the request is loaded by saving a reference for the RTCs IDs in the buffer.

In case of an update, the request is represented in the RequestQueue[14] by the of lists of RTCs. Each list constructs an RTClass of the newly arrived update.

Updating a set of dependent cells is done under the same rules of updating a cell described in Assumption 5.3. The updating cells are the triggered elements of the arriving lists.

3- Calculating the cost of quality factors for the system: same as Assumption 5.4.

---

[14] RequestQueue: is a queue that includes the selected requests, which will proceed to the adaptation algorithm.

4- Adaptation algorithm:
Also here, we assume that the dependent sets have their modified arrival times and deadlines calculated offline, same as in Assumption 5.4.
The adaptation algorithm is done the same as in Assumption 5.4. However, when calculating the time characteristics of the variants in the columns that are updating dependent variants, the following is applied:

If the found updating point after NHP is UP[x,y], the arrival time of the j_{th} variant is set to the arrival time of the updated variant. The execution time for the j_{th} variant is set to $(y + C_j – y`)$, where y` is the relative updating point time for the counterpart updating point.
The specified absolute deadline for the j_{th} variant is set to max[$((D_j –$ y`) + (Arrival time of the running variant + y)), AbsoluteDeadline of the running variant that should be updated].

If a solution could be found, then the substitution process is done as in Assumption 5.4.
Note: The Active property is set for the alternatives and the alternated cells as in Assumption 5.4.
However, the update here could be for more than one aperiodic cell.

5- Activate the accepted requests, and update the AEC: Same as in Assumption 5.4. In case the request is an update for one or several active RTCs, it replaces the RTClasses in the WorkingRTCArray, which include the RTC/RTCs that should be updated by the RTClass/RTClasses of the newly arrived request.

6- Turning the triggered requests into non-triggered: same as in Assumption 5.4.

7- Notify the system, in case the requests are not accepted: same as in Assumption 5.4.

Chapter 6 Proof of Boundedness and Complexity Estimation

In this chapter, we provide a boundedness proof for each step, and an abstracted version of the code which reflects the algorithmic structure. This structure follows directly from the concept presented in Section 5.5.
In order to ease reading, the concept as described in Section 5.5 is reapeated here in expanded form, i.e. the texts of assumtions 1 - 4 referred to in the description of assumption 5 are inserted here.

The elementary operations inside the algorithmic constructs are listed as well. Based on this structure, we calculate the time complexity. Each substep is represented by a Nassi-Schneidermann diagram [88], where complexities are also assigned to each code statement. This in turn allows to understand the complexity calculation for each substep. Complexities are written to the left side of each statement. Complexity of a diagram is the sum of complexities for levels of this diagram. For and While loops multiply the number of times a loop runs by the complexity of the statements inside the loop. If we have an If else blocks, then we choose the higher complexity of the two blocks.

We show that the computation time needed for the modification process is bounded.

The algorithm is bounded if each of the steps is bounded.
A step is bounded, if it needs bounded time. We analyze each of the steps for boundedness in terms of the time to be spent to execute the respective step.

Boundedness is equivalent to the fact that the algorithm terminates after a finite time whenever being started. There are three major possible reasons for violating boundedness (for causing non termination):

1) There might be external influences which are not under control of the algorithm. These may hinder termination. (The Priority Inversion problem is an example for this as there might be an unknown number of unknown tasks which by having intermediate priority may cause unbounded blocking). In our algorithm external influences do not exist.

2) There might be deadlocks. We assume in our algorithm that the underlying RTOS applies SRP (the Stack Resource Protocol). SRP excludes not only unbounded blocking time but also deadlocks.

3) There might be while/until loops which are not terminating:
A classical test in this case is checking whether the function to be calculated is bounded, mononotonic and not asymptotic. If these three conditions are true then we are sure that the respective loop will terminate.

In the following, we present the definition of bounded, mononotonic and asymptotic functions. Then we go through the steps of Assumption 5.5, and we analyze the time spent in applying them:

Bounded functions: "A function is bounded from below if there is k such that for all x, $f(x) \geq k$. A function is bounded from above, if there is K such that, for all x, $f(x) \leq K$." [89]

Monotonic functions: "A function is monotonically increasing on an interval *I* if for any *x*1 and *x*2 in *I*, *x*1 is less than *x*2 implies that *f*(*x*1) is less than or equal to *f*(*x*2).

A function is monotonically decreasing on an interval *I* if for any *x*1 and *x*2 in *I*, *x*1 is less than *x*2 implies that *f*(*x*1) is greater than or equal to *f*(*x*2)." [90]

Asymptotic functions: "A function that increases or decreases until it approaches a fixed value, at which point it levels off," (when x tends versus infinity). [91]

During the execution of the following steps, there are no external factors (as e.g, potential blockings or shared resouces, etc.) that may influence the boundedness.

The reason is that:
- communications between nodes or with other systems are assumed to take place on a deterministic basis. For this reason, blockings by I/O operations do not exist.
- No blocking on resources can happen, as each RTC has its own set of resources. The existence of such available set (not used by any other RTC) is checked before accepting the RTC.
- Communication between processes is not within the scope of the thesis. This negates potential blocking for a process by other processes. [92]

The variables, which construct the complexities are:

SC: The upper bound of the dependent cells, that may construct a request.

h: The upper bound of the number of columns in the RTCArray (number of RTClasses).

f: The upper bound of the number of RTCs in an RTClass.

b: The upper bound of the newly deployed RTCs.

PN: an upper bound of number of parameters in the system.

QB: The upper bound of requests that can be received in each execution of the Engine-Cell.

m1:The sum of utilization factors (execution time / period) for the periodic load and AEC (assuming that period of AEC is least common multiple of periods of periodic RTCs), approximated to the next integer number.

n: is the number of updating points in a cell.

GRP: The value of the greatest period available in the RTCArray, including to the expected period of the Engine-cell.

NInd: Number of individuals in a generation within the genetic algorithm solving the Knapsack problem.

All parameters are assumed to have a predefined upper bound which guarantees boundedness of computation time. No parameters other than the previous ones may influence the computation.

Assumption: Let us assume a set of hard-deadline periodic and aperiodic tasks. Aperiodic tasks might have dependencies between each other. These tasks have to be scheduled with the modified earliest deadline first algorithm (EDF*) on a uni-processor system. The system might receive new requests at run time as a result of some environmental changes. These requests might include adding a new task or a set of dependent new tasks to the system. It might also include updating a task or a set of dependent tasks, which already exists in the system. The requests might be also deleting a task or a set of dependent tasks.
We assume that by updating dependent tasks, no new tasks are added to the graph, and no existing tasks are deleted.

Problem: If the new requests are not going to result in a feasible schedule, then a new technique should be developed to reconfigure the system so that it can adapt the new requests at run time preserving all real-time constraints.

Solution: Here we follow the same principles as described in section 5.5.

And the Engine-cell algorithm turns to be as follows:

1- **Gathering and Filtering the newly deployed RTCs:**

   As newly deployed RTCs could be added at run time, the first step of the AEC is to collect the RTCs, and store them in the a WorkingRTCArray (a copy of RTCArray) following a procedure that ensures to keep the upper

bound of the WorkingRTCArray dimensions preserved. The upper bound of the WorkingRTCArray dimensions is equal to the upper bound of RTCArray dimensions.
The newly deployed RTCs remain located in the WorkingRTCArray. They might be activated later, only if they become part of a request, and this request is accepted. Newly deployed RTCs enlarge the solution space when applying the adaptation algorithm, because they represent new cells or new variants of cells that can raise the adaptation possibilities according to the events that trigger adaptations or according to time and cost characteristics. Newly deployed RTCs become part of the WorkingRTCArray. Requests, however may include a set of Newly deployed RTCs or not. Newly deployed RTCs are deployed offline by parties different from those who select requests. Requests are selected online according to the environmental changes.

Let b be the upper bound of newly deployed
RTCs that can arrive at this step.
Here is a description for the procedure:
For the purpose of providing predictability we restrict ourselves to fixed upper bounds in both dimensions of the WorkingRTCArray.
Let us assume that f is the upper bound of the different variants of each class in the WorkingRTCArray, and h is the upper bound of the different RTClasses that can be stored in the WorkingRTCArray, as pointed out by Figure 17. If the newly imported RTCs may cause the exceed of the upper bound of variants in a column, or the upper bound of columns, the Engine-cell preserves the upper bounds by applying a filter procedure. In this context, we discuss the following different cases. In the first case, a newly deployed RTC has to be added. In the second case, a newly deployed RTClass that has a ClassID which exists in the WorkingRTCArray has to be added.

1- Each time a set of newly deployed RTCs should be added to the WorkingRTCArray, the filter procedure classifies the arrived RTCs according to their IDs. Each RTC is treated alone. If the respective RTClassID does not exist in the array, this means that we are adding a new column. If the RTClassID exists and the VariantID does not exist, then we are adding a new variant to an existing column.
If the upper bound of any of the influenced dimensions would not be exceeded, then the RTC is added to the WorkingRTCArray.
The RTCs that are adding new columns are processed before the RTCs that are adding new variants to the columns. The reason is that adding a new column might cause the deletion of another column from the WorkingRTCArray. In this case no new variants can be added to the deleted column.

As one possible heuristic for removal of an RTClass an ImportanceFactor-based approach may be applied, as described below.
If the newly deployed RTC is adding a new RTClass and the influenced dimension is exceeded, then the RTClasses that could be chosen to be replaced by the newly arrived one are the classes that are not essential or activated. We then make a list of these classes in addition to the added one. We exclude the class with the smallest ImportanceFactor.
(Other strategies could also be used to perform removal of a class).

Note: If the newly deployed RTC is adding a new RTClass, and all classes in the array are activated or essential, then if the upper bound of classes number in WorkingRTCArray could be exceeded, and the arrived RTC is more urgent than one of the active cells, this active cell can be scheduled to be deleted on the next possible point (the natural updating point for a periodic cell, and the next possible updating point for an aperiodic cell. In this case, the execution time of the cell is substituted by $y_{UpdatingPoint}$. If the absolute deadline of the aperiodic ARTC exceeds NHP, then a deletion request for this ARTC is added to the requests buffer). As there is an executable copy of the active RTC in the memory, we can delete the RTClass at this step and replace it with the newly arrived (newly deployed) RTC. The reason is that any alternatives for the deleted RTC will not be needed later. As a result, the RTClass, which includes the deleted RTC, will not be needed.

If the upper bound will be exceeded and the RTC adds a new periodic variant to an existing column, then a decision should be taken, which RTC to drop. To make a decision, we copy the elements of the column in addition to the new RTC and put it into a vector.

There might be many heuristics for dropping variants and classes, as for example LRU [93][94], Pseudo LRU[95], or RR[96].
Arbitrary dropping can be executed in constant time while any of the mentioned heuristics may be implemented in a way that it demands not more than linear time.
Assuming that the periodic cells in each column are ordered according to utilization factor (C_i/T_i), from the smallest to the highest value, the new RTC is added to the vector, so that it preserves the order of it.
After that we examine the RTC in the vector with the highest C_i/T_i. If it does not result in a successful schedulability test together with the AEC (the AEC is considered in the test with its execution time and period that were assigned to it before it is activated), then we exclude it. However, if it results in a successful schedulability test, then we exclude the RTC with

the highest cost. The exclusion is applied on the WorkingRTCArray, if the excluded RTC belongs to the array.

In case the newly deployed RTC adds a new variant to an existing column, the upper bound will be exceeded, and the column includes aperiodic cells, then a decision should be taken, which RTC to drop.
To make a decision, we copy the elements of the column in addition to the new RTC and put it into a vector (excluding any variant that could be activated).
Assuming that the aperiodic cells in each column are ordered according to their costs, from the smallest to the highest value, the new RTC is added to the vector, so that it preserves the order of it.
Many heuristics for dropping variants are possible. One option may be that we exclude the RTC that consumes the highest cost. An arbitrary exclusion can also take place.
The exclusion is done on the WorkingRTCArray, if the excluded RTC belongs to the array.

We assume that the b is smaller or equal to the upper bound QB of the number of requests that is allowed to arrive in each execution of the Engine-cell. The reason is that in case of all arrived newly deployed RTCs are urgent, and all classes which fill the upper bound of number of classes in the WorkingRTCArray are active, we will have a number of requests equal to b.
These requests have to be stored in a buffer of an upper bound size QB.
Any other circumstances (RTCs with equal costs, or all essential classes do already exist in the WorkingRTCArray) can let the newly deployed RTCs be a part of the comparison process. The reason is that:
- In case of adding a variant: we perform arbitrary removal, when costs are equal.
- In case of adding a class: the number of essential classes is always smaller than the upper bound permitted for the classes number in the array. We assume that determining the number of essential classes belongs to the system design. Determining the number of permitted classes in the array depends on the system resources. We assume initially that system design and array dimensions fit together. Both numbers can change at run time. When system resources change, this may allow for a change in the number of essential classes.
The upper bounds of the WorkingRTCArray might change over time according to the available resources. This enables to add additional classes.
Obviously when applying the previous heuristic, it restricts the search space of the subsequent activities of the algorithm. E.g. a variant may be dropped from the WorkingRTCArray due to the fact that at the moment

there is no sufficient processor capacity available. At other points of time exactly this variant might be preferred because it might provide the highest quality (i.e. lowest cost) among all respective variants. Similarly a variant being dropped due to its high costs may be the preferred one at another point of time because it consumes sufficiently low processor capacity (due to its low utilization) to enable an overall balance of the system to be executed. This variation in decision to refuse or add a variant could refer to changes in the system resources, e.g., the processor capacity.

2- If the respective RTClassID exists in the array, this means that we are updating an existing column. If no variant is active in the existing column, we substitute it by the newly arrived one. If there is an active variant in the existing column, we add the newly arrived RTClass to a queue. We call it the UpdateQueue. The upper bound of ist capacity is equal to QB. Each element of the UpdateQueue is an array. The array includes one column, the newly deployed RTClass, in case this RTClass does not have dependencies. The array includes more than one column, the newly deployed RTClass and the other RTClasses in the dependency graph, in case a set of dependent RTClasses arrive.

Boundedness proof:

Uploading newly deployed RTCs to the local node:

Each (hyperperiod) only the newly deployed RTCs pending at the beginning of the hyperperiod are handeled and this set of newly deployed RTCs contains at most b elements, where b is a fixed given parameter. In addition we assume a deterministic communication channel between the remote node and the local one, i.e. the transmission of each request is handled in bounded time.

The Engine-cell applies a filter procedure when adding newly deployed RTCs to WorkingRTCArray. The procedure preserves the upper bounds of the WorkingRTCArray. It applies a set of steps. All of them consist of a bounded number of iterations, determined by one of the WorkingRTCArray dimentions. All dimensions are assumed to have a predefined upper bound which guarantees boundedness.

Abstract Code:

1.1 Gathering the newly deployed RTCs:

b for (number of newly deployed RTCs)

Const save the RTC in a message
Const Send the message to the Active Engine-Cell

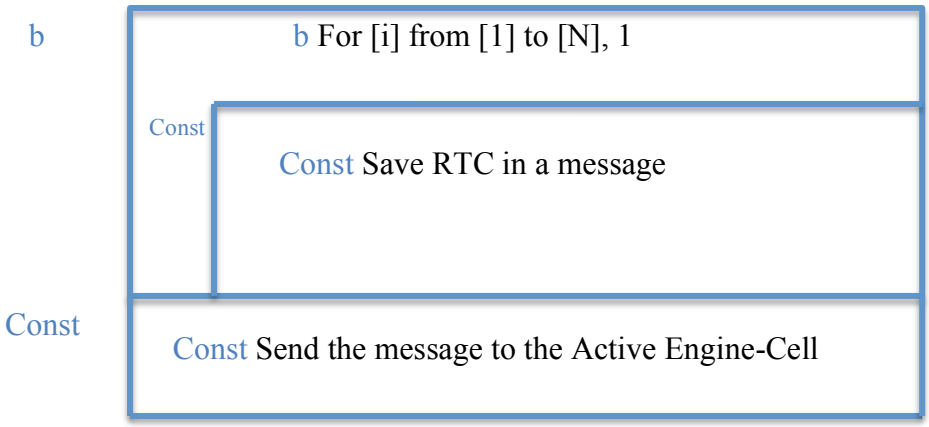


Diagram 1.1

Complexity of 1.1 is O((b ∗ Const) + Const) = O(b)

1.2 Filtering the newly deployed RTCs:

f ∗ h Copy RTCArray to WorkingRTCArray
b for (number of newly deployed RTCs){
O(b ∗ (log f + h ∗ f)) Adding a new variant, a new column to the WorkingRTCArray, or updating a column

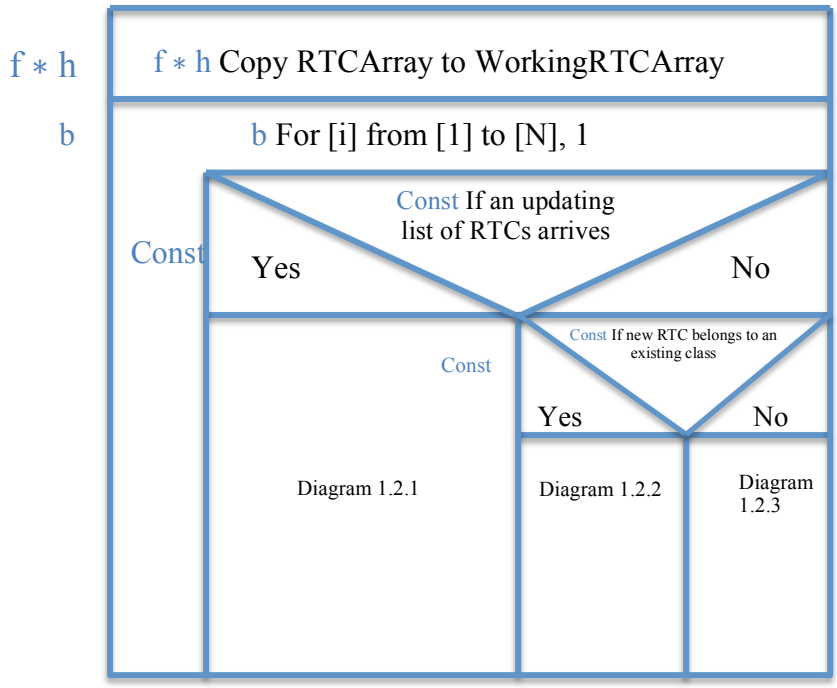


Diagram 1.2

Complexity of Diagram 1.2 is O(h ∗ f + b ∗ (max (Complexity of Diagram 1.2.1, max(Complexity of Diagram 1.2.2, Complexity of Diagram 1.2.3)))) = O(h ∗ f + b ∗ (max(h ∗ f,max(h + f, log f))) = O(h ∗ f + b ∗ h ∗ f) = O(b ∗ h ∗ f)

Const if an updating column (list of RTCs) arrives then{
h ∗ f if the updated column has an active variant
Const add the updating column to the UpdateQueue
Const else{
f Replace the updated column by the updating column
}}

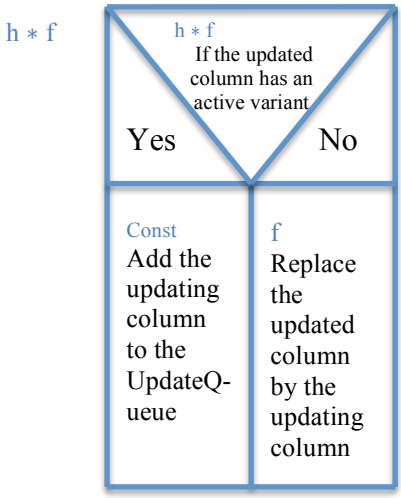


Diagram 1.2.1

Complexity of Diagram 1.2.1 is O(h ∗ f + max(Const, f)) = O(h ∗ f + f) = O(h ∗ f)

Const if RTC is adding a column then{
Const if there is a place in WorkingRTCArray for a new column
Const Add the column
Const else{
h Determine the triggered or essential classes
f Excludes the class with smaller ImportanceFactor
and add the newly arrived class.
}}

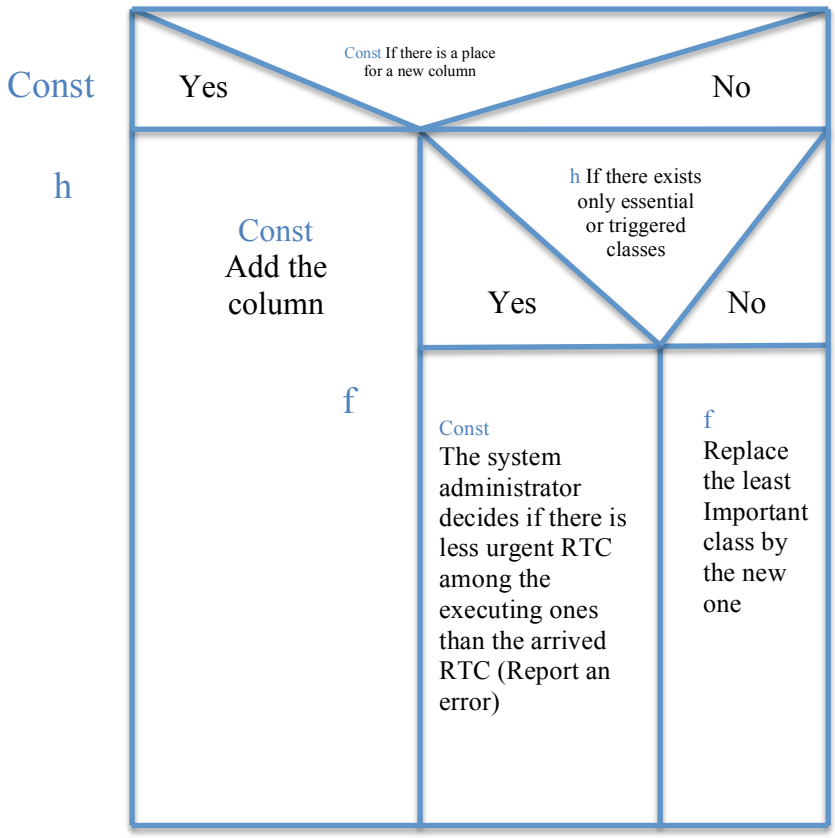


Diagram 1.2.3

Note: In Diagram 1.2.3, If there exists only essential or triggered classes, the system administrator decides if there is a less urgent RTC among the executing ones than the newly arrived RTC.
Whenever the system administrator has to decide something, this means an exception. I.e. the algorithm has to be left and some outside activitiy has to take place. From the point of view of the algorithm this is equal to an error.

Complexity of diagram 1.2.3 is O(Const + max (Const , h + max (Const,f)) = O(h + f)

Const if RTC is adding a variant then{
Const if there is a place in WorkingRTCArray for the variant
Const Add the variant
else{
if variant is aperiodic then{
if last element is not active then
log f insert the RTC at the proper location in the column with respect to the ordering by cost and drop the previously last element in this list.
else
log f insert the RTC at the proper location in the column with respect to

the ordering by cost and drop the element previous to the last one in this list.
}
else {

if last element is not active then
log f insert the RTC at the proper location in the column with respect to the ordering by utilization factor and drop the previously last element in this list.
else
log f insert the RTC at the proper location in the column with respect to the ordering by utilization factor and drop the element previous to the last one in this list.
}
}}}

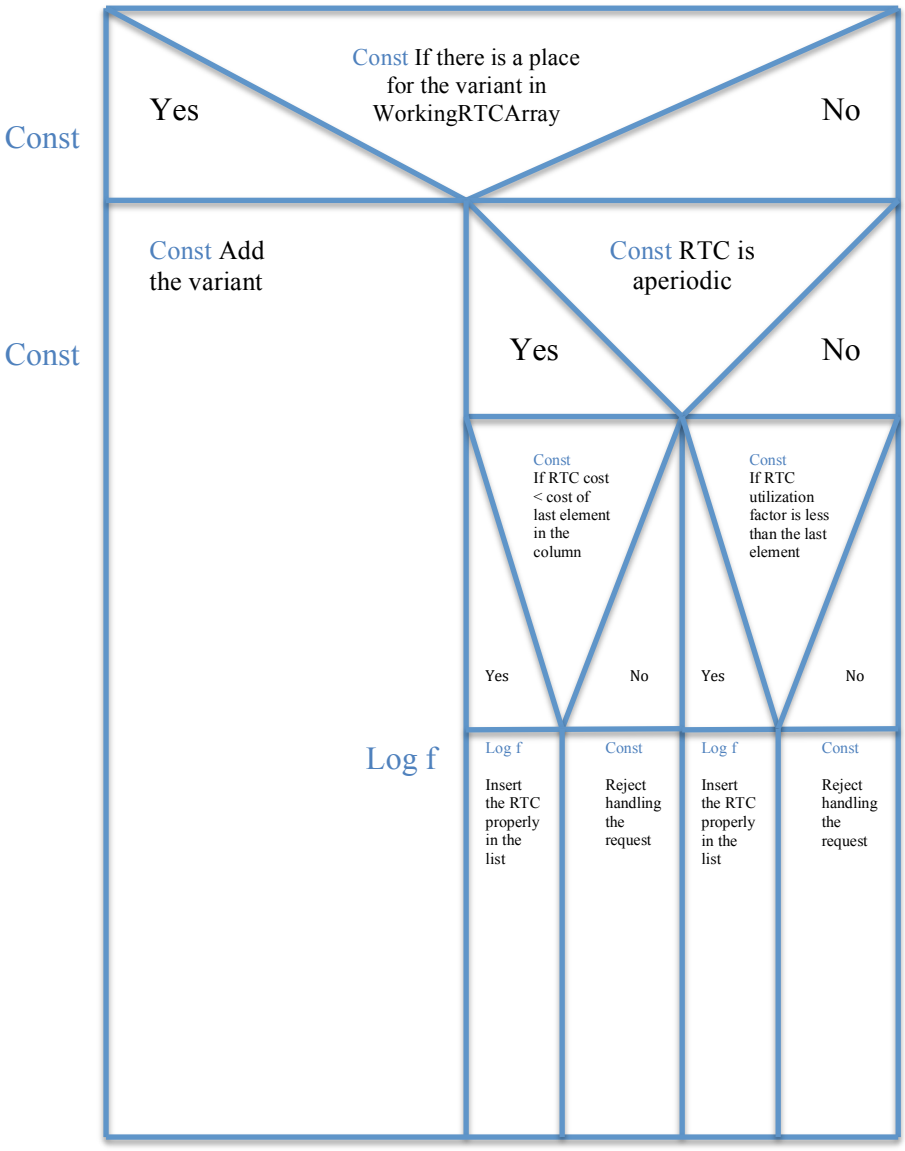


Diagram1.2.2

Complexity of diagram 1.2.2 is O(Const + max (Const, Const + max(Const + max(log f, Const),Const + max (log f,Const)))) = O(log f).

Complexity of 1.2 is O(b ∗ h ∗ f)

Complexity of step 1: Contribution of Diagram 1.1 + Contribution of Diagram 1.2 = O(b) + O(b ∗ h ∗ f)= O(b ∗ h ∗ f)

2- Triggering and handling the newly arrived requests:

We assume that the arrival frequency of requests is substantially smaller than the frequency of the Engine-cell ($1/WCT_{EC}$).

In the previous step, an UpdateQueue has been constructed. The update queue includes updating requests. In this step, another queue is constructed. It is called the TriggeredQueue. Requests that are added to this queue are chosen from the WorkingRTCArray. Triggered requests may set addition as well as deletion for RTCs. The Engine-cell makes a first iteration over items in the UpdateQueue and the TriggeredQueue in parallel. AEC selects either an update request or a triggered request. The decision depends on the priority of the request, which in turn might have been decided offline by the administrator of the system. For simplicity, the decision can be made arbitrarily.
If the number of chosen requests is less than the upper bound of requests, then another iteration is done on the UpdateQueue and the TriggeredQueue parallelly to choose requests, which were not chosen in the first iteration.
This iteration continues until the number of requests is equal to the upper bound or until no further requests exist.

Selected requests will be stored in a buffer called RequestQueue. The buffer size is constant.

Triggering a request from the WorkingRTCArray turns the Triggered Boolean property into true. The arrival time of a request, which is assigned when triggering the request, depends on the next time point where the current hyperperiod is completed (NHP). This point is a point where all accepted periodic requests are simultaneously activated. I.e. all periodic tasks are assumed to have a phase of 0.

More than one request can be handled by the AEC during one execution. If the arrival time of the periodic requests that have to be processed is not equal to the current NHP, then their TriggeringRange is examined. If

TriggeringTime ≤ NHP ≤ TriggeringTime + TriggeringRange, then the arrival time is set to the NHP. Otherwise the requests, which do not satisfy the previous condition, are not accepted and deleted from the buffer. After that a notification is sent to the system administrator.
The DeletionTime of the requests that have to be deleted is set to the next natural updating point.

The requests, which satisfy the condition, proceed to the adaptation algorithm. When these requests have been considered by the AEC, the buffers (UpdateQueue, TriggeredQueue, and RequestQueue) become empty again.

If we have aperiodic requests, and their arrival times are greater than NHP, they stay the same. If they are smaller than NHP, we set their arrival times the same way as for periodic requests. This is because the AEC can be preempted during its execution, and as a result a decision about the newly arrived requests might not take place before NHP.

We assume that the maximal number of requests that can be handled in this step is equal to the buffer size[15].

In case the request is a newly triggered set of dependent RTCs, or deleting a set of dependent ARTCs, a request can be accepted only if the entire set can be accepted.
When the request is adding a set of dependent RTCs triggered from the WorkingRTCArray, the request is loaded by saving a reference for the RTCs IDs in the buffer.

If the request includes a set of dependent cells, we assume that their modified arrival times and deadlines are calculated offline by EDF* [7]. When they arrive to the system, they are independent.
a_i = current time + $a_{i,modified}$, and d_i = current time + $d_{i,modified}$ for any of the dependent variants; i = 1..DEP_n. DEP_n is the number of dependent cells.
If $a_{modified}$ is smaller than the NHP, then $a_i = a_{i,modified}$ + FixedAmount, and $d_i = d_{i,modified}$ + FixedAmount; where FixedAmount = NHP - $a_{i,modified}$; i = 1..DEP_n. This offset is necessary.

An update request is represented by a list of RTCs that constructs the RTClass of the update.
The triggered RTC in this list is the one that is going to replace the running RTC, in case VariantsAllowed = false, otherwise a best variant

[15] Buffer size: is equal to the upper bound of requests QB. Buffer refers to the RequestQueue.

should be chosen from the list. Best variants are variants that provide sufficient utilization for meeting the deadlines with lowest costs.

Updating a set of dependent cells is done under the same rules as updating a cell. The updating cells are the triggered elements of the arriving lists.

Boundedness proof:

Loading the request:

By setting priorities, that the administrator of the system implicitly has already decided offline, either an updating request or a triggered request (add/delete) is chosen.

2.1- A queue of triggered requests (add/delete) is constructed.
2.2- A first iteration is done over the triggered and the UpdateQueue parallelly to decided whether to have an update request or a trigger/delete an RTC/RTCs request. The decision depends on the priority of the request, which in turn might have been decided offline by the administrator of the system. For simplicity, the decision can be made arbitrarily. If the number of chosen requests is less than QB, then another iteration is done on the UpdateQueue and the TriggeredQueue parallelly to choose requests, which were not chosen in the first iteration. This iteration continues until the number of requests in QB or until no further requests exist.
2.3- The newly decided request is stored in a variable dedicated for it.
2.4- The request type is then determined.
2.5- The arrival time of requests is set according to NHP.

There is a fixed upper bound QB of requests number. This is different from b, the upper bound of newly deployed RTCs. A newly deployed RTC has been added to the WorkingRTCArray. It can construct a request, only if it is triggered in this step.
An iteration 2.1 to 2.5 is done for an upper bound of times equal to the the upper bound of requests number. Each operation 2.1 to 2.5 is done in a bounded time. This proves the boundedness of the step.

Abstract Code:

In this step, the UpdateQueue and WorkingRTCArray are inputs. A TriggeredQueue is constructed. The output is a requests queue. Properties (Triggered, Deletion, and Arrival times) are set for the requests. Also types of requests are saved.

Const A request is assumed to be stored in an array, called RequestArray
// An array data structure is used here as the request may contain a set
// of dependent RTCs. Information about dependencies are collected
// from the property RelatedCells introduced in Solution4, Section 5.4.
// In case dependencies exist, each dependent cell represents a column
// in the array. A column is filled with the different variants of the cell

// Constructing the TriggeredQueue. A triggered request consists of
// RTCs that are chosen from the WorkingRTCArray. In order not to
// trigger the same request twice, we copy WorkingRTCArray into a
// temporary array, and then each time a request is triggerd, we remove
// the triggered classes from the temporary array. A random request can
// then be chosen from the temporary array.

h ∗ f Copy WorkingRTCArray into a temporary array

QB For QB times, repeat the following{

h ∗ SC Exclude triggerd columns from temporary array
Const Trigger a request randomly from temporary array }

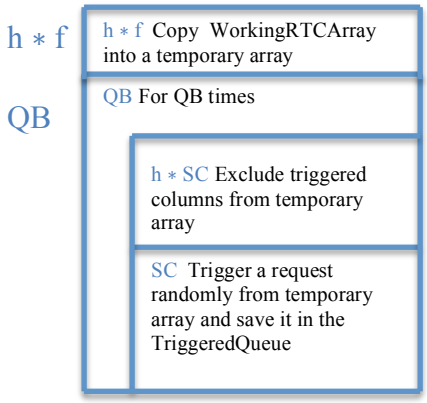


Diagram 2.1

Complexity of diagram 2.1 is O(h ∗ f + QB(h ∗ SC +SC)) = O(h ∗ f + QB ∗ h ∗ SC + QB ∗SC) = O(h ∗ f + QB ∗ h ∗ SC)

We iterate over the TriggeredQueue and classify the requests in it. A request is set as a deletion request if the selected RTC is active (running). A request is set as an adding request if the RTC is not active (not running). After that a triggering time is assigned to the request according to its type. Distinction between deletion requests and adding requests is necessary for constructing

ExpAARTCs and ExpPARTCs and for constructing AdaptationRTCArray (see step 4).

QB Iterate over the TriggeredQueue{
Const If the request is active then
Const Set it as a deletion request
Const If the request is not a deletion request then{
Const Set the triggered property to true
Const Assign a triggering time to it equal to the current time}}

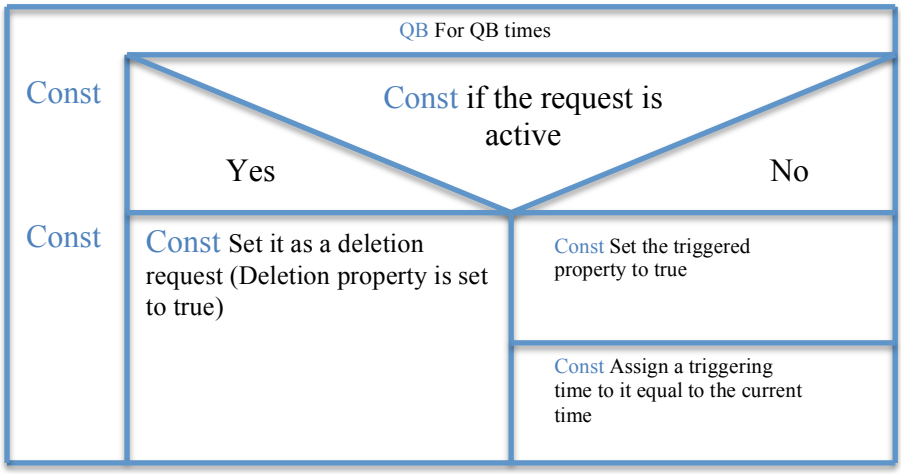


Diagram 2.2

Complexity of Diagram 2.2 is O(QB ∗ Const + max (Const, Const)) = O(QB)

QB For QB times do the following{

// QB is the upper bound of requests that could be processed at each
// execution of the AEC

Const Make a random decision whether to have the update or trigger a request from the WorkingRTCArray (making a random decision or priority decision between updating and triggering/deleting achieves an acceptable balance for serving the different types of requests).

O(SC) Set the triggered property of the request in WorkingRTCArray (Diagram 2.3.1)
O(1) Set arrival time of the triggered request (Diagram 2.3.1.1)
Const Store the selected request including its type in the RequestQueue
}

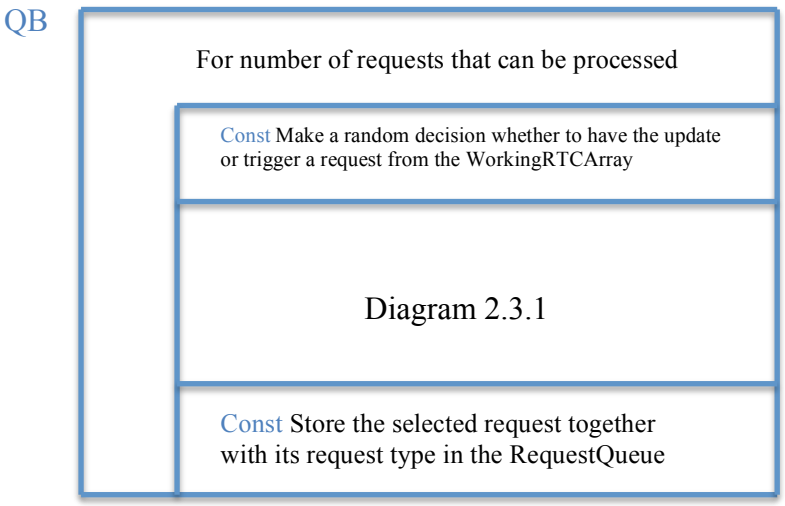


Diagram 2.3

Complexity of Diagram 2.3 $O(QB * (Const + Complexity\ of\ Diagram\ 2.3.1 + Const)) = O(QB * (Const + SC * h * f + SC^2 + Const)) = O(QB * (SC * h * f + SC^2))$

Here, we check the chosen request. If it is a triggered request, we set the triggered property for its counterpart constituents in WorkingRTCArray to true.

Const If the request is not a deletion request
{

SC For (number of related cells in the selected RTC)

Const Set the Triggered property for the request RTCs in
WorkingRTCArray to true
O(1) Set arrival time for the request

}

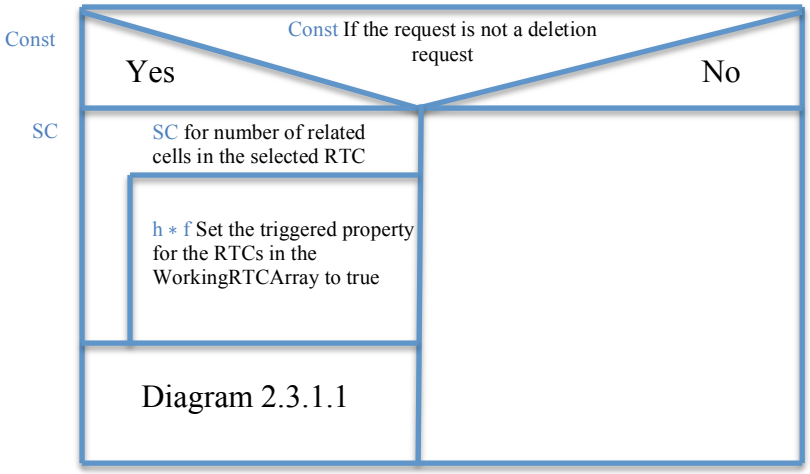


Diagram 2.3.1

Complexity of diagram 2.3.1 is $O(SC * h * f + SC^2)$

In case of a triggered request, periodic or aperiodic, we set the arrival time for it equal to NHP if (NHP <= TriggeringTime of the RTC + TriggeringRange of the RTC).

Const If number of RTCs in the request is equal to 1, then{
Const If Arrival time of the RTC is not equal to NHP, then
Const if (NHP <= TriggeringTime of the RTC + TriggeringRange of the RTC) then
Const Set Arrival time is equal to NHP
Else
Const Return "The newly arrived request is not accepted"}

Else{
// Set absolute arrival time and deadline for dependent RTCs[16]
SC For all dependent cells{
Const Absolute arrival time = current time + $a_{modified}$
Const Absolute deadline = current time + $d_{modified}$
}
$O(SC^2)$ Find smallest absolute arrival time
Const Calculate FixedAmount = NHP – smallest absolute arrival time
SC For all dependent cells{
Const Absolute arrival time = Absolute arrival time + FixedAmount

[16] The only case to have more than one RTC in a request is the case of having dependencies between RTCs

Const Absolute deadline = Absolute deadline + FixedAmount

}}

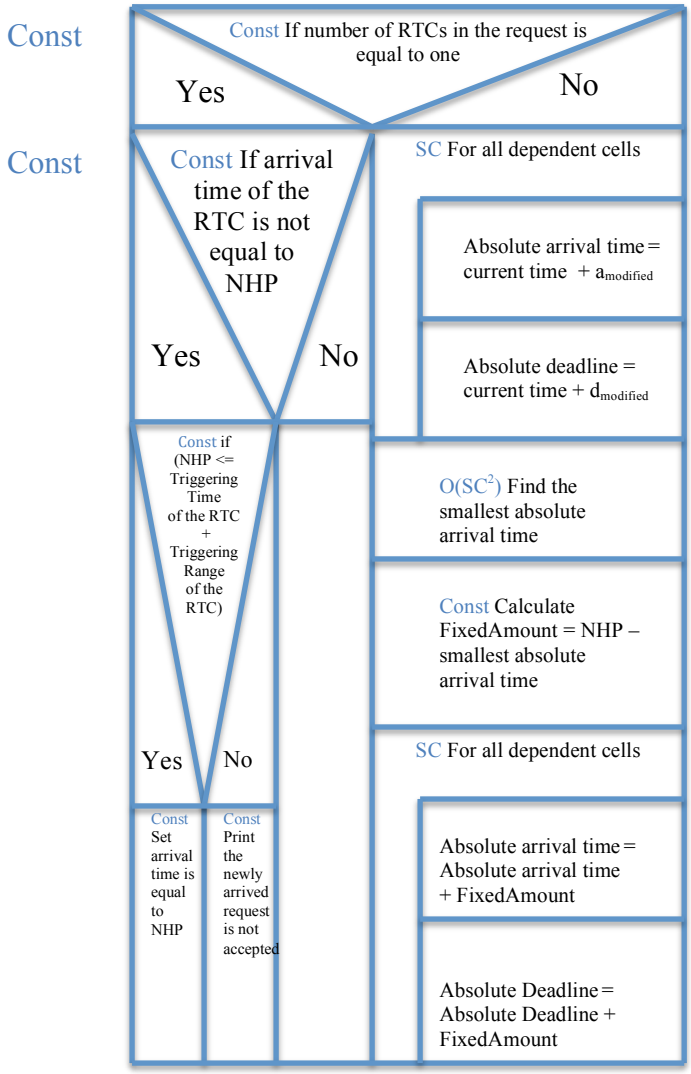


Diagram 2.3.1.1

Complexity of diagram 2.3.1.1 is O(Const + max(Const + Const + max(Const,Const), SC ∗ (Const + Const) + SC^2 + Const + SC ∗ (Const + Const))) = O(SC^2)

Const Set Request Type.

Complexity of step 2: Contribution of Diagram 2.1 + Contribution of Diagram 2.2 + Contribution of Diagram 2.3 = O(QB ∗ (h ∗ f ∗ SC + SC^2))

3- Calculating the cost of quality factors for the system:

The parameters of the system (could be cost parameters or other parameters that play a role in creating a new request) have to be read. A part or the whole set of local parameters might represent the quality factors available by the local node. Parameters of the system should be read in each execution of the Engine-cell because they might change. This change affects the result of the adaptation. E.g., adding new resources may allow accepting a set of requests, that is not accepted with less resources.
Quality factors available by a certain node are those, which are normally considered by cells (it means, play a role when calculating cost of cells). The total cost of factors available by a node is called $Cost_{total}$.

Boundedness proof:

Calculating the cost of quality factors for the system:

- The AEC calculates the total cost available by the system. This is done by iterating over a fixed finite set of parameters of the system. Summing the results of multiplication of each parameter amount and weight constructs the total cost of the system. The number of parameters is bounded. This allows to prove the boundedness of this step.

Abstract Code:

O(PN) The total cost is calculated by some function based on PN parameters.

Complexity of step 3: O(PN)

4- Adaptation algorithm:

In this step, we calculate the lowest-cost feasible solution over the entire set of RTClasses stored in AdaptationRTCArray[17] introduced in the following (This is under the assumption, that the periodic alternatives will substitute the current periodic ARTCs on the NHP, and the aperiodic alternatives will substitute the current aperiodic ARTCs on the next possible updating point).

[17] AdaptationRTCArray: is an array of RTCs, on which we solve a knapsack problem, so that we choose an RTC from each column. The chosen RTCs are the ones that will execute in the next hyperperiod.

NHP is the end of the current AEC execution/beginning of the next one. The results of the current AEC execution are guaranteed to be ready not earlier than NHP. For this reason, ARTCs may be substituted at the next possible updating point only when such updating points which lay after NHP are considered.

Constructing AdaptationRTCArray:

- We copy the variants of the WorkingRTCArray into a temporary array AdaptationRTCArray.
- We then reduce AdaptationRTCArray to contain only the RTCs variants, which RTClassID exists in the ExpPARTCs and ExpAARTCs with absolute deadlines exceeding NHP. For each ExpPARTC or ExpAARTC, that has the property VariantsAllowed set to false, we don´t consider variants that hold the same RTClassID in AdaptationRTCArray, other than the ARTC itself.
- For each aperiodic ARTC that should be deleted (stored in the requests buffer), and has absolute deadlines exceeding NHP, we add a column including the ARTC as the only variant. If a next possible updating point exists, its execution time is set to $y_{UpdatingPoint}$. The mentioned updating point is the next possible updating point. The reason is that updating points are the most suitable to apply deletion, as partial results are delivered on these points. Deleting a cell suddenly on an arbitrary point may cause errors.
- We then add a column that includes the AEC as a periodic cell.
- We also add the newly triggered requests. If their properties VariantsAllowed are set to true, we add columns that represent RTClasses equal to the RTClasses of the newly triggered variants. If, however, their properties VariantsAllowed are set to false, we add only columns containing the newly triggered variants (a column for each RTC). The value of VariantsAllowed might be different among the different requests.

- In case there is an update request for a periodic or aperiodic RTC,

  - Adding an aperiodic update is done (only if there exists an updating point after NHP in the aperiodic variant that is running) by adding the arrived RTClass which includes the triggered updating variant. If VariantsAllowed is equal to true, the class contains all variants which belong to this update. Otherwise, it contains only the triggered updating variant. The replacement is done at the next updating point (after NHP) of the running updated variant. In the following, we summarize how to check the existence of an updating

point after NHP (Only in this case, the updated variant should be excluded when constructing ExpAARTCs), and how to set the time characteristics for the variants in the updating column:

First (Determining the set of ARTCs that can be updated):

Amount = Hyperperiod – $[(\sum_{i=1}^{NumOfPARTCs}((\text{Hyperperiod } /T_i) * C_i)) + \text{WCET}_{EC} + \sum_{i=1}^{NumOfAARTCs-NumofANHP}(C_i - ET_{executed_i})]$[18]

Amount is the time left in the current hyperperiod, after excluding the time that should be spent in executing the periodic ARTCs, and aperiodic ARTCs, which deadlines do not exceed NHP[19]. Amount1 = Amount.

If (Amount1 > 0) then {

- For (i = 1 to NumofANHP){

If ($C_i - ET_{executed_i} \leq$ Amount1) then{
Amount1 = Amount1 – ($C_i - ET_{executed_i}$)

}

else{
$C_{i,new}$ = ($C_i - ET_{executed_i}$) – Amount1.

If there exist an updating point in $C_{i,new}$
Add this ARTC to the set of variants that can be be updated
}
}}

else{
For (i = 1 to NumofANHP){
$C_{i,new} = C_i - ET_{executed_i}$

[18] By $ET_{executedi}$ is denoted the accumulated execution time spent during hyperperiods prior to the current one. It is assumed that this value is stored in the respective task control block.
[19] Amount will be needed later when calculating time characteristics for the running aperiodic ARTCs that are stored in AdaptationRTCArray, and deadlines exceed the NHP. In part "first" this value will me modified. Therefore here we work on an intermediate copy Amount1.

If there exists an updating point in $C_{i,new}$
Add this ARTC to the set of variants that can be updated

}}

Second (Calculation of time characteristics for the updates):

If the found updating point is UP[x,y], the arrival time of the j_{th}[20] variant is set to the arrival time of the updated variant. The execution time for the j_{th} variant is set to $(y + C_j – y`)$, where y` is the relative updating point time for the counterpart updating point. The specified absolute deadline for the j_{th} variant is set to max[$((D_j – y`) +$ (Arrival time of the running variant + y)), AbsoluteDeadline of the running variant that should be updated]

- Adding a periodic update is done by adding the arrived RTClass, which includes the triggered updating variant to AdaptationRTCArray. If VariantsAllowed is equal to false, only the triggered updating variant exists in the column. Otherwise, all variants which belong to the update exist in the column. The updated variant has to be excluded when constructing ExpPARTCs, because executions of periodic instances are completed in each hyperperiod. This means, that when periodic update is applied in the next hyperperiod, we do not assume any execution of the updated variant.

- In case there is an update request for a set of aperiodic dependent RTCs, we assume that the dependent sets have their modified and arrival times and deadlines calculated offline. When calculating the time characteristics of the variants in the columns that are updating dependent variants, the following is applied:

If the found updating point after NHP is UP[x,y], the arrival time of the j_{th} variant is set to the arrival time of the updated variant. The execution time for the j_{th} variant is set to $(y + C_j – y`)$, where y` is the relative updating point time for the counterpart updating point.The specified absolute deadline for the j_{th} variant is set to max[$((D_j – y`) +$ (Arrival time of the running variant + y)), AbsoluteDeadline of the running variant that should be updated].

---

[20] j_{th}: j indicates the index of the variant in the updating column.

In this way, we can use the reduced array in the next step for the adaptation algorithm as each column represents a participant in the test.

Let us assume that:

- The number of columns in AdaptationRTCArray = Num.
- N` is the number of columns, which represent the newly triggered aperiodic requests.

If (NumOfANHP > 0) then {

- We calculate the arrival times, execution times, and Cost_Update for the running aperiodic ARTCs that are stored in AdaptationRTCArray, and deadlines exceed the NHP as follows:

  - New execution time is assigned to the variants:

    Amount as calculated under part "First" is the time left in the current hyperperiod, after excluding the time that should be spent in executing the periodic ARTCs, and aperiodic ARTCs, which deadlines do not exceed NHP.

    If (Amount > 0) then {
    - We construct a vector V of the running aperiodic ARTCs, which deadlines exceed NHP.

    - We order the elements of this vector according to the increasing absolute deadlines.

    - For (i = 1 to NumOfANHP){

      If ($C_i - ET_{executed_i} \leq$ Amount) then {
      Amount = Amount – ($C_i - ET_{executed_i}$)
      Exclude the column of the i_{th} aperiodic variant from AdaptationRTCArray.
      Decrease Num by 1.
      }
      else{
      $C_{i,new}$ = ($C_i - ET_{executed_i}$) – Amount.

Set the execution time of the variant in AdaptationRTCArray that is equal to the i_{th} variant to $C_{i,new}$.

Set the arrival time of the variant in AdaptationRTCArray that is equal to the i_{th} variant to Arrival time = NHP, if Arrival time < NHP

Amount = 0.
}
}
}

else{

For (i = 1 to NumofANHP){
$C_{i,new} = C_i - ET_{executed_i}$

Set the execution time of the variant in AdaptationRTCArray that is equal to the i_{th} variant to $C_{i,new}$.

Set the arrival time of the variant in AdaptationRTCArray that is equal to the i_{th} variant to Arrival time = NHP, if Arrival time < NHP

}}

- Cost_Update = the cost of the RTC

- The following iteration is done over the aperiodic RTCs in AdaptationRTCArray, which do not belong to the newly triggered requests:

  For ($k = Number\ of\ periodic\ columns + 1..Num - N$`){

  If (VariantsAllowed = true) && (there exists an updating point in the part of the running variant dedicated for $C_{i,new}$ (after NHP)) then{

  The following calculations are done to include the possible alternatives for the active aperiodic cells in the knapsack problem:

- a new arrival time, execution time, cost and absolute deadline is calculated for the variants of the k_{th} column in AdaptationRTCArray, excluding the running variant in the k_{th} column:

  - Arrival time = Arrival time of the active variant in the k_{th} column.

  - New execution time is assigned to each variant in the k_{th} Column, excluding the running variant:

    $C`_{k,new} = (C` - y`) + (C_{k,new} - (C_{running,variant} - y))$

    C` is the execution time of the j_{th} variant, for which we are calculating the attributes, in the k_{th} column.
    $C_{k,new}$ is the calculated execution time of the running variant in the k_{th} column.
    $C_{running,variant}$: is the original execution time of the running variant in the k_{th} column.
    y is the relative updating point time of the next updating point in the running variant.
    y` is the relative updating point time of the counterpart updating point in the j_{th} variant.

  - $Cost\_Update_j$ = Maximum of (Cost of the running variant, cost of the j_{th} variant).

  - Specified absolute deadline = max (Specified absolute deadline for the running variant, NHP + ($C_{k,new}$ – ($C_{running,variant}$ – y)) + specified relative deadline).

  }
else
We choose the running variant in the k_{th} column.
  }
}

To find the solution, we solve the following multiple choice multi dimensional knapsack problem:

$$\text{Max } \sum_{i=1}^{Num} \sum_{j=1}^{n_i} -Cost_{ij} x_{ij}$$

Subject to: $\sum_{i=1}^{Num} \sum_{j=1}^{n_i} W_{ij}^k \, x_{ij} \leq R^k$

Where: $\sum_{j=1}^{n_i} x_{ij} = 1; i = 1..m \; \& x_{ij} \in \{0,1\}$; i = 1..m and j = 1..$n_i$, k = 1:3

- $W_{ij}^1 = Factor_1 / Factor_2$

For any of the periodic RTCs: $Factor_1 = C_{ij}$, $Factor_2 = T_{ij}$
For the AEC, $Factor_1 = WCET_{ECTEMP}$, $Factor_2 = WCT_{ECTEMP}$

The expected hyperperiod is calculated as the least common multiple of the periods of the periodic ExpPARTCs in AdaptationRTCArray, and the periods of the newly triggered periodic requests in AdaptationRTCArray. The resulting value is set as initial value for the AEC's expected period. If the resulting utilization of the RTCs is below 1 then, we examine the total utilization (AEC and RTCs). If it is smaller or equal to 1, we have found the shortest possible expected period for AEC (which at the same time by definition is the hyperperiod). If the total utilization is beyond 1 then the expected hyperperiod has to be extended by a harmonic multiple until the total utilization is no longer beyond 1.
If the resulting utilization of the RTCs is 1, the set of chosen RTCs results in a non-feasible solution.

In each hyperperiod, only one execution of the AEC is assumed. For this reason, we finally update WCT_{ECtemp}, the expected period of the AEC, to be equal to the expected hyperperiod.

For any of the aperiodic RTCs: $Factor_1 = 0$, $Factor_2 = 1$

- $W_{ij}^2 = Factor_1 - Factor_2$

For any of the periodic RTCs and the AEC: $Factor_1 = 0$, $Factor_2 = 0$
For any of the aperiodic RTCs: $Factor_1 = d_{Specified,ij}$, $Factor_2 = d_{Calculated,ij}$

Where:

$d_{Specified,ij}$: The specified absolute deadline for any aperiodic variant, which belongs to an aperiodic variant in AdaptationRTCArray is equal to its arrival time + relative deadline of the variant.

$d_{Calculated,ij} = \max\{d_{Calculated(i-1)j_{i-1}}, ArrivalTime_{ij}\} + C_{ij,new}/Us$.
$d_{Calculated(Number\ of\ priodic\ columns\ in\ Array1+1)j_{Number\ of\ periodic\ columns\ in\ Array1+1}} = 0$.
$U_s = 1-U_p$.

Depending on the different kinds of RTCs to be considered in solving the Knapsack problem, W_{ij} is defined as follows:

- W_{ij}^3 = Cost for periodic RTCs stored in AdaptationRTCArray

W_{ij}^3 = Cost_Update for running aperiodic RTCs that are stored in AdaptationRTCArray

W_{ij}^3 = Cost for added aperiodic RTCs stored in AdaptationRTCArray

$$R^1 = 1$$
$$R^2 = 0$$
$$R^3 = \text{Cost}_{\text{total}}$$

The limit $\text{Cost}_{\text{total}}$ is optional. If it is set to infinity, then the optimization process tries just ti find the lowest-cost solution. If the limit is set to a finite value, then the solution space is further limited.

If a solution is found, the newly arrived requests are accepted.

If the newly arrived requests are accepted by the system, the ARTCs set or subset (which is represented in AdaptationRTCArray) is set to be modified and to be substituted by the chosen alternatives (Replacing a periodic ARTC means deleting the periodic ones that should be substituted and loading the periodic alternatives at NHP. Replacing an aperiodic ARTC means, the replaced RTC can be treated as a deletion request. When the deletion takes place, the information necessary for replacing the ARTC (transferred from replaced RTC to the replacing one to) should be stored. This can be differentiated from a normal deletion request by comparing the RTClassID of the RTC to be deleted with other RTClassIDs in the ready queue. If there exists an identical RTClassID, then it is replacement process. The comparison is done by the scheduler, and the replacement is done at the next updating point that happens after the NHP).
The chosen alternatives are stored in a ready queue. At the NHP, the Active property of the alternatives and for the newly triggered requests is set to true. The Active property of the alternated cells is set to false once they are replaced (deleted). In the aperiodic case, the part of the updated cell that follows the first updating point after NHP is to be replaced.
At the replacement point for aperiodic cells, any data of the altered cells or updated cells that might be necessary for the alternatives or updating variants is stored.

The Active property becomes true for the alternatives. After that, step 6 is

applied.

The AEC takes care also that the deletion of ARTCs takes place before any triggered request that happens at the same time.

Boundedness proof:

The objective of this step is finding the best solution by choosing appropriate variants for currently executing RTCs, and requests. If a solution is found, the accepted requests are activated, and the properties of the Engine-Cell are updated.

Here we are solving a knapsack problem. The algorithm, which we are applying is a genetic algorithm. In the algorithm, an individual contains exactly one variant for each column in AdaptationRTCArray. And a generation may contain one or more individuals.
In total there exist up to f^h individuals. Each of them is a potential solution of the Knapsack problem. In the Genetic algorithm to solve this Knapsack problem we select smaller subsets of individuals and call them Generations. The lowest-cost individual of a generation is a preliminary solution of the Knapsack problem. A generation is constructed from a previous one by applying selection and mutation. This process is iterated until no improvement can be observed or a given time limit is reached.
We set the first generation to include at least two individuals. The first one is given by selecting from each RTClass the variant with the lowest respective utilization (defining utilization for aperiodic RTCs as C/D).The second one is given by the current selection of variants for all RTClasses which are not affected by the adaptation together with with all adaptation requests. The first initial individual allows a simple decision whether a solution exists, as if this individual does not fulfill the constraints then there cannot exist any solution. The second initial individual is a promising one in the first generation under the assumption that before adaptation we had an optimized system.

Let us assume that the number of individuals in a generation ≤ upper bound of number of RTCs in a class in the WorkingRTCArray. Any other bound would work as well. The remaining individuals of the first generation may be chosen by any procedure, e.g. by randomly exchanging the selected variants in the columns.

If a column includes only one RTC because its VariantsAllowed =

false, then we have to choose exactly this element to be part of the currently constructed individual.

After that WCT_{ECTemp}, server utilization, and absolute deadlines for aperiodic load are calculated for each individual according to TBS. The individuals of a generation are sorted by increasing total costs. This implies that the first individual of this list, provided that the constraints are satisfied, constitutes the preleminary optimum. The previous operations are bounded by NInd, upper bounds of RTCArray dimensions, and the given time bound for the iteration.

If the knapsack constraint $\sum_{i=1}^{Num} \sum_{j=1}^{n_i} W_{ij}^{k} x_{ij} \leq R^{k}$ (See Section 5.2) has no solution for the first generation, even under the assumption of R3 = infinite, then the adaptation has to be rejected. Otherwise, if it has a solution for a set of individuals, we choose as an intermediate solution the individual which minimizes the accumulated cost of the chosen RTCs.

In order to potentially improve the solution with the objective to minimize the accumulated cost, we iterate to choose different generations by applying selection and mutation on the individuals, until we either have no further improvement or we reach our predefined time limit.

As an example we assume that the selection process is done by rejecting all constraint-violating individuals and a certain amount of the worst individuals of a generation and that the mutation process is done by replacing an arbitrary RTC in the remaining individuals by another arbitrary RTC from the same column. Selection also implies that the size of the generations may vary (remains bounded).
We can decide whether there exists a feasible solution or not in bounded time. Feasibility can be decided already based on the first generation. The optimization is done in bounded time as well. The reason is that we loop from generation to generation until we either have no further improvement or we reach our predefined time limit. The latter termination condition guarantees boundedness.

Abstract Code:

h For (all elements in active periodic RTCs)
Const Assign element to ExpPARTCs
h For (all elements in active aperiodic RTCs)
Const Assign element to ExpAARTCs

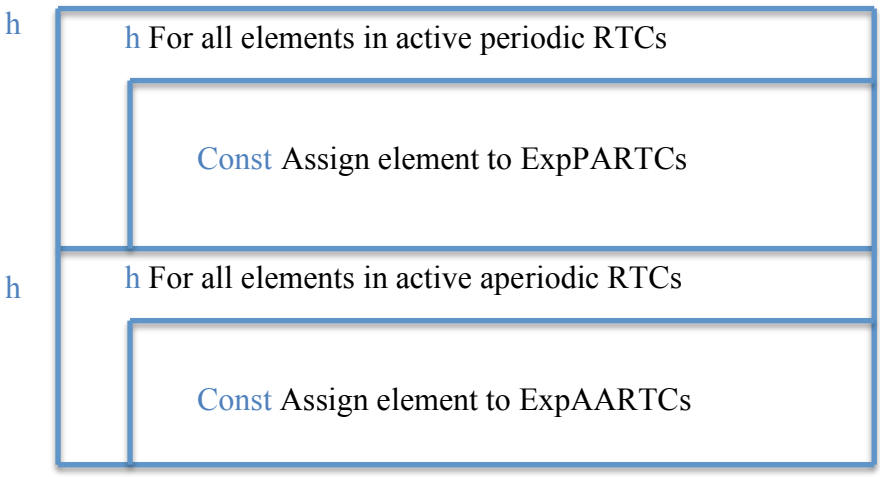


Diagram 4.1

Complexity of diagram 4.1 is O(h)

// In the following we exclude the deletion requests from the lists
// ExpAARTCs and ExpPARTCs. These lists are required to construct
// AdaptationRTCArray

QB For (all requests in the buffer){
SC Iterate over elements in each request[21]{
Const If the element is a periodic RTC, then{
h For (all elements in ExpPARTCs)
Const If (the ClassID of the element is equal to the classID in the current element of the ExpPARTCs list), then
Const Remove the element from ExpPARTCs
}
else{
h For (all elements in ExpAARTCs)
Const If (the ClassID of the element is equal to the classID in the current element of the ExpAARTCs list), then
Const Remove the element from ExpAARTCs
}}}

[21] A request may consist of a set of dependent cells, their number is bounded by SC.

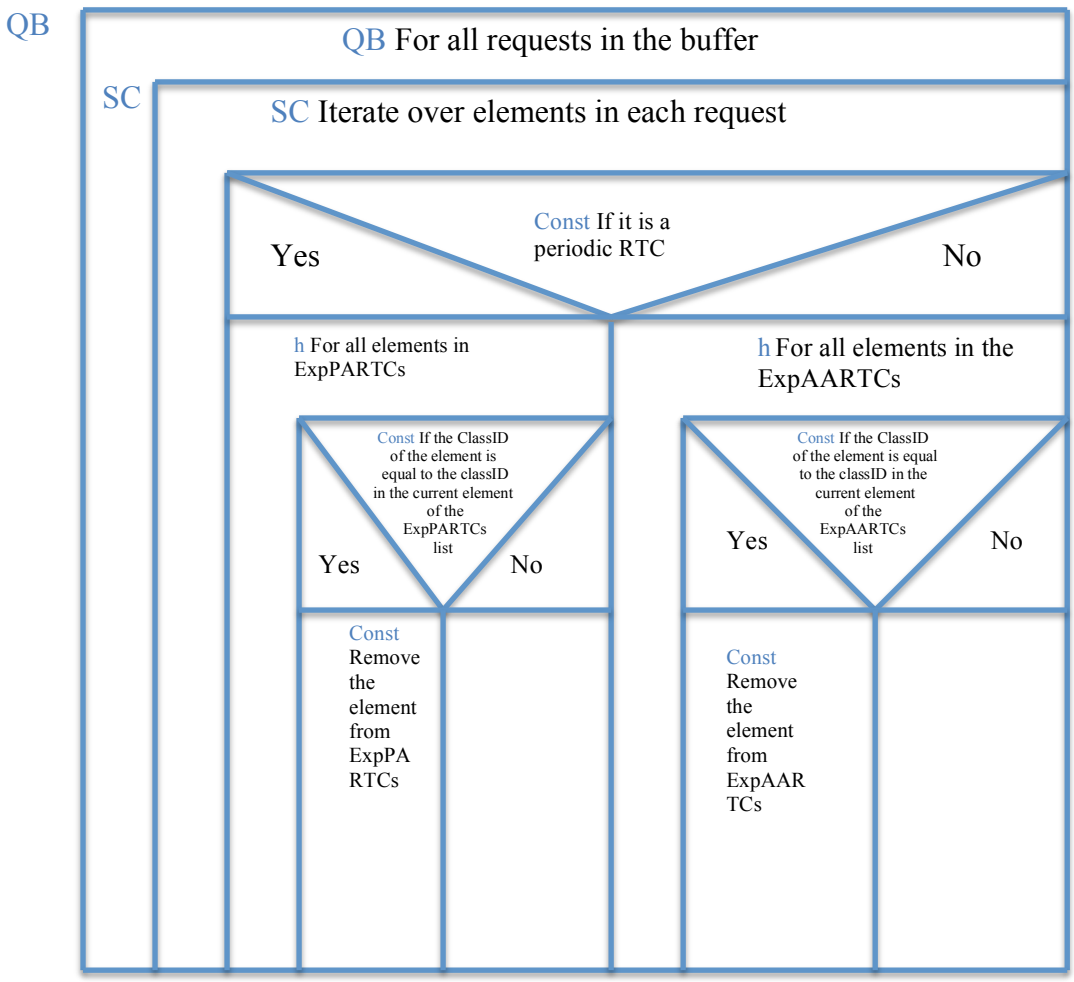


Diagram 4.2

Complexity of diagram 4.2 is O(QB ∗ SC ∗ Const + max (h ∗ Const, h ∗ Const)) = O(QB ∗ SC ∗ h)

h ∗ Const Remove all elements in ExpAARTCs, which has AbsoluteDeadline smaller or equal to NHP

h ∗ QB ∗ Const Remove the variants that should be updated from ExpPARTCs

// The complexity is h ∗ QB, because h is the upper bound of
// elements in ExpPARTCs, QB is the upper bound of update requests.
// Here a comparison of IDs is done between ExpPARTCs and
// requests queue, in order to determine the RTCs that should be
// updated in ExpPARTCs.

// The following code finally constructs the AdaptationRTCArray, on
// which the problem is solved, and remove the variants that should be
// updated from ExpAARTCs and from AdaptationRTCArray

$h * f$ Copy the variants of the WorkingRTCArray into a temporary array AdaptationRTCArray.
$h^2 * f$ Reduce AdaptationRTCArray to contain only ExpPARTCs and ExpAARTCs with absolute deadlines exceeding NHP
// Iterating over AdaptationRTCArray has complexity $h * f$. Iterating
// over ExpPARTCs or ExpAARTCs has complexity h

$QB * SC$ Add Deletion requests (only one running variant in each column)
Const Add AEC
$QB * f * SC$ Add the newly triggered requests
QB For any update{
Const If the update is aperiodic{
$O(n)$[22] Check if there exists an updating point after NHP in the running updated variant
Const If the updating point exists{
$h * SC * f$ Calculate time characteristics for the updating classes
// Calculating the execution time left in this hyperperiod need h
// complexity because it iterates over AARTCs (running aperiodic
// RTCs). Compexity $SC * f$ denotes the upper bound of RTCs in an
// update request and the different variants for each RTC
}}

Const If the update is periodic then
f Add periodic update requests
}

$(h+1) * f$ Remove the columns of the updated variants from AdaptationRTCArray
// The updated variants become known from their ClassID
// h+1 is the upper bound of columns in AdaptationRTCArray:
// h is the upper bound of classes in WorkingRTCArray. The
// additional one is for AEC.

$h * (h+1) * f$ Setting the time characteristics for the running AARTCs, with deadlines exceeding NHP in AdaptationRTCArray
// h+1 is the upper bound of columns in AdaptationRTCArray
// h is the upper bound of elements in AARTCs
// f is the upper bound of elements in a column in
// AdaptationRTCArray

---

[22] n: is the upper bound of updating points in a cell.

// To set time characteristics for the running AARTCs variants, if with
// deadlines of AARTCs exceeding NHP, we first have to find the
// counterpart upating point in the variants other than the
// running one, then we make the calculations of time characteristics

$O(h * n)$ Find the counterpart upating point in the variants other than the running one

$h * (h+1) * f$ Setting time characteristics for the running AARTCs, with deadlines exceeding NHP in AdaptationRTCArray

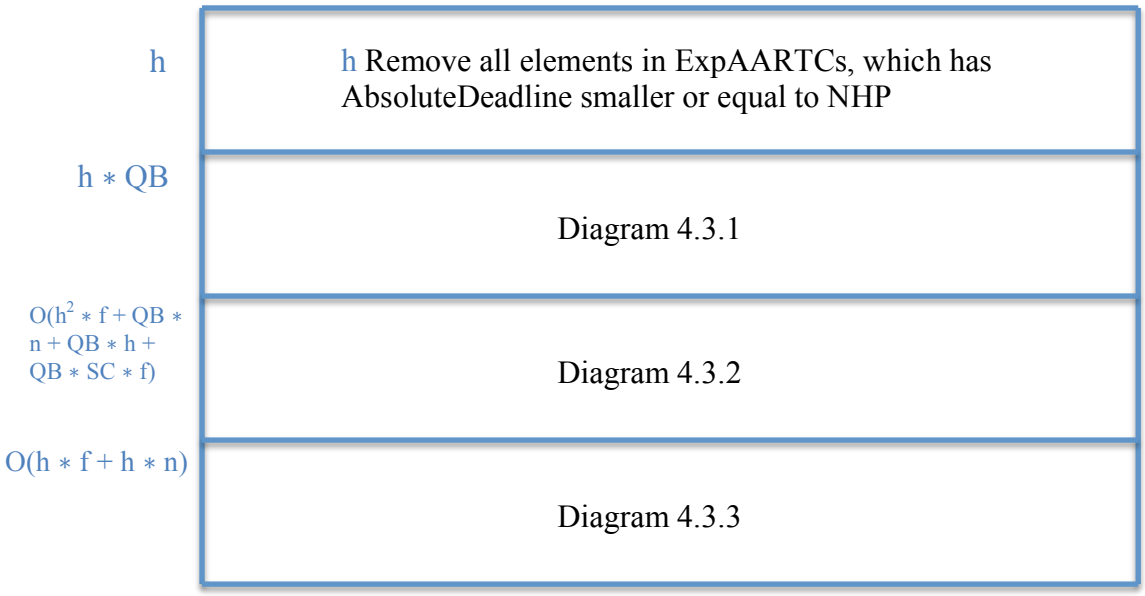


Diagram 4.3

Complexity of diagram 4.3 is O(h) + Complexity of diagram 4.3.1 + Complexity of diagram 4.3.2 + Complexity of diagram 4.3.3 = $O(h) + O(h * QB) + O(h^2 * f + QB * SC * f * h + QB * n) + O(h * f + h * n) = O(h + h * QB + h^2 * f + QB * n + QB * SC * f * h + h * f + h * n) = O(h^2 * f + QB * n + QB * h + QB * SC * f * h + h * n)$

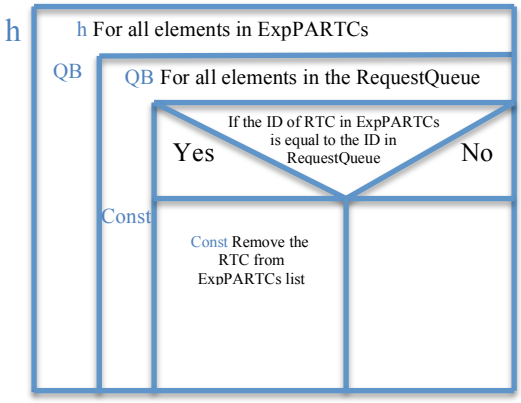


Diagram 4.3.1

Complexity of Diagram 4.3.1 is $O(h * QB * (Const + Const)) = O(h * QB)$

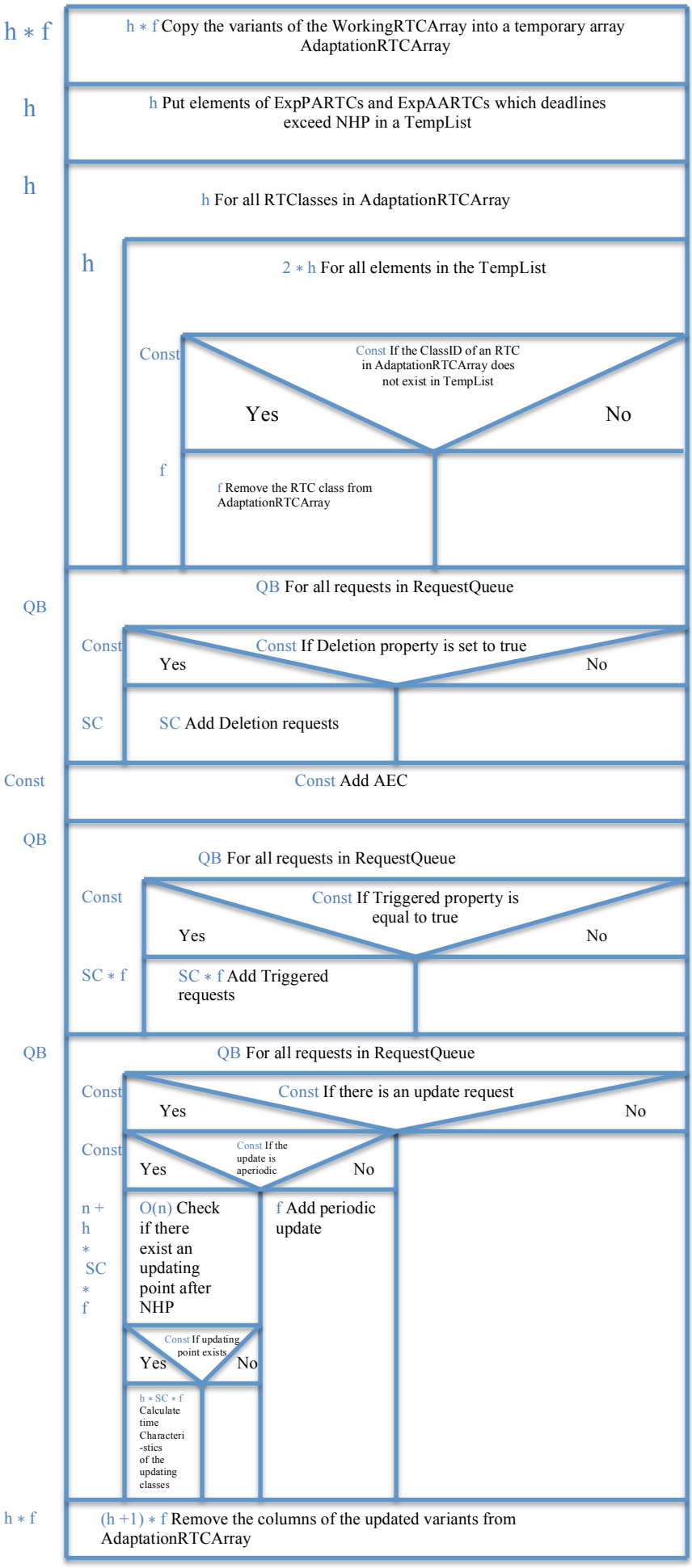

h ∗ f
h ∗ f Copy the variants of the WorkingRTCArray into a temporary array AdaptationRTCArray
h
h Put elements of ExpPARTCs and ExpAARTCs which deadlines exceed NHP in a TempList
h
h For all RTClasses in AdaptationRTCArray
h
2 ∗ h For all elements in the TempList
Const
Const If the ClassID of an RTC in AdaptationRTCArray does not exist in TempList
Yes
No
f
f Remove the RTC class from AdaptationRTCArray
QB
QB For all requests in RequestQueue
Const
Const If Deletion property is set to true
Yes
No
SC
SC Add Deletion requests
Const
Const Add AEC
QB
QB For all requests in RequestQueue
Const
Const If Triggered property is equal to true
Yes
No
SC ∗ f
SC ∗ f Add Triggered requests
QB
QB For all requests in RequestQueue
Const
Const If there is an update request
Yes
No
Const
Const If the update is aperiodic
Yes
No
n + h ∗ SC ∗ f
O(n) Check if there exist an updating point after NHP
f Add periodic update
Const If updating point exists
Yes
No
h ∗ SC ∗ f Calculate time Characteri -stics of the updating classes
h ∗ f
(h +1) ∗ f Remove the columns of the updated variants from AdaptationRTCArray

Diagram 4.3.2

Complexity of Diagram 4.3.2 is $O(h * f + h + h^2 * (Const + f) + QB * (Const + SC) + Const + QB * (Const + SC * f) + QB * (Const + Const + \max(n + h * SC * f, f) + h * f) = O(h^2 * f + QB * SC + QB * SC * f + QB * (n + h * SC * f)) = O(h^2 * f + QB * SC * f + QB * n + QB * SC * f * h) = O(h^2 * f + QB * SC * f * h + QB * n)$

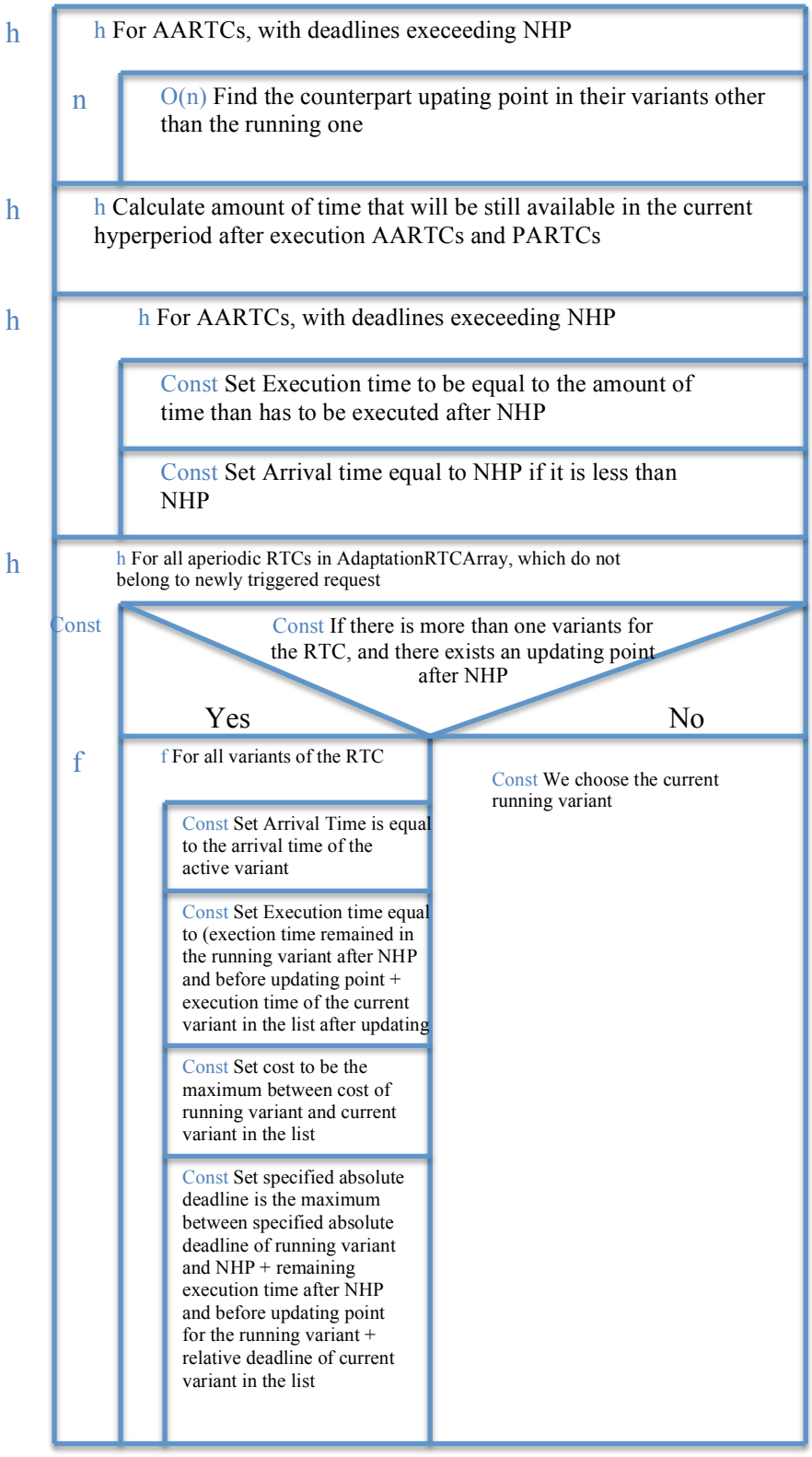


Diagram 4.3.3

Complexity of Diagram 4.3.3 is O(h ∗ n + h + h ∗ (Const + Const) + h ∗ (Const + max (f, Const))) = O(h ∗ f + h ∗ n)

a) setting the search space for the optimization

// In the first generation there are at least two individuals. As first
// individual we take the, one which is formed by selecting the lowest
// utilization variant in each column. This individual then can be tested
// immediately whether is fulfills the constraints (with W_{ij} set to
// infinity). If so, we have a valid solution (probably far away from being
// optimal but by following this strategy we surely have an "Anytime
// Algorithm"). If it does not fulfill the constraints then we know that no
// solution can exist and we can refuse the requested adaptation. As
// second individual,we choose this one, which keeps the variant selection
// of all columns which are not affected by the requested adaptation and
// assumes accepting all adaptation requests. Under the assumption that
// the system before adaptation is (nearly) optimized, this is a very
// promising individual in the first generation.

// Let us start by choosing the first two individuals

h For (all columns in AdaptationRTCArray){
f For (all elements in the respective column){
Const Choose the RTC with the smallest utilization and store it in first individual}}
Const Add first Individual to initial generation

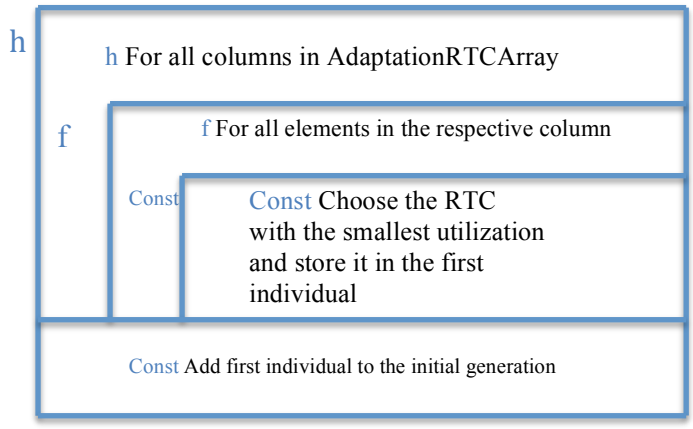


Diagram 4.4

Complexity of diagram 4.4 is O(h ∗ f ∗ Const + Const) = O(h ∗ f)

// Let us evaluate the first individual. The following calculations is
// done on RTCs in the individual

```
Const Set Periodic utilization = 0
Const Set aperiodic lateness = 0
h for all periodic RTCs in the individual
Const Periodic utilization = Periodic utilization + utilization of RTC
h + m1 + log GRP Calculate the expected period of AEC
// Complexity h for calculating the sum of periodic RTCs utilizations.
// Complexity log GRP is to calculate an initial period of AEC (least
// common multiple of periods for periodic RTCs).
// Complexity m1 to multiply the initial period of AEC by an integer,
// so that its utilization + periodic utilization is smaller or equal to 1
// Server utilization is equal to 1 – periodic utilization
h for all aperiodic RTCs in the individual{
Const calculated deadline = max (calculated absolute deadline of
aperiodic RTC with earlier calculated absolute deadline, arrival time
of current RTC) + execution time/server utilization
Const lateness of RTC = Specified absolute deadline –
calculated deadline
Const aperiodic lateness = aperiodic lateness + lateness of RTC
}
If a (Periodic utilization < = 1) and (aperiodic lateness < = 0) then{
Const Individual is accepted
h for all RTCs in the individual
Const Calculate total cost = total cost + cost of RTC
}
```

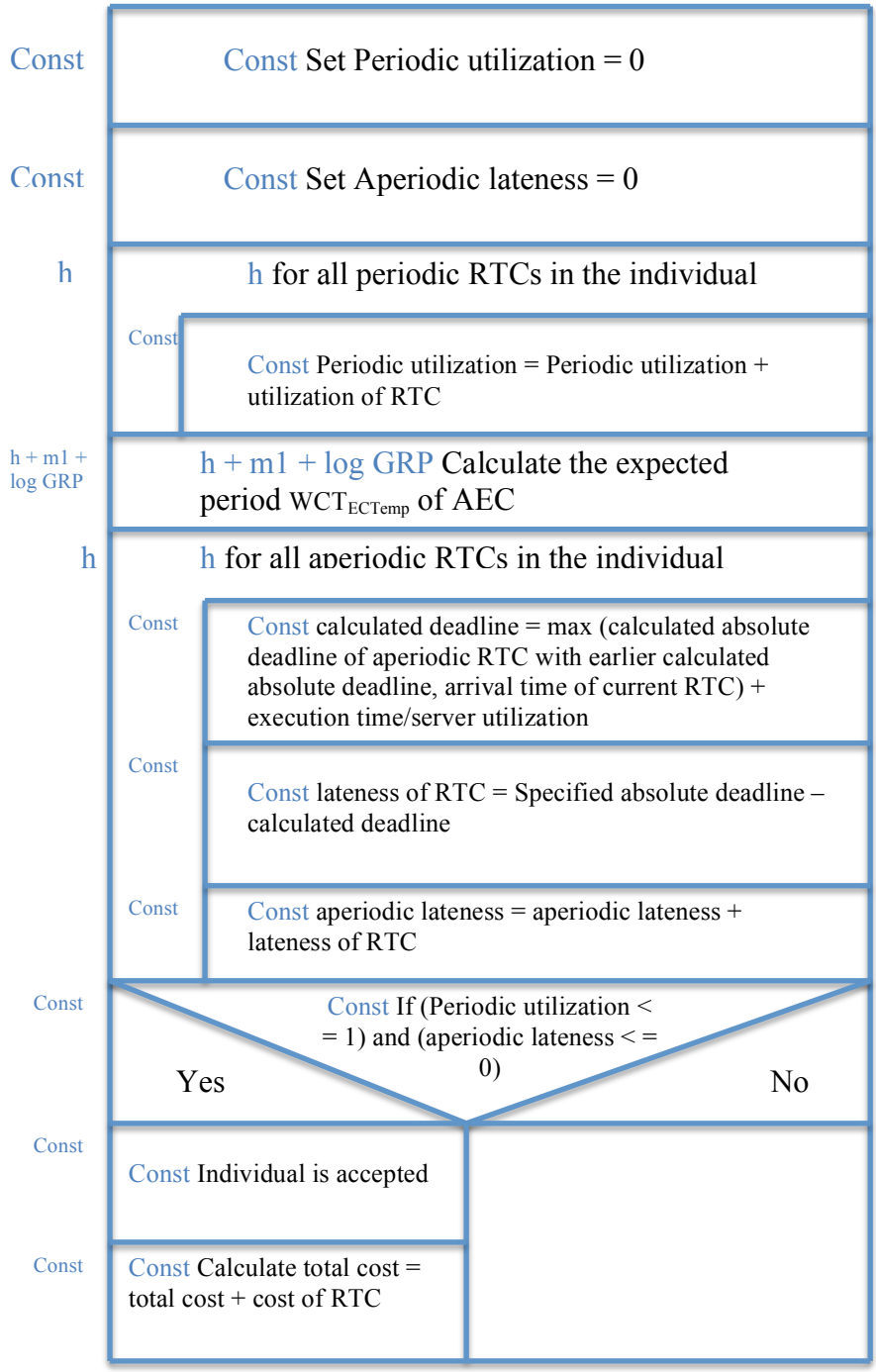

Const
Const Set Periodic utilization = 0
Const
Const Set Aperiodic lateness = 0
h
h for all periodic RTCs in the individual
Const
Const Periodic utilization = Periodic utilization + utilization of RTC
h + m1 + log GRP
h + m1 + log GRP Calculate the expected period WCT ECTemp of AEC
h
h for all aperiodic RTCs in the individual
Const
Const calculated deadline = max (calculated absolute deadline of aperiodic RTC with earlier calculated absolute deadline, arrival time of current RTC) + execution time/server utilization
Const
Const lateness of RTC = Specified absolute deadline – calculated deadline
Const
Const aperiodic lateness = aperiodic lateness + lateness of RTC
Const
Const If (Periodic utilization < = 1) and (aperiodic lateness < = 0)
Yes
No
Const
Const Individual is accepted
Const
Const Calculate total cost = total cost + cost of RTC

Diagram 4.5

Complexity of Diagram 4.5 is O(Const + Const + h ∗ Const + h + m1 + log GRP + h ∗ (Const + Const + Const) + Const + Const + Const) = O(h + m1 + log GRP)

// If first individual is not accepted, the adaptation is refused

// If first individual is accepted, we construct the second individual

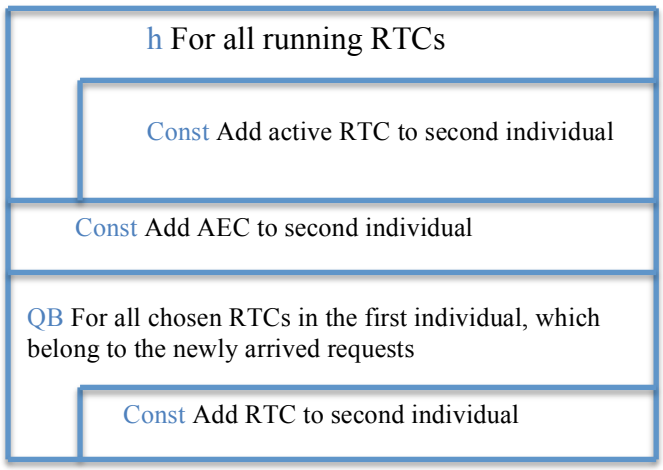


Diagram 4.6

Complexity of Diagram 4.6 is O(h ∗ Const + Const + QB ∗ Const) = O(h + QB)

// Construct the remaining individuals of the first generation

f For (number of individuals in a generation - 2){

Const Choose an RTC randomly from the previous individual
h ∗ f Combine the chosen RTC with RTCs that have lowest utilization in the other columns after excluding the ones chosen in the previous individual from these columns.
}

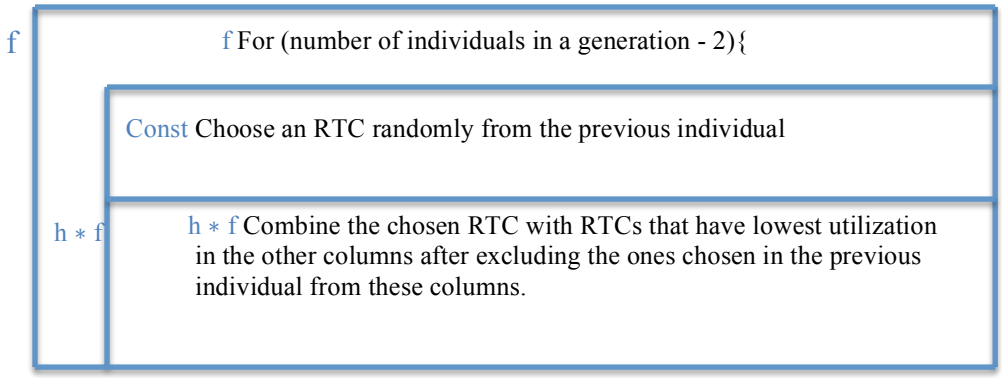


Diagram 4.7

Complexity of Diagram 4.7 is $O(f * (Const + f * h)) = O(f^2 * h)$

$O(f^2)$ Sort the individuals with respect to the global costs.
Const The best one is the preliminary solution.
Const PassIndivid = 0; // PassIndivid number of individuals that can
// solve the knapsack problem

f For all individuals in the initial generation{
h + m1 + log GRP Evaluate the individual
Const If solution is found (individual is accepted) then
Const PassIndivid = PassIndivid + 1
}
Const If PassIndivid is equal to 0 then
Const Return "No solution can be found"

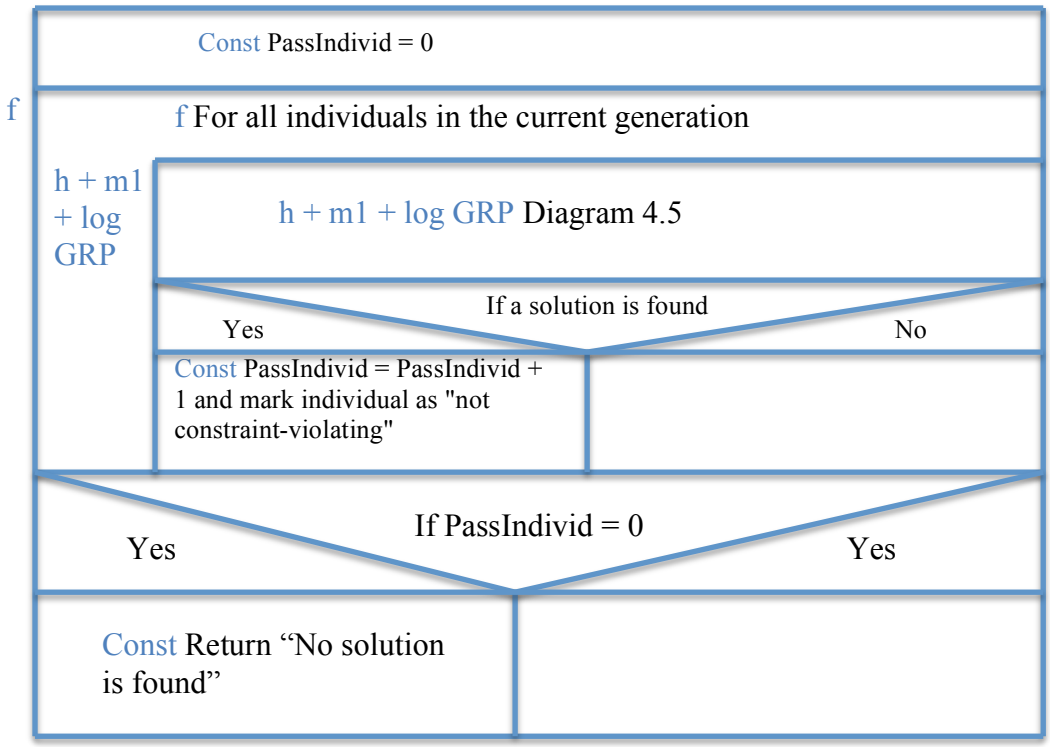


Diagram 4.8

Complexity of Diagram 4.8 is O(Const + f ∗(h + m1 + log GRP + Const + Const) + Const + Const) = O(f ∗ h + f ∗ m1 + f ∗ log GRP)

b) Selection and Mutation

Const If a solution is found in the initial generation then{

Const PreliminaryOptimum = lowest cost configuration contained in the initial generation

Const Iterate until timelimit reached or no improvement can be observed in the present generation compared to PreliminaryOptimum{

// Produce a new generation

f selection: drop all constraint-violating individuals and a predefined number of the worst individuals from the present generation
f mutation: from the selected individuals, generate new individuals
by replacing an arbitrary RTC in each individual by another arbitrary RTC from the same column (where the RTC belongs in the AdaptationRTCArray)
O(f ∗ h + f ∗ m1 + f ∗ log GRP)
Evaluate individuals in the new generation
O(f^2) Sort the individuals with respect to the global costs.

// x is the number of current generations
O(x) PreliminaryOptimum = lowest cost configuration obtained up to now
}
Const Result = PreliminaryOptimum

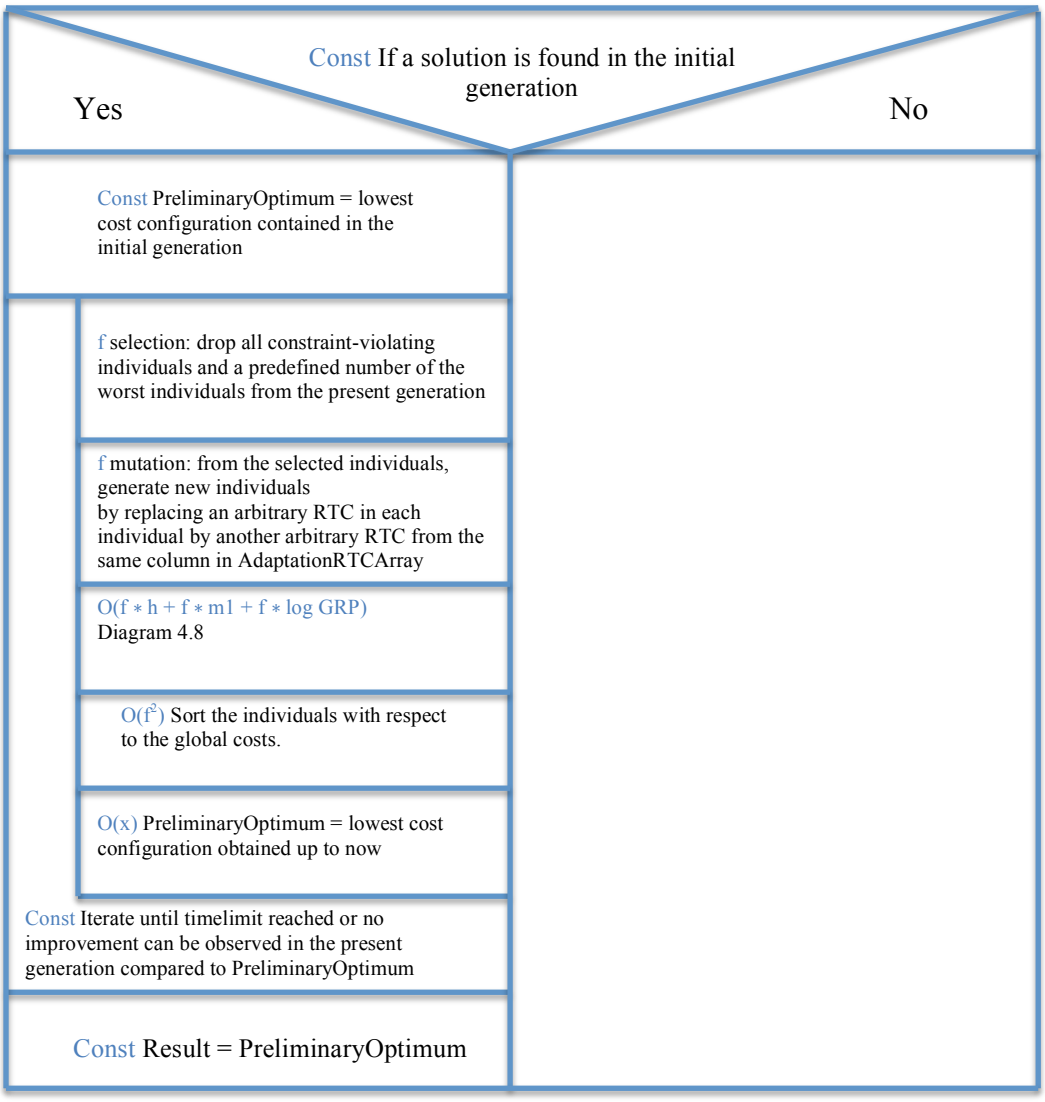


Diagram 4.9

Complexity of diagram 4.9 is O(1)[23]

Complexity of step 4: Contribution of Diagram 4.1 + Contribution of Diagram 4.2 + Contribution of Diagram 4.3 + Contribution of Diagram 4.4 + Contribution of Diagram 4.5 + Contribution of Diagram 4.6 + Contribution of Diagram 4.7 + Contribution of Diagram 4.8 + Contribution of Diagram 4.9 = $O(h) + O(h * QB * SC) + O(h^2 * f + QB * n + QB * h + QB * SC * f * h + h * n) + O(h * f) + O(h + m1 + \log GRP) + O(h + QB) + O(f^2 * h) + O(f * h + f * m1 + f * \log GRP) + O(1) = O(h + h * QB * SC + h^2 * f + QB * n + QB * h + QB * SC * f * h + h * n + h * f + h + m1 + \log GRP + h + QB + f^2 * h + f * h + f * m1 + f * \log GRP) = O(QB * SC * f * h + h^2 * f + QB * n + h * n + f^2 * h + f * m1 + f * \log GRP)$

5- Activate the accepted requests, and update the AEC:

If the newly triggered requests are accepted, the Active property of their RTCs becomes true. They are put into the ready queue, and the AEC schedules the first arrival of each request to be at NHP. This is done by loading the accepted RTCs into the memory (transforming them into ARTCs). The scheduler is responsible for loading the accepted RTCs at NHP.
The AEC updates then its properties according to the changes that will take place. Here, NumOfARTCs is modified according to the accepted requests. NumOfAARTCs is increased by number of accepted aperiodic RTCs, if the newly triggered RTCs are aperiodic, or the NumOfPARTCs is increased by number of accepted periodic RTCs, if the newly triggered RTCs are periodic. NumOfAARTCs is descreased by number of aperiodic RTCs that are to be deleted. NumOfPARTCs is decreased by number of periodic RTCs that are to be deleted.

WCT_{EC} get assigned the temporary value which is calculated in step 4 (See Diagram 4.5) as follows:

- $WCT_{EC} = WCT_{ECtemp}$

The hyperperiod is updated according to step 4. Hyperperiod = WCT_{EC}. The AEC schedules the first arrival of periodic requests at the NHP. The scheduler is responsible for loading the accepted RTCs (periodic and aperiodic) at NHP.

[23] This is the case as we assume a predefined fixed upper time limit.

In case the request is an update for one or several active RTCs, it replaces the RTClasses in the WorkingRTCArray, which includes the RTC/RTCs that should be updated by the RTClass/RTClasses of the newly arrived request. We set the Active property of the triggered elements in the newly arrived RTClasses to true.
After that, AdaptationRTCArray is set to empty.

Boundedness proof:

Activating the accepted requests is done in constant time by turning the Active property into true. As number of newly arrived request is bounded, this step is bounded.
Updating each of the AEC properties (NumOfAARTCs, NumOfPARTCs, period of the Engine-Cell) is done also in constant time.

Abstract Code:

Const If a solution is found, then {
(h+1) Set AdaptationRTCArray to the solution
// h + 1: h is the upper bound of classes in WorkingRTCArray. The
// additional one is for AEC. This represents the upper bound of RTCs
// that might construct a solution
(h+1) Activate all the elements in AdaptationRTCArray
(h+1) ∗ QB ∗ Const Store the solution in a ReadyQueue and store the type of each accepted request in RequestTypeQueue
// We iterate over all requests in the request buffer (QB) and the
// accepted requests in the solution (h+1) to store the type of each
// request to a queue.
h ∗ Const Set Active property for elements in ReadyQueue to true
// h is the upper bound of the RTCs in the solution
QB ∗ SC ∗ h ∗ f Add the arrived updating RTClass/RTClasses to WorkingRTCArray, and delete the RTClass/RTClasses that should be updated[24]
// For this replacement, and iteration is done over the accepted
// requests (QB). For each updating request we iterate over its
// RTCs (SC). We then make the replacement by iterating over the
// WorkingRTCArray (h ∗ f).

[24] As mentioned on page 47, we assume that an update point technically means that a context switch is located at the respective location in the code. In case of no update this is a context switch to "self" (which then can be ignored). In case of an update, this is a context switch to the new address. By this context switch the old code automatically becomes dead code which can be removed (this removal may be part of the context switch operation).

$h * f$ For (all elements in WorkingRTCArray)
h For (all elements in the ReadyQueue)
Const if an element is identical in both structures, then
Const Set the Active property of it to true
Const Put the elements in the ready queue according to the increasing order of arrival times
$(h + 1) * h$ Update NumOfAARTCs and NumOfPARTCs
// An iteration is done over the solution RTCs (h+1). For each
// solution RTC, an iteration is done over the active aperiodic RTCs
// to determine if it is new or part of the existing set. According to
// this NumOfAARTCs is updated. Same is done for the periodic
// RTCs.
Const Set the period of the Engine-Cell to its calculated period
WCT_{ECTEMP}}

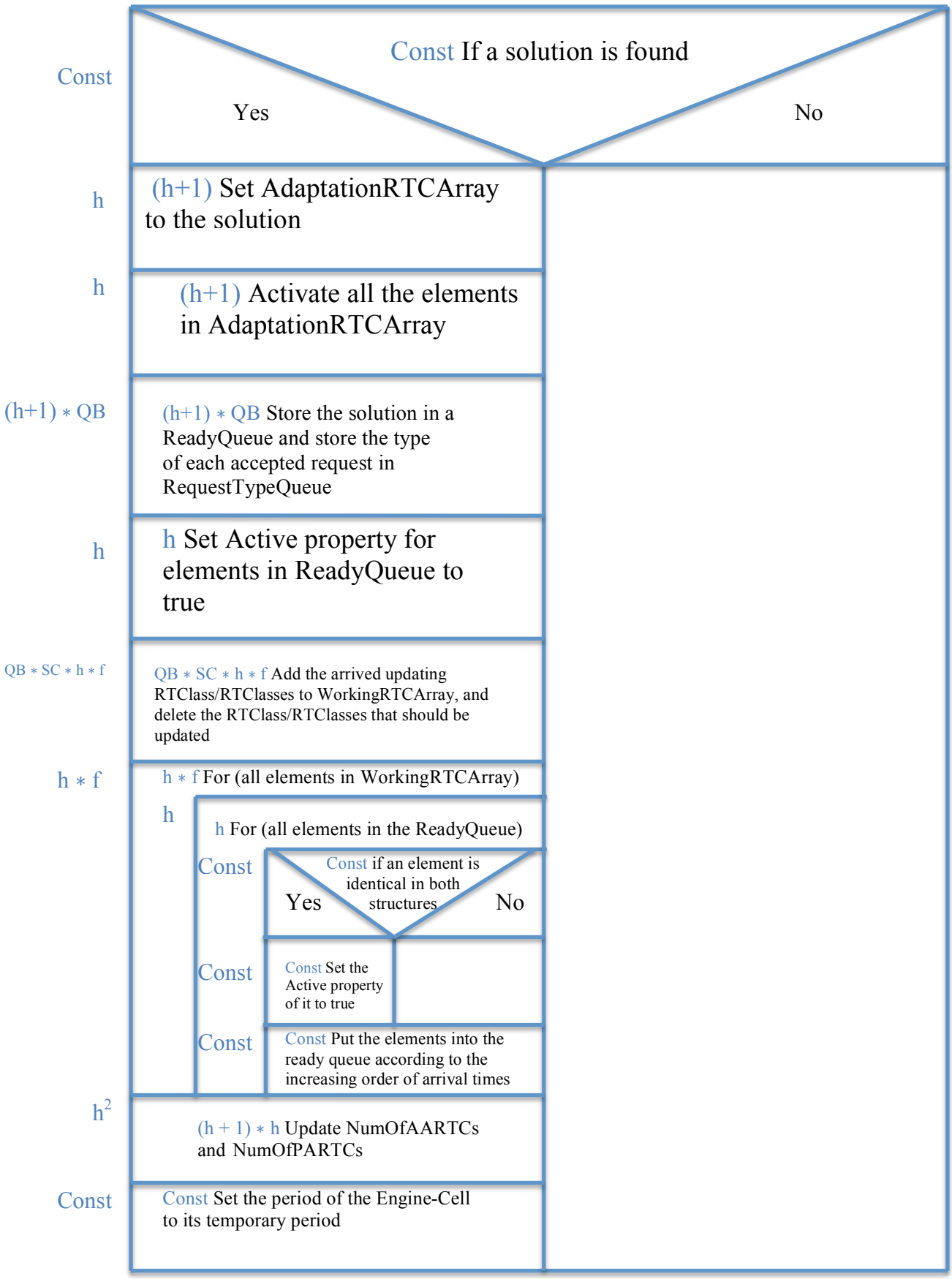

Const
Const If a solution is found
Yes
No
h
(h+1) Set AdaptationRTCArray to the solution
h
(h+1) Activate all the elements in AdaptationRTCArray
(h+1) ∗ QB
(h+1) ∗ QB Store the solution in a ReadyQueue and store the type of each accepted request in RequestTypeQueue
h
h Set Active property for elements in ReadyQueue to true
QB ∗ SC ∗ h ∗ f
QB ∗ SC ∗ h ∗ f Add the arrived updating RTClass/RTClasses to WorkingRTCArray, and delete the RTClass/RTClasses that should be updated
h ∗ f
h ∗ f For (all elements in WorkingRTCArray)
h
h For (all elements in the ReadyQueue)
Const
Const if an element is identical in both structures
Yes
No
Const
Const Set the Active property of it to true
Const
Const Put the elements into the ready queue according to the increasing order of arrival times
h2
(h + 1) ∗ h Update NumOfAARTCs and NumOfPARTCs
Const
Const Set the period of the Engine-Cell to its temporary period

Diagram 5.1

Complexity of diagram 5.1 is O(Const + h + h + h ∗ QB + h + QB ∗ SC ∗ h ∗ f + h ∗ f + h^2 + Const) = O(QB ∗ SC ∗ h ∗ f + h^2 ∗ f)

Complexity of step 5: O(QB ∗ SC ∗ h ∗ f + h^2 ∗ f)

6- Turning the triggered requests into non-triggered:

The Triggered Property of requests RTCs is turned into false. If the arrived requests are updates and they are not accepted, they are deleted. After that, WorkingRTCArray is copied to RTCArray, and then it is set to empty.

Boundedness proof:

The Triggered property of requests RTCs is turned to false in constant time.
Iterating over the requests in WorkingRTCArray is done in bounded time.
Copying WorkingRTCArray to RTCArray is done in bounded time, as dimensions of WorkingRTCArray are assumed to have a predefined upper bound which guarantees boundedness.
Resetting WorkingRTCArray is done in constant time.

Abstract Code:

h ∗ f ∗ QB ∗ SC Turn the triggered property of triggered RTCs in WorkingRTCArray to false
h ∗ f Copy WorkingRTCArray to RTCArray
Const Reset WorkingRTCArray to empty

Complexity of step 6: O(h ∗ f ∗ QB ∗ SC)

7- Notify the system, in case the requests are not accepted:

If the set of proceeded requests cannot be accepted, then a notification is sent by the AEC to the system for substituting the proceeded set by another set of requests. $Cost_{total}$, WCT_{ECtemp}, The expected hyperperiod, AdaptationRTCArray, ExpPARTCs and ExpAARTCs are reset to their initial values.

Boundedness proof:

If the request is not accepted, the Triggered property is set to false in constant time.

If the request is not accepted then a notification is sent to the system administrator in constant time.
$Cost_{total}$, WCT_{ECtemp}, The expected hyperperiod, AdaptationRTCArray, WorkingRTCArray, ExpPARTCs and ExpAARTCs are reset in constant time.

Abstract Code:

Const Notify the system "The newly arrived requests cannot be accepted"
Const Reset $Cost_{total}$, WCT_{ECtemp}, AdaptationRTCArray, WorkingRTCArray, The expected hyperperiod, ExpPARTCs and ExpAARTCs.

Complexity of step 7: O(1)

Complexity of the algorithm: Complexity of step 1 + Complexity of step 2 + Complexity of step 3 + Complexity of step 4 + Complexity of step 5 + Complexity of step 6 + Complexity of step 7 = $O(b * h * f) + O(QB * h * f * SC + SC^2) + O(PN) + O(QB * SC * f * h + h^2 * f + QB * n + h * n + f^2 * h + f * m1 + f * \log GRP) + O(QB * SC * h * f + h^2 * f) + O(h * f * QB * SC) + O(1) = O(b * h * f + PN + f * m1 + f * \log GRP + f^2 * h + h^2 * f + QB * n + h * n + h * f * QB * SC + SC^2)$

Chapter 7 Summary and Future Work

In this thesis, we have developed an approach that provides real-time operating systems by organic adaptability feature. The idea is to let any of the mentioned systems react to environmental changes as organic objects do. This implies building an infrastructure of the system, which can change its behaviour at runtime. The basic unit in this infrastructure is a cell. A cell is a task that can change ist structure and behaviour by selecting a variant of it at run time. Cell variants can be added online.
Variants of a specific cell differs in their time and quality characteristics. The way they are chosen at run time follows resource and time limitations, in order to enhance the quality of the system. The boundedness of our algorithm has been proven.

The main contributions of the thesis can be summarized by:

- Developing the concept of a task from a static component that have a static structure and behavior to a dynamic component that can change itself at run time.

- Handling different kinds of requests. These kinds include all expected changes that a real-time system may require at run time.

- Handling many requests at once. Each request can have a type different from the others. This enables the system to react to many events at once, and as a result fasten system reaction and make it more accurate.

- Solving the problem of adapting requests within the real-time constraints, and other optimization constraints. The solution algorithm enables to always consider new constraints according to the data, components, and parameters of the system.

- Having a real-time system that can behave as organic objects do. A self-adaptable real-time system is obtained that can intelligently behave with no or limited human interaction.

Many new trends can be developed in the context of the described problem in the future, e.g.,

- How can we distribute the central algorithm that is run by the Engine-cell on several nodes? Such a distribution may help to save more processor utilization on one node. This in turn will increase the self-adaptability of this node.

- If we have a distributed algorithm, how can we obtain fault tolerance? Fault tolerance is necessary to recover the system, in case any of its components falls down.

- How to deal with the boundedness of the algorithm in case of a non-deterministic network, e.g., if the nodes that are holding the cells can move or change their places? This question applies to multi-agent real-time systems, which may exchange information in order to enhance the total behaviour of the system, and makes it highly adaptable.

- In our proposal, we have only one remote node, where newly developed cells are loaded. In case we have a network, the newly developed RTCs can be divided according to their type. Only specific types can be loaded on dedicated parts of the network. This can save the costs of developing the system. As nodes can be placed, where appropriate developers exist.

- In a multi-agent system that is connected with a network, we might have several controlling cells (other than the Engine-cell). These cells might be distributed among the agents and communicate with each other in order to achieve a specific goal.
This allows for several adaptaion techniques. Each one is hold by a controlling cell. It also allows for applying algorithms other than adaptation, or algorithms that may cooperate with the adaptation, in ordert o achieve further goals of the system. In this case, each algorithm can be hold or partially hold by a controlling cell.

- Currently we apply a genetic algorithm to solve the knapsack problem. Later, we can improve the algorithm by applying different genetic algorithms or solve the knapsack problem in a different way. Each way may result a different optimization output.

- We may measure the optimization output in the future, by running the algorithm on a real-time operating system and observing the results.

- Currently we are not discussing task communication, or interrelated tasks. This is an additional impose that can be considered later.

Appendix 1: List of Abbreviations

Appendix 2: List of Figures

References

[1] Edward A. Lee., “Cyber Physical Systems: Design Challenges”, 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC), 2008.

[2] Oliver Imbusch, Falk Langhammer, Guido von Walter, “Ercatons and Organic Programming: Say Good-Bye to Planned Economy”, Dagstuhl Seminar Proceedings 2006.

[3] John E. Speich and Jacob Rosen, “Medical Robotics, Encyclopedia of Biomaterials and Biomedical Engineering”, 2004.

[4] Giorgio C. Buttazzo, “Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and Applications”, Third Edition, Springer, 2011.

[5] H. Rosen, “Handbook of Graph Theory”, Series Editor Kenneth, CRC Press, edited by Jonathan L. Gross, Jay Yellen, 2004.

[6] J.Matravers, “Introduction to Computer Systems Architecture and Programming”, University of London International Programmes, 2011.

[7] H. Ghetto, M. Silly, and T. Bouchentouf. “Dynamic scheduling of real-time tasks under precedence constraints”, Journal of Real-Time Systems, 2, 1990.

[8] Marco Spuri, Giorgio C. Buttazzo, “Efficient Aperiodic Service under Earliest Deadline Scheduling”, Real-Time Systems Symposium, 1994.

[9] Lial Khaluf, Franz-Josef Rammig, “Organic Programming of Real-Time Operating Systems”, In the ninth international conference on Autonomic and Autonomous Systems (ICAS), 2013.

[10] Thomas Lengauer, “Combinatorial Algorithms for Integrated Circuit Layout”, Applicable Theory in Computer Science, A Wiley-Teubner Series, ISBN 0 471 92838 0 (Wiley). ISBN 3 519 02110 2 (Teubner), 1990.

[11] Thomas. Weise, “Global Optimization Algorithms - Theory and Application”, Self-Published, second edition, 2009. Online available at http://www.it-weise.de/.

[12] Andreas Geyer-Schulz, “Holland classifier systems”, In APL’95: Proceedings of the international conference on Applied programming languages, pages 43–55, San Antonio, Texas, United States. ACM Press, New York, NY,

USA. ISBN: 0-8979-1722-7, 1995.

[13] Nikolaus Hansen, Andreas Ostermeier, and Andreas Gawelczyk, “On the adaptation of arbitrary normal mutation distributions in evolution strategies: The generating set adaptation”. In Proceedings of the 6th International Conference on Genetic Algorithms, pages 57–64, 1995. In proceedings [33].

[14] Silja Meyer-Nieberg and Hans-Georg Beyer, “Self-adaptation in evolutionary algorithms”. In Parameter Setting in Evolutionary Algorithms. Springer, 2007. In collection [34].

[15] Oliver Kramer, “Self-Adaptive Heuristics for Evolutionary Computation”, volume 147 of Studies in Computational Intelligence. Springer Berlin / Heidelberg, July 2008. ISBN: 978-3-54069-280-5. Series editor: Janusz Kacprzyk.

[16] David B. Fogel, “System Identification through Simulated Evolution: A Machine Learning Approach to Modeling”, Ginn Press, Needham Heights, MA, June, 1991. ISBN: 978-0-53657-943-0.

[17] David B. Fogel, “Evolving a checkers player without relying on human experience”, Intelligence, 11(2):20–27, 2000. ISSN: 1523-8822.

[18] David B. Fogel, “Blondie24: playing at the edge of AI. The Morgan Kaufmann Series in Artificial Intelligence”, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, September 2001. ISBN: 978-1-55860-783-5.

[19] Lawrence Jerome Fogel, and David B. Fogel, “A preliminary investigation on extending evolutionary programming to include self-adaptation on finite state”. Informatica (Slovenia), 18(4), 1994.

[20] Vincent W. Porto, David B. Fogel, and Lawrence Jerome Fogel. “Alternative neural network training methods. IEEE Expert: Intelligent Systems and Their Applications”, 10(3):16–22, 1995. ISSN: 0885-9000.

[21] John R. Koza, “Non-Linear Genetic Algorithms for Solving Problems”, United States Patent and Trademark Office, 1988. United States Patent 4,935,877. Filed May 20, 1988. Issued June 19, 1990. Australian patent 611,350 issued september 21, 1991. Canadian patent 1,311,561 issued december 15, 1992.

[22] John R. Koza, “Hierarchical genetic algorithms operating on populations of computer programs”. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89, pages 768–774, 1989. In proceedings [60].

[23] Fred Glover, Gary A. Kochenberger, “Handbook of Metaheuristics”,

© 2003 Kluwer Academic Publishers, eBook ISBN: 0-306-48056-5.

[24] Jakob Puchinger, Günther R. Raidle, “Combining Metaheuristics and Exact Algorithms in Combinatorial Optimization: A Survey and Classification”, Institute of Computer Graphics and Algorithms. Vienna University of Technology. IWINAC 2005, LNCS 3562, pp. 41–53, 2005.

[25] S. Kirkpatrick, C. Gellat, and M. Vecchi, “Optimization by simulated annealing”. Science, 220:671–680, 1983.

[26] H. R. Louren, OC. Martin, and T. Stützle, “Iterated local search”, In Glover and Kochenberger [27], pages 321–353.

[27] T. Bäck, D. Fogel, and Z. Michalewicz, “Handbook of Evolutionary Computation”, Oxford University Press, New York NY, 1997.

[28] Peter Rossmanith, “Simulated Annealing”, Algorithmus der Woche – Informatikjahr, 2006.

[29] H.R. Lourenço, O Martin, and T. Stützle, “Iterated Local Search”, In *Handbook of Metaheuristics*, F. Glover and G. Kochenberger, (eds.), Kluwer Academic Publishers, International Series in Operations Research & Management Science, pp. 321-353, 2003.

[30] David Pisinger, “Algorithms for knapsack problems”, Dept of Computer Science, University of Kopenhagen, PhD thesis, February, 1995.

[31] Mhand Hifi, Mustapha Michrafy, Abdelkader Sbihi, “Heuristic algorithms for the multiple-choice multidimensional knapsack problem”, Journal of the Operational Research Society, Palgrave Macmillan, vol. 55, pp. 1323-1332, 2004.

[32] Alejandra Duenas, Christine Martinelly, G. Tütüncü, “A Multidimensional Multiple-Choice Knapsack Model for Resource Allocation in a Construction Equipment Manufacturer Setting Using an Evolutionary Algorithm”, APMS 2014, Part I, IFIP AICT 438, pp. 539–546, 2014.

[33] Larry J. Eshelman, editor. Proceedings of the Sixth International Conference on Genetic Algorithms, July 15-19, 1995, Pittsburgh, PA, USA. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN: 1-5586-0370-0.

[34] Fernando G. Lobo, Cláudio F. Lima, and Zbigniew Michalewicz, editors. Parameter Setting in Evolutionary Algorithms, volume 54 of Studies in Computational Intelligence, Springer-Verlag Gmbh, Berlin, Germany, March 1, 2007. ISBN: 3-5406-9431-5, 978-3-54069-431-1.

[35] F. Glover and G. Kochenberger, editors. Handbook of Metaheuristics, volume 57 of International Series in Operations Research & Management Science. Kluwer Academic Publishers, 2003.

[36] John H Holland, Lashon Booker, Marco Colombetti, Marco Dorigo, David E. Goldberg, Stephanie Forrest, Rick L. Riolo, Robert E. Smith, Pier Luca Lanzi, Wolfgang Stolzmann, Stewart W. Wilson. "What is a Learning Classifier System?", LCS´99, LNAI 1813 pp. 3-32, 2000.

[37] https://ipfs.io/ipfs/QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco/wiki/Evolution_strategy.html.

[38] Hans-Paul Schwefel, "Numerical Optimization of Computer Models", John Wiley and Sons Ltd, New York, NY, USA, June 17, 1981. ISBN: 0-4710-9988-0, 978-0-47109-988-8.

[39] Hans-Georg Beyer, "Toward a theory of evolution strategies: The (μ, λ)-theory", Evolutionary Computation, 2(4): 381–407, 1994.

[40] Yoshiyuki Matsumura, Kazuhiro Ohkura, and Kanji Ueda, "Advantages of global discrete recombination in (μ/μ,λ)-evolution strategies". In Proceedings of the 2002 Congress on Evolutionary Computation CEC2002, volume 2, pages 1848–1853, 2002. doi:10.1109/CEC.2002.1004524. In proceedings [45].

[41] Hannes Geyer, Peter Ulbig, and Siegfried Schulz, "Encapsulated evolution strategies for the determination of group contribution model parameters in order to predict thermodynamic properties", In PPSN V: Proceedings of the 5th International Conference on Parallel Problem Solving from Nature, pages 978–987, 1998. In proceedings [46].

[42] Hannes Geyer, Peter Ulbig, and Siegfried Schulz, "Use of evolutionary algorithms for the calculation of group contribution parameters in order to predict thermodynamic properties – part 2: Encapsulated evolution strategies", Computers and Chemical Engineering, 23(7):955–973, July 1, 1999. doi:10.1016/S0098-1354(99)00270-7.

[43] Hannes Geyer, Peter Ulbig, and Siegfried Schulz, "Verschachtelte evolutionsstrategien zur optimierung nichtlinearer verfahrenstechnischer Regressionsprobleme", Chemie Ingenieur Technik, 72(4):369–373, 2000. ISSN: 1522-2640. doi:10.1002/1522-2640(200004)72:4<369::AID-CITE369>3.0.CO;2-W.

[44] Ingo Rechenberg, "Evolutionsstrategie '94", volume 1 of Werkstatt Bionik und Evolutionstechnik, Frommann-Holzboog Verlag, Stuttgart, Germany,

September 1994. ISBN: 978-3-77281-642-0.

[45] David B. Fogel, Mohamed A. El-Sharkawi, Xin Yao, Garry Greenwood, Hitoshi Iba, Paul Marrow, and Mark Shackleton, editors. Proceedings of the IEEE Congress on Evolutionary Computation, CEC2002, May 12–17, 2002, Honolulu, HI, USA. IEEE Press, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA. ISBN: 0-7803-7278-6. CEC 2002 – A joint meeting of the IEEE, the Evolutionary Pro- gramming Society, and the IEE. Held in connection with the World Congress on Computational Intelligence (WCCI 2002).

[46] Àgoston E. Eiben, Thomas Bäck, Marc Schoenauer, and Hans-Paul Schwefel, editors. Proceedings of the 5th International Conference on Parallel Problem Solving from Nature – PPSN V, volume 1498/1998 of Lecture Notes in Computer Science (LNCS), September 27–30, 1998, Amsterdam, The Netherlands. Springer. ISBN: 3-5406-5078-4, 978-3-54065-078-2. doi:10.1007/BFb0056843.

[47] John R. Koza, "The genetic programming paradigm: Genetically breeding populations of computer programs to solve problems", Technical Report STAN-CS-90- 1314, Computer Science Department, Stanford University, Margaret Jacks Hall, Stan- ford, CA 94305, June 1990.

[48] John R. Koza, "A hierarchical approach to learning the boolean multiplexer function", In Proceedings of the First Workshop on Foundations of Genetic Algorithms, pages 171–191, 1990. In proceedings [61].

[49] John R. Koza, "Genetic evolution and coevolution of computer programs. In Artificial Life II", pages 603–629, 1990. Revised November 29, 1990. In proceedings [62].

[50] John R. Koza, "Evolution and coevolution of computer programs to control independent-acting agents", In From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior, pages 366–375, 1990. In proceedings [63].

[51] John R. Koza, "Genetic Programming: On the Programming of Computers by Means of Natural Selection", A Bradford Book. The MIT Press, Cambridge, Massachusetts, USA, 1992. ISBN: 0-2621-1170-5, 978-0-26211-170-6.

[52] John R. Koza, "Genetic Programming: On the Programming of Computers by Means of Natural Selection", A Bradford Book. The MIT Press, Cambridge, Massachusetts, USA, 1992. ISBN: 0-2621-1170-5, 978-0-26211-170-6.

[53] Walter Alden Tackett, "Recombination, Selection, and the Genetic Construction of Computer Programs", PhD thesis, Faculty of the Graduate

School of the University of Southern California, Los Angeles, CA 90089-2562, USA, April 17, 1994.

[54] John R. Koza, “Concept formation and decision tree induction using the genetic programming paradigm”, In Parallel Problem Solving from Nature - Proceedings of 1st Workshop, PPSN 1, pages 124–128, 1990. In proceedings [55].

[55] Hans-Paul Schwefel and Reinhard Männer, editors. Proceedings of the 1st Workshop on Parallel Problem Solving from Nature, PPSN I, volume 496/1991 of Lecture Notes in Computer Science (LNCS), October 1–3, 1990, FRG, Dortmund, Germany. Springer. ISBN: 3-5405-4148-9, 978-3-54054-148-6. doi:10.1007/BFb0029723. Published 1991.

[56] Prajna Kunche, K.V.V.S. Reddy, “Metaheuristic Applications to Speech Enhancement”, Springer, 2016.

[57] Leonora Bianchi, Marco Dorigo, Luca Maria Gambardella, Walter J. Gutjahr, “A survey on metaheuristics for stochastic combinatorial optimization”. Natural Computing: an international journal. 8 (2): 239-287, 2009.

[58] http://cgi.di.uoa.gr/~ys02/siteAI2008/local-search-2spp.pdf.

[59] Felix Streichert, “Introduction to Evolutionary Algorithms”, University of Tuebingen, 2002.

[60] N. S. Sridharan, editor. Proceedings of the 11th International Joint Conference on Artificial Intelligence, volume 1, August 1989, Detroit, MI, USA. Morgan Kaufmann, San Francisco, CA, USA. ISBN: 1-5586-0094-9.

[61] Bruce M. Spatz and Gregory J. E. Rawlins, editors. Proceedings of the First Workshop on Foundations of Genetic Algorithms (FOGA), July 15–18, 1990, Indiana University, Bloomington Campus, Indiana, USA. Morgan Kaufmann Publishers, Inc., 2929 Campus Drive, Suite 260, San Mateo, CA 94403, USA. ISBN: 1-5586-0170-8. Published July 1, 1991.

[62] Christopher G. Langton, C.E. Taylor, D.J. Farmer, and S. Rasmussen, editors. Artificial Life II: Proceedings of the Workshop on Artificial Life, volume X of Santa Fe Institute Studies in the Science of Complexity, February 1990, Santa Fe, New Mexico, USA. Addison-Wesley. ISBN: 0-2015-2571-2, 978-0-20152-571-7. Published 1992, republished by Westview Press (March 28, 2003).

[63] Jean-Arcady Meyer and Stewart W. Wilson, editors. From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior, September 24–28, 1990, Paris, France. The MIT Press,

Cambridge, MA, USA. ISBN: 0-2626-3138-5. Published in 1991.

[64] Simon Oberthür, Leszek Zaramba, and Hermann-Simon Lichte, “Flexible Resource Management for Self-X Systems: An Evaluation”, in Proceedings of ISORCW´10, pp. 1-10, 2010.

[65] Simon Oberthür, “Towards an RTOS for Self-Optimizing Mechatronic Systems, Dissertation”, Paderborn, Germany, October 30, 2009.

[66] Cheng at al. (EDs.): “Self-Adaptive Systems”, LNCS 5525, pp. 1-26, Springer Verlag, Berlin Heidelberg, 2009.

[67] R. de Lemos et al. (Eds.): “Self-Adaptive Systems”, LNCS 7475, pp. 1-32, Springer Verlag, Berlin Heidelberg, 2013.

[68] J.C. Laprie, “From dependability to resilience”, In: International Conference on Dependable Systems and Networks (DSN 2008), Anchorage, AK, USA, pp. G8-G9, 2008.

[69] E.B. Swanson, “The dimensions of maintenance”, In Proceedings of the 2nd International Conference on Software Engineering (ICSE 1976), pp. 492-497. IEEE Computer Society Press, 1976.

[70] A. Carzaniga, A. Gorla, N. Perino, M. Pezzè, “Automatic workarounds for web applications”, In: FSE 2010: Proceedings of the 2010 Foundations of Software Engineering Conference, pp. 237-246. ACM, New York, 2010.

[71] A. Carzaniga, A. Gorla, M. Pezzè, “Self-healing by means of automatic workarounds”, In SEAMS 2008: Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems, pp. 17-24. ACM, New York, 2008.

[72] Object Management Group (OMG): “Software & Systems process Engineering Meta Model Specification (SPEM)”, Version 2.0, 2008.

[73] D. Garlan, S.W. Cheng, A.C. Huang, B. Schmerl, P. Steenkiste, “Rainbow: Architecture-based self-adaptation with reusable infrastructure”, IEEE Computer 37, 46-54, 2004.

[74] IBM: “An architectural blueprint for autonomic computing”, Tech. rep. IBM, January 2006.

[75] P. Oreizy, M.M Gorlick, R.N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, A.L. Wolf, “An architecture- based

approach to self-adaptive software", IEEE Intelligent Systems 14, 54-62, 1999.

[76] Y. Brun, N. Medvidovic, "An architectural style for solving computationally intensive problems on large networks", In Proceedings of Software Engineering for Adapting and Self-Managing Systems, SEAMS 2007, Minneapolis, MN, USA, May 2007.

[77] I. Georgiadis, J. Magee, J. Kramer, "Self-Organizing Software Architectures for Distributed Systems", In: 1st Workshop on Self-Healing Systems. ACM, New York, 2002.

[78] S. Malek, M. Mikic-Rakic, N. Medvidovic, "A Decentralized Redeployment Algorithm for improving the Availability of Distributed Systems", In A. Dearle, R. Savani (eds.) CD 2005. LNCS, vol. 3798, pp 99-114. Springer, Heidelberg, 2005.

[79] P. Vromant, D. Weyns, S. Malek, J. Andersson, "On interacting Control loops in self-adaptive systems", SEAMS 2011, Honolulu, Hawaii, 2011.

[80] D. Weyns, S. Malek, J. Andersson, "On decentralized self-adaptation: lessons from the trenches and challenges for the future", In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2010, pp. 84-93. ACM, New York, 2010.

[81] Alejandra Duenas, Christine Di Martinelly, and G. Yazgi Tütüncü, "A Multidimensional Multiple-Choice Knapsack Model for Resource Allocation in a Construction Equipment Manufacturer Setting Using an Evolutionary Algorithm". B. Grabot et al. (Eds.): APMS 2014, Part I, IFIP AICT 438, pp. 539-546, 2014.

[82] https://www.cisco.com/c/en/us/support/docs/lan-switching/spanning-tree-protocol/5234-5.pdf.

[82] Ryan J. Urbanowicz, Jason H. Moore, "Learning Classifier Systems: A Complete Introduction, Review, and Roadmap". Journal of Artificial Evolution and Applications. 2009: 1-25.

[84] Linus Pauling, "The Nature of the Chemical Bond", Cornell Univ. Press, Ithaca, New York, 1960. ISBN: 0-8014-0333-2.

[85] Herbert Spencer, "The Principles of Biology", volume 1. London & Edinburgh: Williams and Norgate, first edition, 1864 and 1867.

[86] Charles Darwin, “On the Origin of Species”, John Murray, sixth edition, November 1859.

[87] Arnold Neumaier, “Global optimization and constraint satisfaction”, In I. Bomze, I. Emiris, Arnold Neumaier, and L. Wolsey, editors, Proceedings of GICOLAG workshop (of the research project Global Optimization, Integrating Convexity, Optimization, Logic Programming and Computational Algebraic Geometry), December 2006.

[88] Nassi and Schneiderman, “Flowchart Techniques for Structured Programming”, Technical Contributions, Sigplan Notices, 1973.

[89] http://math.feld.cvut.cz/mt/txtb/3/txe3ba3c.htm.

[90] https://study.com/academy/lesson/monotonic-function-definition-examples.html.

[91] http://www.nlreg.com/asymptot.htm.

[92] John K. Ousterhout, “Scheduling Techniques for Concurrent Systems”, IEEE, 1982.

[93] Theodore Johnson, Dennis Shasha, “A low Overhead High Performance Buffer Management Replacement Algorithm”, Proceedings of the 20th VLDB Conference Santiago, Chile, 1994.

[94] Elizabeth J. O'Neil, Patrick E. O'Neil, Gerhard Weikum, “The LRU-K Page Replacement Algorithm for Database Disk Buffering”, Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data. SIGMOD '93. New York, NY, USA: ACM: 297-306.

[95] http://hm.hgesser.de/ra-ws2010/folien/Musterloesung-prakt08.pdf.

[96] Anupam Bhattacharjee, Biolob Kumar Debnath, “A New Web Cache Replacement Algorithm1”, proceedings of PacRim 2005, IEEE conference on Communications, Computers and Signal Processing, Victotia, B. C. Canada, August 24-26, 2005.

[97] Claudio Comis Da Ronco Ernesto Benini, “A Simplex-Crossover-Based Multi-Objective Evolutionary Algorithm”, IAENG Transactions on Engineering Technologies, Volume 247 of the series Lecture Notes in Electrical Engineering pp 583-598, 2013.

[98] P.H.Mills, “Extensions to guided local search”, A thesis submitted to the degree of Ph.D., University of Essex, 2001.