An Algorithmic Solution for Adaptable Real-Time Applications

Lial Khaluf
Email: lial.khaluf@googlemail.com

and

Franz-Josef Rammig
Email: franz@upb.de
University of Paderborn
Paderborn, Germany

Abstract — **Most real-life facilities nowadays belong to real-time systems. E.g., airplanes, trains, cars, medical machines, alarming systems and robotics, etc. Some of these systems behave on their own, separately or in cooperation. Some of them interact with humans. Operating and interaction is done under conditions defined inside the systems and the environment. However, these systems and environments might grow or change over time. In order to develop safe and high-quality real-time applications, we need to make them highly self-adaptable systems. In this paper, we describe the detailed steps of a real-time aware adaptation algorithm and we go through the boundedness proof of each step. The algorithm mimics the behaviour of organic cells. It introduces a new kind of real-time tasks. This kind enables tasks to change their behaviour at run time according to internal changes inside the system and external changes in the environment, preserving all real-time constraints. This includes real-time constraints for the adaptation process itself. Following this concept, real-time tasks are turned to be real-time cells.**

keywords – adaptation array; genetic optimization; organic behavior; boundedness.

I. INTRODUCTION

Current trends of adaptation mechanisms have been applied in traditional software systems as well as real-time (RT) systems. RT systems, however, lack the required flexibility for adapting themselves to changes being unpredictable at design time. In this paper, we try to overcome this limitation by providing an adaptation solution at run time. The problem we solve assumes a system that has to fulfil a set of hard-deadline periodic and aperiodic tasks. Aperiodic tasks might have dependencies between each other. Modifications to the current set of tasks may happen at run time as, e.g., a result of environmental changes. Modifications may include adding a new task or a set of dependent tasks, updating a task or a set of dependent tasks, and deleting a task or a set of dependent tasks. We assume that by updating dependent tasks, no new tasks are added to the dependability graph, and no existing tasks are deleted. The goal of the approach presented in this paper is to adapt the newly arrived modifications at run time without breaking any of the stated RT constraints.

In [1], we have introduced a summary of the adaptation algorithm. In this paper, we go through details of it, and prove the boundedness of each step. In our solution, we assume a bounded, but online expandable, ecosystem of RT tasks. Each RT task may exist by means of a bounded, but online expandable, set of variants. All variants of a task share the same principal functionality, however, at different quality levels. Whenever activated, the algorithm tries to find a solution that can accept all currently requested modifications. For this reason, it tries to find a selection of task variants such that all RT constraints are satisfied and at the same time the overall quality over all tasks is maximized. The underlying ecosystem is represented by a two-dimensional data structure called $RTCArray$ [2]. It is divided into classes. Each class is represented by a column. A class represents a unique functionality. Each variant within a column provides the same principal functionality as the other variants, but with different time and quality characteristics. We model quality by a cost function where cost is defined as the inverse of quality, i.e., highest possible quality is modelled by cost 0. We call a variant an RT cell. The reason is that the structure and behaviour of variants can change at run time following the environment of the system. Modification requests may arrive at any time. According to their priority, they are put into a queue with a predefined capacity. The queue is handled by a central cell called the Engine-Cell (EC). EC tries to find a best combination of variants such that the arrived modifications can be accepted. The problem of selecting a best combination of variants is mapped on solving a Knapsack problem, executed on a so-called $AdaptationRTCArray$. This is a subset of $RTCArray$ restricted on currently running cells that have to continue execution and augmented by newly arriving modifications. As this Knapsack problem has to be solved under RT constraints, an "anytime algorithm" is needed to solve it. We decided for a genetic algorithm with a trivial initial population. The individuals of populations are evaluated concerning their respected overall cost under the constraint that all RT constraints have to be satisfied. It is assumed that the RT system is running under Earliest Deadline First (EDF) algorithm [16] as principal scheduling algorithm. EDF is used to schedule a set of independent tasks by always giving the priority to the task with earliest absolute deadline [16]. As the challenge we are aiming to overcome in our approach considers also the case of dependent tasks, Earliest Deadline First with Precedence Constraints algorithm (EDF*) [5] is used. EDF* transforms a set of dependent tasks into a set of independent tasks by recalculating the timing parameters of the

tasks. The resulting independent tasks can then be scheduled by EDF. The dependent set is schedulable if and only if the independent set can be schedulable [16]. For considering aperiodic tasks, we use the Total Bandwidth Server (TBS) [6]. TBS calculates new absolute deadlines of aperiodic tasks, so that the server can behave as a periodic task, which is schedulable with the other periodic tasks in the system under EDF. In our approach, the RT constraints of all periodic tasks can be calculated using a utilization bound 1 and those for aperiodic tasks by checking that deadlines calculated by TBS are less or equal than specified hard deadlines.

In Section II, we present the related work. Section III presents the definitions of some basic concepts. Section IV includes a scenario example of an RT application, in which our approach can be applied. Section V describes the algorithm and boundedness proof of each step. In the last section, we present a conclusion and future work.

II. Related Work

In this paper, we are solving an optimization problem. The goal is to provide enough processor capacity for the current requests and the currently running cells while minimizing the costs. The problem can be modelled by a multidimensional multiple-choice knapsack problem (MMKP). Many approaches were defined for solving this problem. In [7] a metaheuristic approach is developed. It simplifies the MMKP into multiple-choice knapsack problem (MCKP) by applying a surrogate condition for cost. In the MCKP, there are several groups of items. It is required that one item is selected from each group, so that the total benefit is maximized without exceeding the capacity of the knapsack. For finding a feasible solution and enhancing it in a short time, the algorithm in [7] is considered to be a good choice. In our approach, however, we use a genetic algorithm. It can decide already in the first step whether a feasible solution exists or not. Enhancing the solution is bounded by a specific time.

In [8] a heuristic algorithm is used for solving the MMKP by using convex hulls. The idea is to simplify the MMKP into MCKP. This is done by multiplication of a transformation vector. Once the MCKP is constructed, each group of items can be represented on X-Y Axes. X represents the resources used by the items. Y represents the benefit that should be maximized. An initial solution is found by selecting an item from each group. The selected item is the one with lowest benefit. After that, three iterations are done. In each iteration, the penalty vector is used to turn each of the resource consumption vectors into a single dimension vector. The frontier segments of the items are calculated. “A segment is a vector with two items representing a straight line.” [8]. According to the angle of the segment, it is ordered within the list of segments. The segments should be put in a descending order. For each segment, P1 and P2, the items associated with the segment, are considered. A current solution is calculated by selecting the item associated with P1 and the same is applied for P2. If utility of the current solution is smaller than the utility of the saved solution, the saved solution is kept. Penalty is adjusted for the next iteration. After iterations are done, if the current solution is not feasible, then no solution is found, else the current solution is set to be the final solution. Following this method requires the values and weights to be known before penalty vectors and convex hulls are constructed. In our approach this is not possible because a pre-knowledge of members to be selected in each group are required to calculate values of weights. The reason is that one of the considered weights requires for its calculation the deadline, which can be calculated according to TBS [6]. The calculation of a deadline, which belongs to a selected item in a group depends on the deadlines of selected members in previous groups.

In [10], a reduce and solve algorithm is used for solving MMKP. The approach depends on group fixing and variable fixing to reduce the problem. Then it runs CPLEX [14] to solve the reduced problem. Also here it is required to know the benefits and weights before start solving. In our approach this is not possible.

In [9], three algorithms are introduced to solve MMKP. The first algorithm tries to find an initial solution for the guided local search. It is applied by assigning a ratio for each item in the groups. The ratio is the value divided by the Scala Product of the weight of the item, and total capacity of the knapsack. Items, which own best ratios are selected. If a feasible solution is not reached, then an item with heaviest ratio is chosen to be swapped with another item from the same group. If this does not result in a feasible solution, then the lightest item from the same group is selected. This iteration continues until a feasible solution is found. The initial solution can be enhanced by applying the second algorithm, a complementary constructive procedure (CCP). It consists of two stages. The first stage swaps selected items with other items within their groups to enhance the already found feasible solution. If the resulting solution is feasible then the swapping is considered to be valid. The second stage replaces the old item by the newly selected one. The third algorithm is the derived algorithm. It starts by applying the constructive procedure of the first algorithm in order to get an initial feasible solution. If the solution cannot be improved by CCP, a penalty parameter is applied to transform the objective function, and a new solution is acquired by CCP. If the new solution achieves improvement, then a normalization phase is applied to get the original values of profits. If it does not achieve any improvement, then a normalization phase is applied and a penalty is applied again. The iteration continues until a stopping condition is reached. In the previously three described algorithms, it is required to know the benefits and weights before start solving. In our approach this is not the case.

In [11], a reactive local search algorithm to solve MMKP is presented. A constructive procedure (CP) and a complementary procedure CCP are used to acquire an initial solution. CP uses a greedy procedure to acquire the initial solution. CCP enhances the solution acquired by CP. The enhancement is done by following an iterative approach that swaps elements in the same class (group). The reactive local search (RLS) is applied to enhance the solution acquired by CCP. RLS consists of two procedures. The first one is called degrading strategy, and the second one is called deblock procedure. The degrading strategy consists of three steps. It selects an arbitrary class, changes arbitrary elements inside it if this exchange will result a feasible solution, repeats steps 1 and 2 several times, and terminates with a new solution. The deblock strategy starts by constructing a set of elements. Each element consists of two classes. The strategy runs a loop on the set until no element exists in it. In each run of the loop, it investigates if there is a couple of items in both chosen classes, so that the objective

value of resulting feasible solution is better than the previous solution value. The resulting feasible solution considers the couple of items in chosen classes, in addition to the fixed items, which belong to the classes other than the chosen ones. The deblock strategy exits with the best solution. [11] describes also the modified reactive local search algorithm (MRLS). It replaces the deblock procedure by a memory list to enhance computation time. In RLS and MRLS, again it is required to know the benefits and weights before start solving. In our approach this is not the case.

Evolutionary algorithms have been studied in a couple of previous works for solving different kinds of knapsack problems. For example, in [12], a general approach of using genetic algorithm to solve MMKP is described. It starts by selecting an initial population. This can be done randomly. The fitness of the population is evaluated according to the fitness function. The fitness function is defined according to the objective function of the knapsack problem. Then a loop runs until a predefined condition is satisfied. In each iteration, a new population is selected from the previous one using the roulette wheel selection [15]. Crossover and mutation are applied. The resulting population is evaluated. The current generation is combined with previous one. Finally, the resulting individuals are ordered to find a best solution. The algorithmic principle in [12] is similar to our approach, however, the selection process differs from the selection process in our approach. In our approach, we can determine if a feasible solution exists once we construct the first individual. Further steps work only on optimizing the solution. In [12], it is not explained if one can recognize the existence of a feasible solution once the first individual is constructed.

In [13], an evolutionary algorithm is used to solve MMKP. The idea is to solve a manufacturing problem. Operators should be distributed among machines to process components of products. This has to be done in an efficient way to keep work hours within a predefined limit. Operators differ regarding their experience or working ability. To model this situation, binary coded chromosomes are constructed. Each chromosome has three dimensions: machines, operators and components. Chromosomes are transferred into two dimensional chromosomes. A first generation of chromosomes is chosen. It should contain only feasible solutions. A loop is run on generations. In each run, a new generation is generated by applying selection and mutation on the previous generation. The loop continues until a predefined condition is satisfied. The algorithmic principle in [13] is similar to our approach, however the principle of setting the fitness function and the principle of the selection process differ from our approach. In our approach, we can determine if a feasible solution exist once we construct the first individual. Further steps work only on optimizing the solution. In [13], it is not explained if one can recognize the existence of a feasible solution once the first individual is constructed.

The comparison in this section between our approach and the previously introduced approaches points out that our approach provides more flexibility in terms of solving a broader set of problems where parameters can be calculated at runtime, and the existence of a feasible solution can be known in a very early stage of the algorithm.

In the next section, we present the basic concepts of RT operating systems, the knapsack problem, and the genetic algorithms.

III. Basic Concepts

RT systems are computing systems, in which the correctness of behavior depends not only on the computation results but also on the response time. “Examples of RT systems could be automotive applications, flight control systems, robotics, etc” [16].

- A real-time task: is characterized by many properties. In the following, we mention some of them:

1) Arrival time (a): it is the time at which the task is released and becomes ready for execution, called also release time. [16]
2) Execution time (C): is the time required to execute the task without interruption. [16]
3) Absolute deadline (d): is the time that the task execution should not exceed. [16]
4) Relative deadline (D): is the time difference between the absolute deadline and the arrival time. [16]
5) Start time (s): is the time of starting the execution of a task. [16]
6) Finishing time (f): is the time of finishing the execution of a task. [16]
7) Criticalness: is a parameter that indicates whether the task is hard or soft. [16]

- A soft RT 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. [16]

- A hard RT task: is a task that may cause catastrophic results in the system environment if its deadline is not met. [16]

- A periodic task: is a task that is activated in regular time periods, where each activation is called an instance of the task. [16]

- An aperiodic task: is a task that is activated in irregular time periods, where each activation is called an instance of the task. [16]

- 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) [16]. DAG has no directed circles [17].

The knapsack problem is an NP-hard problem. In this 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 [18]. There are several types of knapsack problems [18]:

1) 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.
2) 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.
3) 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.

4) 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.
5) Multidimensional 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.

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 1. [19]

A genetic algorithm is a type of evolutionary algorithms. It consists of the following steps: Initial population, evaluation, fitness assignment, selection, and reproduction. [25]

1) Initial population: In this step the initial population is chosen. The initial population consists of individuals. [25]
2) Evaluation: Evaluates the current population according to an objective function. [25]
3) Fitness assignment: Determines the fitness of the population. [25]
4) Selection: Selects the fittest individuals for the reproduction process. [25]
5) Reproduction: Applies crossover and mutation to generate new individuals. [25]

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 randomly, by going through an algorithm, or other methods.

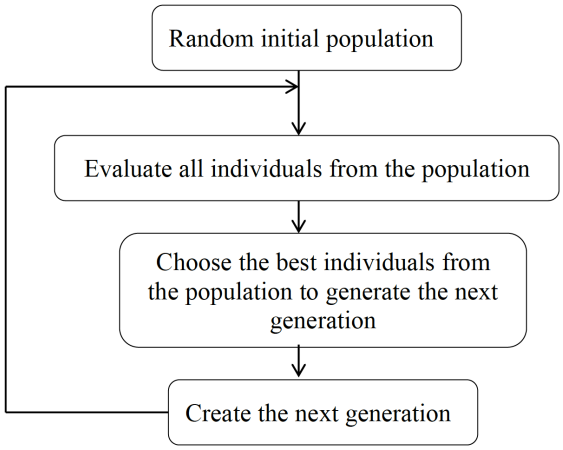

Figure 1. Evolutionary Algorithms. [19]

Boundedness is equivalent to the fact that the algorithm terminates after a finite time whenever being started. To guarantee boundedness, the following conditions should be satisfied:

1) There must be no external influences which are not under control of the algorithm.
2) There must be no deadlocks and no unbounded blocking.
3) There must be no while/until loops which are not terminating: A classical test in this case is checking whether the function to be calculated is bounded, monotonic 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, monotonic and asymptotic functions:

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, $f(x) \leq K$." [20]

Monotonic functions: " A function is monotonically increasing if for all x and y, such that $x \leq y$ one has $f(x) \leq f(y)$, so f preserves the order. Likewise a function is called monotonically decreasing if whenever $x \leq y$ then $f(x) \geq f(y)$, so it reverses the order " [21]

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). [22]

In the next section, we introduce the robotic surgical system as an appropriate example for an RT application, where our approach can be applied.

IV. Scenario

Let us assume a telerobotic surgery system [23], 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. 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. The surgeon side is the Master side. The patient side is the Slave side. See Figure 2. On the Master side, the robotic surgical system provides a vision system that translates the information coming from the Slave side. 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 to adapt itself to the evolutions of surgical actions is limited by the surgeon 's ability to react to these evolutions with the required speed so that the operation is performed successfully.

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 its structure and behavior at run-time to enable the system to adapt itself to environmental changes on the Slave side. The adaptation process should preserve all RT constraints. Here, the overhead imposed by the adaptation process itself has to be considered as well. 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. The measurements of the patient as e.g. pressure, temperature, view of the surgical field, etc. represent the internal parameters. 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. number and kinds of surgical instruments, endoscopic tools, medical equipment, etc, represent the external parameters. This scenario is an example for an RT system, where our approach can be applied. 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 original task.

In the next section, we describe our solution. First, we introduce the concept of RT cells and their properties. Afterwards, we list the steps of the algorithm, and proof of boundedness for each step.

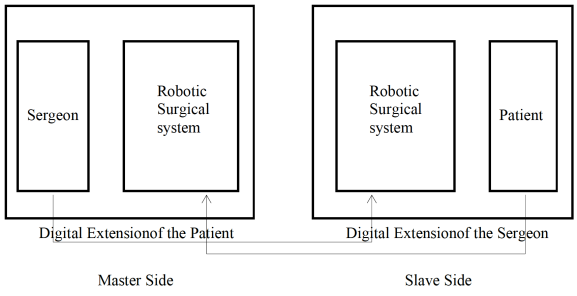

Figure 2. Telerobotic Surgery. [2]

V. SOLUTION

We assume a RT system, which consists of periodic and aperiodic tasks. The scheduling technique to be applied is EDF with relative deadline equal to the period. The aperiodic tasks are served by TBS. All tasks are augmented by additional properties, enabling them to be adapted online. Such an augmented task is denoted by the term "Cell".

We define the $Hyperperiod$ to be an amount of time, that is initially calculated according to the periodic and aperiodic load in the system. The $Hyperperiod$ guarantees a point of time, at which all periodic instances can start their execution. This point of time is the end of current hyperperiod and the beginning of next hyperperiod. The $Hyperperiod$ might change each time the adaptation algorithm takes place. NHP is the point of time at which a hyperperiod ends.

We assume that an adaptation can take place only once per hyperperiod and becomes effective not earlier than the next hyperperiod. The adaptation algorithm is executed by a periodic task with a period equal to the $Hyperperiod$. We define two types of RT cells, the controlling RT cells, and the controlled RT cells. The first one should be able to change the structure and behaviour of the second one. In our approach, we define the EC as the only controlling RT cell in the system. It exists before the system starts. Any other cell in the system is a controlled RT Cell, and abbreviated as RTC. EC becomes an Active Engine-Cell (AEC) once it is activated. An RTC becomes an Active RTC (ARTC) when it is accepted for execution. Each cell inherits the characteristics of RT tasks, and has an additional set of properties. This set enables the organic behaviour to be applied.

A. EC

EC is a periodic cell with period initially calculated as under paragraph 4 in the properties of EC. In the following, we list the properties of the EC:

1) $EC-ID$: the ID of the EC. Each cell in the system has a unique ID.
2) $WorstCaseExecutionTime$ ($WCET_{EC}$): is the worst-case execution time of the AEC.
3) $WorstCasePeriod$ (WCT_{EC}): is the worst-case period of the AEC.
4) $Hyperperiod$: The initial hyperperiod is calculated as the initial NHP (see the calculation of initial NHP on page 7).
5) $NumOfPARTCs$: is the number of periodic ARTCs in the system.
6) $NumOfAARTCs$: is the number of aperiodic ARTCs in the system.
7) $Cost$: is the cost that AEC is assumed to consume. The cost is seen as a function of quality factors.
8) $Active$: is a Boolean variable. It is set to true when the system starts. Whenever the system stops executing, EC becomes not active, and the variable is set to false.

9) $RTCArray$: is the data structure that holds the different cells that exist on the local node and their variants. New cells can be added to the $RTCArray$ at run time. Also, current cells can be updated. Each column is called an $RTClass$. Each $RTClass$ holds a number of variants, which are RTCs dedicated to fulfil the same principal functionality, with different cost and time characteristics. All periodic variants, which belong to the same class, have the same period. The upper bounds of $RTCArray$ dimensions may change online, according to system resources.

B. RTC

An RTC has the following properties:

1) $RTClassID/VariantID$: is a unique ID that differentiates an RTC from other RTCs in the system. Here, $RTClassID$ is the ID of a class of RTCs. In the RTCArray, a different $RTClassID$ is assigned to each column. The $VariantID$ differentiates the different RTCs in the same class (column).
2) $VariantsAllowed$: is a Boolean property that expresses if an RTC is mandatory or not when it should be tested for acceptance by the system. When it is equal to true, all variants that belong to the class of respective RTC should be examined to select the most appropriate 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 considering 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 can be substituted by another variant from the RTCArray. The 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 counterpart point in all the other considered sets. In case of periodic cells, we make a restriction to natural updating points, i.e., the release time of the next instance [4]. Aperiodic RTCs may have a sequence of updating points. 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 computation time between the starting point of the task and 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 the 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 disappears 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 starting the current hyperperiod. We assume that this value is always provided by the underlaying RT operating system (RTOS). $ET_{executed}$ 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 (selected to construct an insertion or deletion request). Otherwise, it is not triggered. Whenever a decision is taken about an RTC to be accepted or not, 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 $RTCArray$ to construct a request). 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 always to set the arrival time of periodic requests 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). Our goal is also to set the arrival time of aperiodic requests greater or equal to NHP.
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 should 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.
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 $ImportanceFactor$. The $ImportanceFactor$ is considered for filtering 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. Implicitly this means that its importance factor is infinitely high.
14) $Cost$: is an abstract concept. It includes a variety of possible constituents, e.g., memory demand or provided quality like precision of computation. 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.
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.

16) $Type$: the type of an RTC could be periodic or aperiodic. All variants, which have the same $RTClassID$ have the same value of property $Type$.
17) $Cost\_Update$: the updated cost, which should be calculated for an RTC, when it replaces another executing RTC.

C. Complexity variables

The algorithm is bounded if each step needs a bounded time. Time complexity depends on a set of variables.

The variables are:

1) h: The upper bound of the number of columns in the $RTCArray$ (number of $RTClasses$).
2) f: The upper bound of the number of RTCs in an $RTClass$.
3) b: The upper bound of the newly deployed RTCs.
4) PN: an upper bound of number of parameters in the system.
5) QB: The upper bound of requests that can be received in each execution of the EC.
6) SC: The upper bound of the dependent cells, which may construct a request.
7) $m1$: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.
8) n: The upper bound of the number of updating points in a cell.
9) GRP: The value of the greatest period available among the periods in the $RTCArray$, and the expected period of the EC.
10) $NInd$: Number of individuals in a generation within the genetic algorithm solving the Knapsack problem.

All parameters have a predefined upper bound, which guarantees boundedness of computation time.

D. Adaptation Algorithm

The following terms are used in the algorithm:

- $ExpPARTCs$: is the set of periodic ARTCs excluding deletion requests.
- $ExpAARTCs$: is the set of aperiodic ARTCs excluding deletion requests.

Calculating an initial NHP is carried out either offline or when starting the system. The initial NHP is calculated as follows:

WCT_{EC} is initially set to the least common multiple of periods of periodic cells in the system. 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 common 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:

$WCET_{EC}/(lcm\_initial \times factor) = 1 - Sum1 - Us\_initial$.

By resolving this equation for factor we obtain

$$factor = \lceil (WCET_{EC}/lcm\_initial \times (1 - Sum1 - Us\_initial)) \rceil \quad (1)$$

The initial NHP is equal to $lcm\_initial \times factor$. This value is set as a period of the EC.

Calculating the initial NHP shows a trade-off. 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 assigned to the EC. A high utilization consumed by EC may serve more requests but with longer reaction time. The execution time of the EC depends on b and QB. For this reason, setting b and QB by the system administrator plays a role in this trade off. The execution time increases as these parameters increase. The $WCET_{EC}$ of the EC depends on a couple of parameters. The respective function will be presented at the end of this paper. It is assumed that based on this function and an appropriate model of the underlying hardware the resulting $WCET_{EC}$ can be estimated with sufficient precision.

Each time the EC is executed, following steps take place:

Step 1: Gathering and Filtering the newly deployed RTCs:

The first step of the AEC is to collect the newly deployed RTCs. It stores them in a $WorkingRTCArray$ (a copy of $RTCArray$) following a procedure that ensures to keep the upper bound of the $WorkingRTCArray$ dimensions preserved. Newly deployed RTCs enlarge the solution space when applying the adaptation algorithm. Let b be the upper bound of newly deployed RTCs that can arrive at this step. 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 in each class of the $WorkingRTCArray$, and h is the upper bound of the different $RTClasses$ that can be stored in the $WorkingRTCArray$. If the newly imported RTCs may cause exceeding the upper bound of variants in a column, or the upper bound of columns, the EC preserves the upper bounds by applying a filter procedure. In this context, we discuss following different cases:

1) If the upper bounds of influenced dimensions in $WorkingRTCArray$ will not be exceeded, the RTCs can be added to the $WorkingRTCArray$. See Figure 3, Figure 5 and Figure 6.
   If upper bound of classes is exceeded, one possible heuristic for replacing $RTClasses$ is the ImportanceFactor-based approach. The $RTClasses$ that could be chosen to be replaced by the newly arrived ones are the classes that are not essential or activated. We exclude the classes with the smallest $ImportanceFactor$. For example, let us suppose that there exists in $WorkingRTCArray$ 20 $RTClasses$. All $RTClasses$ have an $ImportanceFactor$ equal to 3, except the third $RTClass$. It has an $ImportanceFactor$ equal to 1. The upper bound of

classes is 20. If we have to add a newly deployed $RTClass$ with an $ImportanceFactor$ equal to 5, and the third $RTClass$ is not essential and not activated, then we can replace the third $RTClass$ by the newly arrived one.

If RTC adds a new periodic variant to an existing column and the upper bound of variants in the column is exceeded, then a decision should be taken, which RTC to drop. One option may be to examine the RTC with the highest C_i/T_i. If it does not result in a successful schedulability test together with the AEC, we exclude it. If it results in a successful schedulability test, we exclude the RTC with the highest cost. For example, let us suppose that there exists 90 variants in the column, where the newly deployed RTC should be added. The upper bound of variants in the column is 90. If the utilization needed by the newly deployed RTC is 0.3, and the highest utilization needed by the variants in the column is 0.7. If the utilization needed by the AEC is 0.6, then we exclude the RTC, which needs the utilization of 0.7. If, however, the highest utilization needed by the variants in the column is 0.2, then we exclude the variant with the highest cost among variants in the column and the newly deployed RTC. If the cost of the newly deployed RTC is 15, and highest cost of variants in the column is 20, we exclude a variant that has the cost 20.

In case the newly deployed RTC adds a new aperiodic variant to an existing column, and this will cause exceeding the upper bound, we exclude the RTC that consumes the highest cost. An arbitrary exclusion can also take place.

2) If a newly arrived column should update a specific existing column, and no variant is active in the existing column, we substitute it by the newly arrived one. See Figure 3 and Figure 4. If there is an active variant in the existing column, we add the newly arrived RTClass to a queue to be considered later. This $UpdateQueue$ may be ordered by the arrival times or an arbitrary other criterion. The upper bound of its capacity is equal to QB. Each element of the $UpdateQueue$ is an array. The array includes the newly deployed $RTClass$, in case this $RTClass$ does not have dependencies. In case a set of dependent $RTClasses$ arrives, the array includes more than one column, the newly deployed $RTClass$ and the other $RTClasses$ in the dependency graph.

Boundedness proof:

- Transmitting the newly deployed RTCs is bounded by b and time for transmission. As we assume a deterministic communication channel between the remote node where newly deployed RTCs reside and the local one, the transmission time is bounded. - The EC applies a filter procedure to preserve upper bounds of the $WorkingRTCArray$. The filter procedure is bounded by one of the $WorkingRTCArray$ dimensions. The formal proof of the boundedness of this step and the next steps is clear when looking at Nassi-Schneiderman diagrams which represent the steps. In [2], time complexity appears on each diagram, and this in turn points out that the step specified by the diagram is done in a bounded time, and the bound depends only on the parameters, which participate in the time complexity formula.

Step 2: Triggering and handling the newly arrived requests:

In the previous step, an $UpdateQueue$ has been constructed, including update requests. In this step, another queue

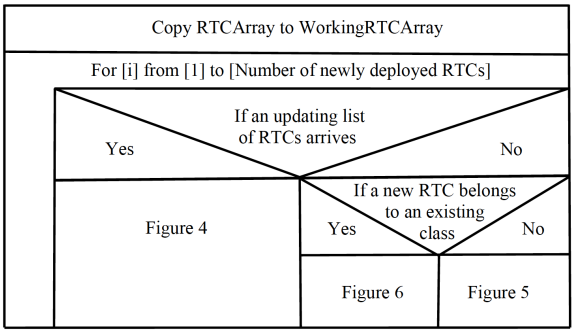

Figure 3. Nassi-Schneiderman Diagram for gathering and filtering the newly deployed RTCs [2].

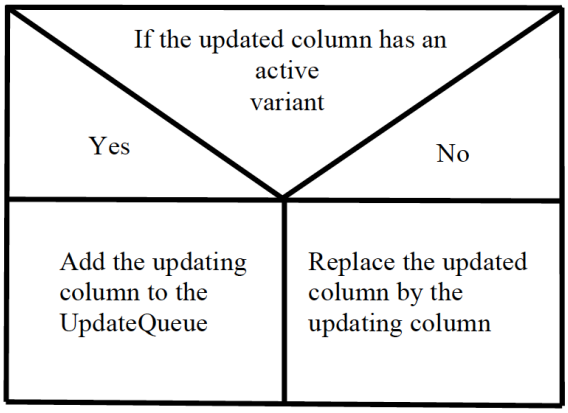

Figure 4. Nassi-Schneiderman Diagram for the arrival of an updating list of RTCs [2].

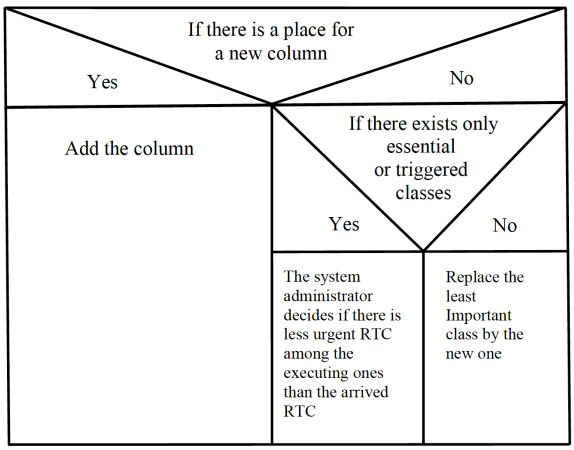

Figure 5. Nassi-Schneiderman Diagram for adding a new column [2].

is constructed. It is called the $TriggeredQueue$, ordered by arrival time or any other criterion. Requests, that are added to this queue, are chosen from the $WorkingRTCArray$. Triggered requests may add or delete RTCs. Triggered requests are chosen according to the necessities of the system. The upper bound of the $TriggeredQueue$ capacity is QB. The EC makes an iteration over items in the $UpdateQueue$ and the $TriggeredQueue$ in parallel. It selects by an arbitrary criterion either an update request or a triggered request. For simplicity, the decision can be made arbitrarily. This selection process is iterated until the number of requests is equal to the upper bound or until no further requests exist. See Figure 7 and Figure 8. Selected requests will be stored in a $RequestQueue$ of bounded size.

If the arrival time of the periodic requests in the $RequestQueue$ is not equal to NHP, their $TriggeringRange$ is examined. If $TriggeringTime \leq$ NHP $\leq TriggeringTime + TriggeringRange$, then the arrival time is set to NHP. Otherwise the requests, which do not satisfy the previous condition, are not accepted and deleted from the $RequestQueue$. After that a notification is sent to the system administrator. For example, if NHP = 10. The arrival time of the request is equal to 9. Its triggeringRange is equal to 5, and the request has been triggered at time 8, we notice that the arrival time of the request is not equal to NHP. For this reason, we examine if $TriggeringTime \leq$ NHP $\leq TriggeringTime + TriggeringRange$. We notice that 8 $\leq$ 10 $\leq$ 8 + 5, so we set the arrival time to 10. The $DeletionTime$ of periodic requests that have to be deleted is set to next natural updating point. If 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.

If the request includes a set of dependent cells, we assume that their modified arrival times and deadlines are calculated offline by EDF*. If one of the modified arrival times is smaller than NHP, then a fixed offset is applied to all arrival times and deadlines to keep them greater or equal to NHP. See Figure 9.

An update request is represented by an array of RTCs that constructs the $RTClass$ of the update. Updating a set of dependent cells is done under the same rules as updating a cell.

When requests in the $RequestQueue$ proceed for processing by the AEC, the buffers ($UpdateQueue$, $TriggeredQueue$, and $RequestQueue$) become empty.

Boundedness proof:

- A queue of triggered requests (add/delete requests) is constructed in a time bounded by QB.
- The first and second iteration over $UpdateQueue$ and $TriggeredQueue$ is bounded by QB.
- Setting the arrival time of requests is bounded by a constant time.

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

A part or the whole set of local parameters might influence the overall quality of the system. This set of parameters includes both parameters of the underlying computing system and of the RT system under consideration. Parameters of the

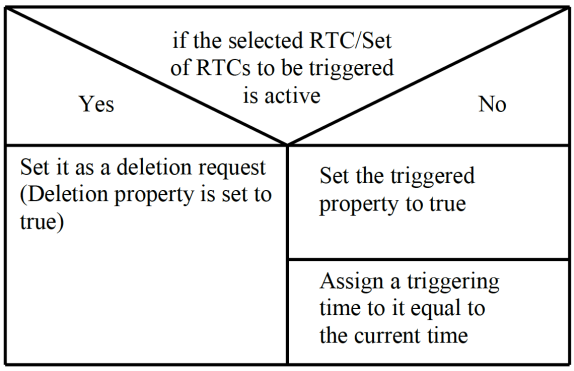

Figure 7. Nassi-Schneiderman Diagram for triggering a request.

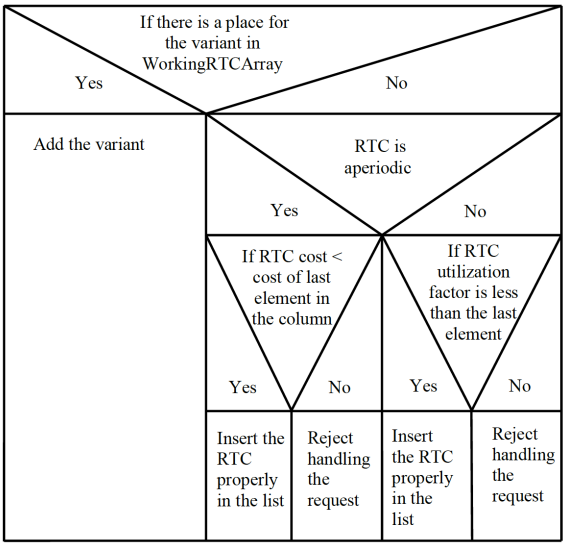

Figure 6. Nassi-Schneiderman Diagram for adding an RTC to an existing column [2].

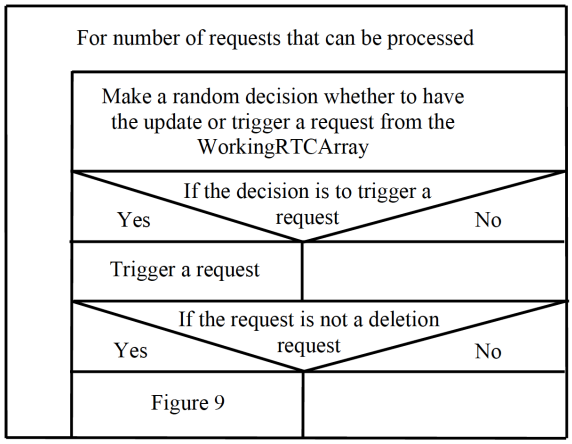

Figure 8. Nassi-Schneiderman Diagram for choosing between an update and a triggered request.

system should be read in each execution of the EC because they might change. This change may affect the result of the adaptation process. E.g., adding new resources may allow accepting a set of requests, which cannot be accepted with less resources. The result of calculating the total cost depending on quality parameters available is called $Cost_{total}$. For example, let us assume that in a remote surgical system, only the following set of parameters are considered: number of cameras, number of robotic arms and number of endoscopic tools. Let us assume that each of these parameters has a weight. Number of cameras is equal to 3. Number of robotic arms is equal to 10. Number of endoscopic tools is equal to 8. The weight of the first parameter is 1. The weight of the second parameter is 4. The weight of the third parameter is 2. Let us assume that $Cost_{total}$ is given by the following function: t $Cost_{total}$ = first parameter $\times$ weight of first parameter + second parameter $\times$ weight of second parameter + third parameter $\times$ weight of third parameter = 3 $\times$ 1 + 10 $\times$ 4 + 8 $\times$ 2 = 3 + 40 + 16 = 59.

Boundedness proof: Under the assumption that an arithmetic operation can be carried out in bounded time, and number of system parameters is bounded by PN, the entire calculation can be carried out in bounded time.

Step 4: Adaptation algorithm:

In this step, we calculate the lowest cost feasible solution over the entire set of $RTClasses$ stored in $AdaptationRTCArray$. The $AdaptationRTCArray$ is constructed as follows:

Constructing AdaptationRTCArray:

1) We copy the variants of the $WorkingRTCArray$ into a temporary array $AdaptationRTCArray$. The $WorkingRTCArray$ is essential for enabling a transaction concept. If the adaptation process turns out to be successful, the $WorkingRTCArray$ will replace the $RTCArray$. If the adaption fails, the $WorkingRTCArray$ will be neglected and the system returns to the previous version.

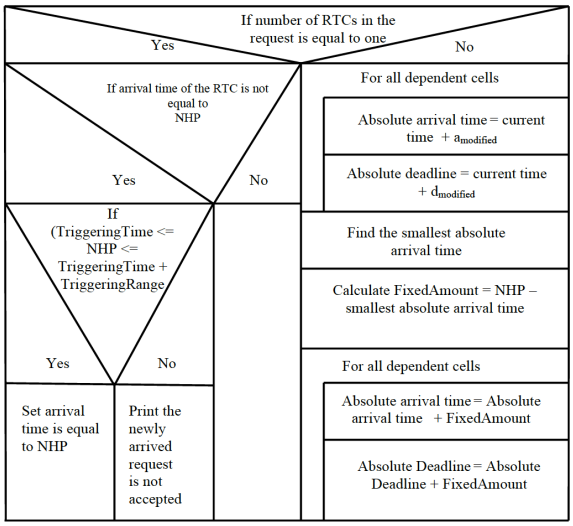

Figure 9. Nassi-Schneiderman Diagram for setting time characteristics of triggered requests [2].

2) We reduce $AdaptationRTCArray$ to contain only the variants, which $RTClassID$ exists in the $ExpPARTCs$ and $ExpAARTCs$ with absolute deadlines exceeding NHP. For each $ExpPARTC$ or $ExpAARTC$, which has the property $VariantsAllowed$ set to false, we do not consider variants that hold the same $RTClassID$ in $AdaptationRTCArray$, other than the ARTC itself.
3) For each aperiodic ARTC that should be deleted, and has absolute deadline 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 reason is that updating points are the most suitable points to apply deletion, as partial results are delivered on these points. Deleting a cell suddenly on an arbitrary point may cause errors.
4) We then add a column that includes the AEC.
5) We also add the newly triggered requests. If their properties $VariantsAllowed$ are set to true, we add columns that represent $RTClasses$ 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.
6) In case there is an update request for an 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 updating $RTClass$ that includes the triggered updating 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: First, we check whether in the current hyperperiod there is capacity left to execute aperiodic ARTCs with deadlines that exceed NHP. This capacity is called "$Amount$".

$$Amount = Hyperperiod - [(\sum_{i=1}^{NumofPARTCS}((Hyperperiod/T_i) \times C_i)) + WCET_{EC} + \sum_{i=1}^{NumofAARTCS-NumOfANHP}(C_i - ET_{executedi})]$$

$NumOfANHP$: refers to the number of aperiodic ARTCs with deadlines that exceed NHP. Based on the value of $Amount$ we now can identify those ARTCs, which definitely result in a completion time later than the current hyperperiod and having an update point after NHP.
$Amount1 = Amount$.

/* We construct a vector of the running aperiodic ARTCs, which deadlines exceed NHP. In the following loop i indicates the i_{th} item in the

vector.*/

If ($Amount1 > 0$) then {
For (i = 1 to $NumofANHP$) {
If ($C_i - ET_{executedi} \leq Amount1$) then {
$Amount1 = Amount1 - (C_i - ET_{executedi})$
}
else {
$C_{i,new} = (C_i - ET_{executedi}) - Amount1$.
If there exist an updating point in $C_{i,new}$, add this ARTC to the set of variants that can be updated
}
}
}
else {
For (i = 1 to $NumofANHP$) {
$C_{i,new} = C_i - ET_{executedi}$
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} variant in the updating $RTClass$ is set to the arrival time of the updated variant. The execution time for the j_{th} variant is set to (y + C_j - $\grave{y}$), where $\grave{y}$ 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 - $\grave{y}$) + (Arrival time of the running variant + y)), Absolute Deadline of the running variant that should be updated]

7) 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 should exist 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, when a periodic update is applied in the next hyperperiod, no execution of the updated variant can take place.
8) In case there is an update request for a set of aperiodic dependent RTCs, modified arrival times and deadlines are calculated offline. When calculating time characteristics of the variants in the columns that are supposed to update dependent variants, same rules of updating one variant are applied.

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 selection process. The columns in the array are reordered, so that periodic columns comes first, then AEC, and finally aperiodic columns.

Let us assume that:
the number of columns in $AdaptationRTCArray = Num$.

$\grave{N}$ is the number of columns, which represent the newly triggered aperiodic requests. They are placed as last columns in $AdaptationRTCArray$.

If ($NumOfANHP > 0$) then {

/*In the following, we calculate arrival times, execution times, and $Cost-Update$ for the running aperiodic ARTCs that are stored in $AdaptationRTCArray$, with deadlines exceeding NHP.*/

If ($Amount > 0$) then {

/* 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 that do not exceed NHP.*/

/*We construct a vector of the running aperiodic ARTCs, which deadlines exceed NHP. We order the elements of this vector according to the increasing absolute deadlines. In the following loop i indicates the i_{th} item in the constructed vector.*/

For (i = 1 to $NumOfANHP$){
If ($C_i - ET_{executedi} \leq Amount$) then {
$Amount = Amount$ - ($C_i - ET_{executedi}$)
Exclude the column of the ith aperiodic variant from $AdaptationRTCArray$. Decrease Num by 1.
}
else{
$C_{i,new} = (C_i - ET_{executedi}) - Amount$.
Set 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 $NumofANHP$){
$C_{i,new} = C_i - ET_{executedi}$
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-$\grave{N}$){

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 are 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:

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

$\grave{C}$ 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.

$\grave{y}$ 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.

}

}

Up to know we have prepared the current ecosystem of RTCs, within which we have to find a valid solution with minimized overall cost by making a proper selection of variants.

To find the solution, we solve the following multiple-choice multidimensional knapsack problem:

/*The fist constraint of the knapsack guarantees minimizing the cost of the solution. The second constraint of the knapsack guarantees the schedulability of periodic and aperiodic cells with hard deadlines.*/

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

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: $Factor1 = C_{ij}$, $Factor2 = T_{ij}$

For the AEC, $Factor1 = WCET_{EC}$, $Factor2 = WCT_{ECtemp}$

For any of the aperiodic RTCs: $Factor1 = 0$, $Factor2 = 1$

WCT_{ECtemp} is calculated as follows:

The expected hyperperiod is calculated as the least common multiple of periods of periodic $ExpPARTCs$ in $AdaptationRTCArray$, and periods of the newly triggered periodic requests in $AdaptationRTCArray$. The resulting value is set as initial value for the expected period of AEC. 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.

W_{ij}^2 is calculated here in a way different from [1]. In [1], W_{ij}^2 is negative if there is an aperiodic lateness. The sum of weights as stated in the knapsack problem is defined as the aperiodic lateness. If this lateness is smaller than zero (the related knapsack constraint succeeds), the aperiodic lateness indicates a deviation from optimal case of meeting hard deadlines of aperiodic RTCs.

In this paper, we set W_{ij}^2 to be zero if hard deadlines are met ($d_{Calculated,ij} \leq d_{Specified,ij}$). Otherwise it will be equal to a positive value expressing the aperiodic lateness. The sum of weights as stated in the knapsack problem is defined as the aperiodic lateness. If the related knapsack constraint succeeds, hard deadlines of aperiodic RTCs are

met.

$W_{ij}^2 = Factor1 - Factor2$ if $Factor1 > Factor2$, otherwise $W_{ij}^2 = 0$.

For any of the periodic RTCs and the AEC: $Factor1 = 0$, $Factor2 = 0$.

For any of the aperiodic RTCs:

$Factor1 = d_{Calculated,ij}$, $Factor2 = d_{Specified,ij}$.

Where:

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

$d_{Calculated,ij} = \max\{d_{Calculated(i-1)j_{i-1}}, ArrivalTime_{ij}\} + C_{ij,new}/U_s$.

$d_{Calculated(lp_l)} = 0$.

Where;

$p = j$

l = Number of periodic columns in $AdaptationRTCArray$ +1

$Us = 1 - Up$.

Depending on the different kinds of RTCs to be considered in solving the Knapsack problem, W_{ij}^3 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.

The algorithm we are applying to solve the knapsack problem is a genetic algorithm. See subsection E for further details.

If the newly arrived requests are accepted by the system, the ARTCs set or subset, which is represented in $AdaptationRTCArray$ is 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) should be stored. 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 5 is applied.

E. Genetic Algorithm

The algorithm we are applying to solve the knapsack problem 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, 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 periodic RTClass the variant with the lowest respective utilization, and from each aperiodic $RTClass$ the variant with lowest respective execution time. The low utilization of a periodic RTC rises the chance of making the sum of all periodic utilizations smaller or equal to 1. The low execution time of an aperiodic RTC rises the chance that the calculated absolute deadline, which is calculated according to TBS, becomes small. As a result, the chance of meeting the hard deadlines of periodic and aperiodic RTCs becomes higher. The second individual is given by the current selection of variants for all $RTClasses$, which are not affected by the adaptation together with all adaptation requests included in the first individual. The first initial individual allows a simple decision whether a solution exists, as if this individual does not fulfil the constraints then there cannot exist any solution. The reason is that we choose the variants in the first individual in a way that reaches the highest chance of satisfying the schedulability test because the periodic utilization is at lowest amount and the calculated aperiodic deadlines according to TBS are at lowest values. 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 $\leq$ upper bound of number of variants in a class in the $WorkingRTCArray$. The remaining individuals of the first generation may be chosen by any procedure, e.g., by randomly exchanging the selected variants in the columns.

After that WCT_{ECTemp}, server utilization, and absolute deadlines for aperiodic load are calculated for each individual according to TBS [6]. 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 preliminary optimum.

If the knapsack constraint $\sum_{i=1}^{Num} \sum_{j=1}^{n_i} W_{ij}^k x_{ij} \leq R^k$ 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). The improvement of the solution is guaranteed by keeping the fittest individuals in the next generation. Choosing an arbitrary variant when applying the mutation may enhance the solution more than choosing a variant with specific characteristics, because there is no characteristic that can guarantee enhancing the solution. We did not apply recombination in our approach. Applying it is a possible option. However, in this case, we should ensure to keep a specific number of individuals in each generation after the selection process, which may enforce keeping a number of constraint-violating individuals in the next generation. This is necessary for the recombination process to take place, because we should assume to have at least two individuals in the previous generation. Figure 10, Figure 12 and Figure 13 describe the solution. Figure 11 is part of the process in Figure 12.

Boundedness proof:

- Operations Step 4 other than the genetic algorithm are bounded by f, h, or f and h.

- The genetic algorithm is bounded because of the following reasons:

1) The operations dedicated to calculate WCT_{ECTemp}, server utilization, and absolute deadlines for the aperiodic load in each individual are bounded by $NInd$, upper bounds of $RTCArray$ dimensions.

2) We can decide whether there exists a feasible solution or not in bounded time. Feasibility can be decided already based on the first generation.

3) The individuals of a generation are sorted by increasing total costs. Sorting is bounded by $NInd$.

4) 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.

Step 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 of the underlying RTOS. 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 its properties, e.g., WCT_{EC} is set to the temporary value WCT_{ECtemp}.

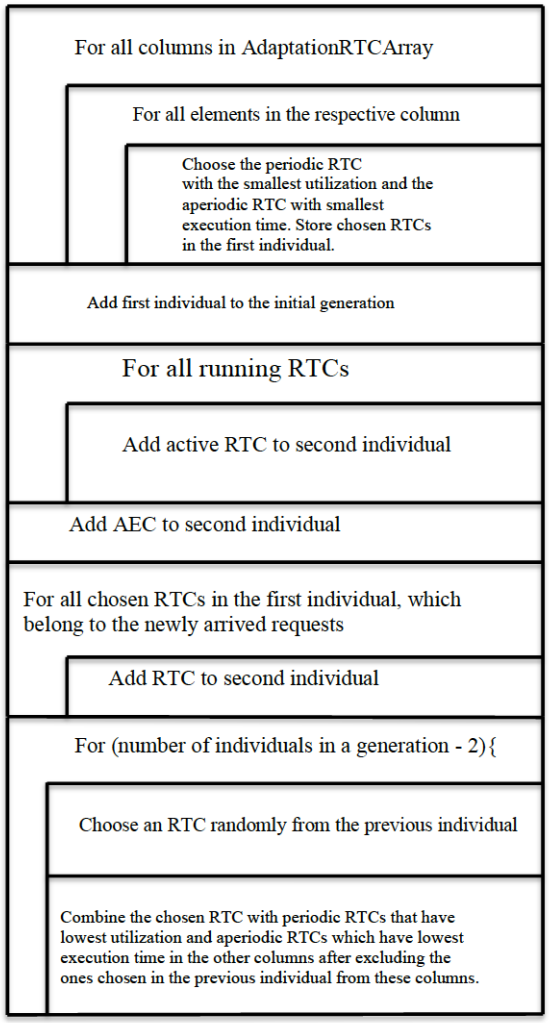

Figure 10. Nassi-Schneiderman Diagram for choosing the initial generation [2].

Hyperperiod = WCT_{EC}.

The AEC updates then its properties according to the changes that will take place. $NumOfAARTCs$ 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 decreased by number of aperiodic RTCs that are deleted. $NumOfPARTCs$ is decreased by number of periodic RTCs that are deleted. WCT_{EC} is set to the temporary value, which is calculated in step 4 as follows:

$$WCT_{EC} = WCT_{ECtemp}.$$

The hyperperiod is updated according to step 4.

$$Hyperperiod = WCT_{EC}.$$

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. See Figure 14.

Boundedness proof:

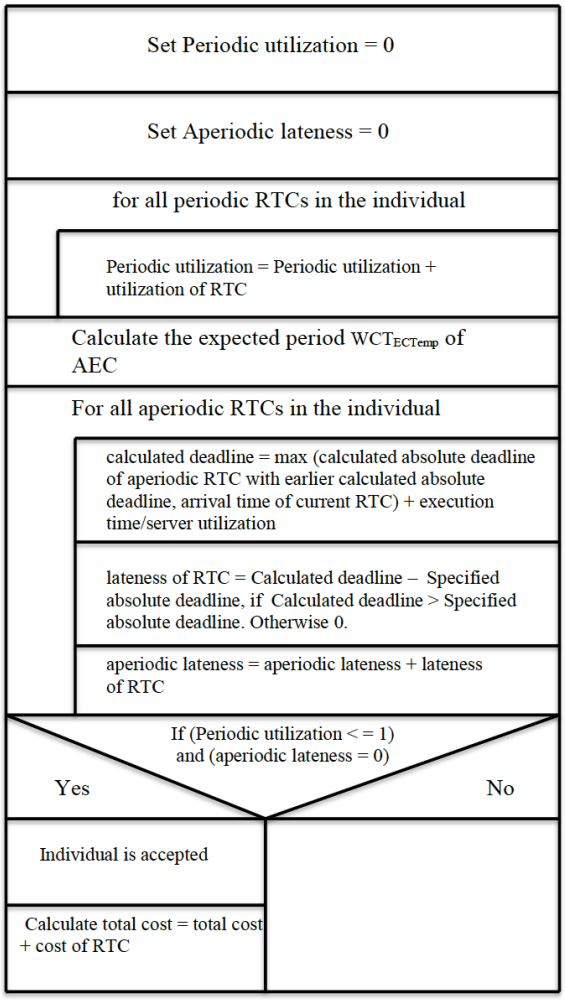

Figure 11. Nassi-Schneiderman Diagram for evaluating an individual [2].

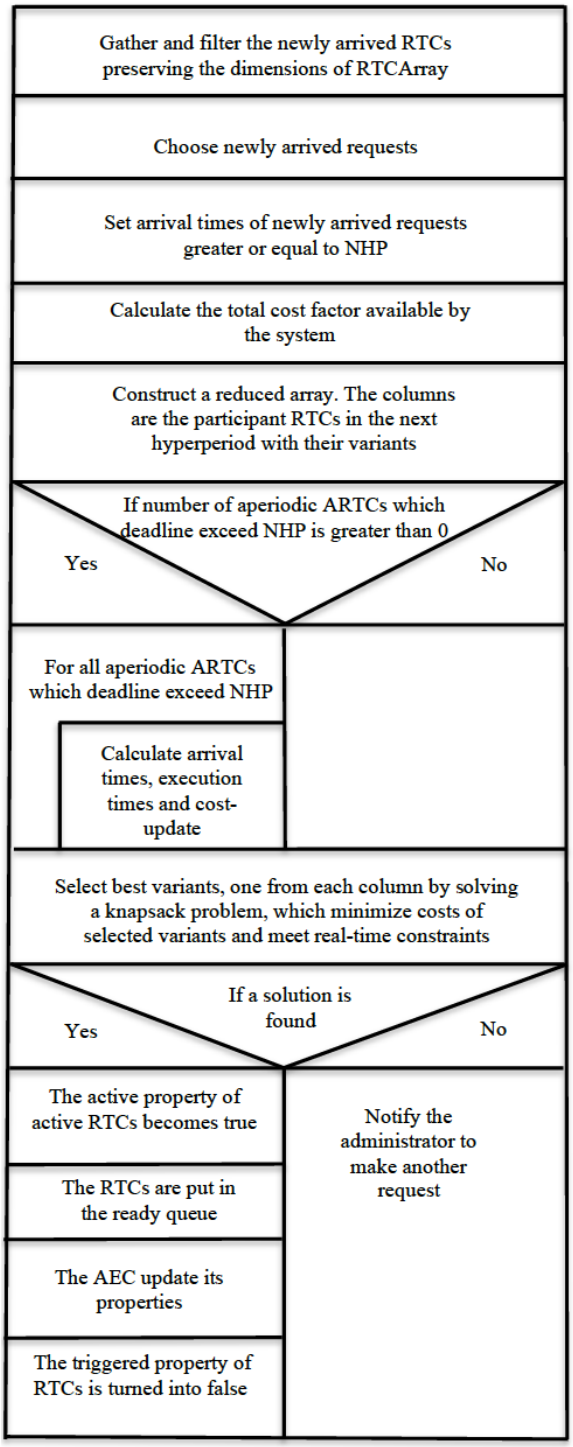

Figure 12. Nassi-Schneiderman Diagram for evaluating a generation [2].

- Activating each accepted request is done in constant time by turning the $Active$ property into true.

- Iterating over newly arrived requests is bounded by QB.

- Updating each of the AEC properties ($NumOfAARTCs$, $NumOfPARTCs$, period of the EC) is also done in constant time.

Step 6: Turning the triggered requests into non-triggered:

The Triggered Property of requests RTCs is turned into false. If the arrived requests are accepted, $WorkingRTCArray$ is copied to $RTCArray$, and then it is set to empty.

Boundedness proof:

- The $Triggered$ property of each request RTC is turned to false in constant time.

- Iterating over the requests in $WorkingRTCArray$ is done in time bounded by h ,f and QB.

- Copying $WorkingRTCArray$ to $RTCArray$ is done in time bounded by f and h. Resetting $WorkingRTCArray$ is done in constant time.

Step 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 of requests by another set. $Cost_{total}$, WCT_{ECtemp}, The expected hyperperiod, $AdaptationRTCArray$, $ExpPARTCs$ and $ExpAARTCs$ are reset to their initial values. $WorkingRTCArray$ is set to empty.

Boundedness proof:

- A notification is sent to the system administrator in constant time. $Cost_{total}$, WCT_{ECtemp}.

- The expected hyperperiod, $AdaptationRTCArray$,

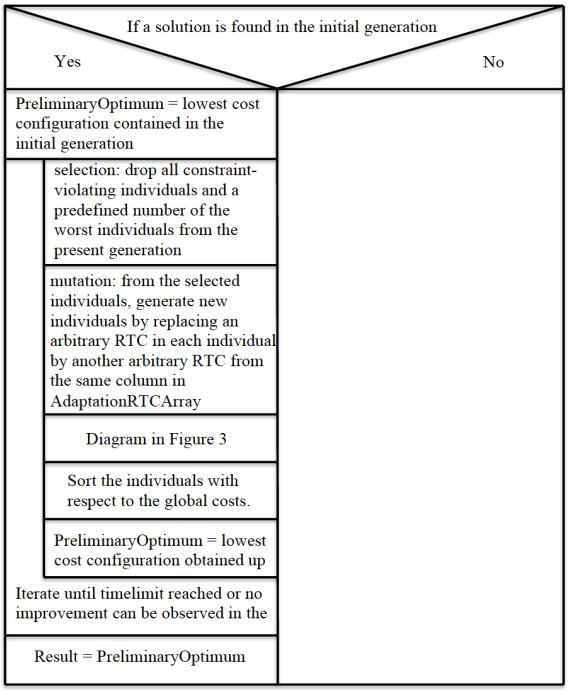

Figure 13. Nassi-Schneiderman Diagram for the genetic algorithm [2].

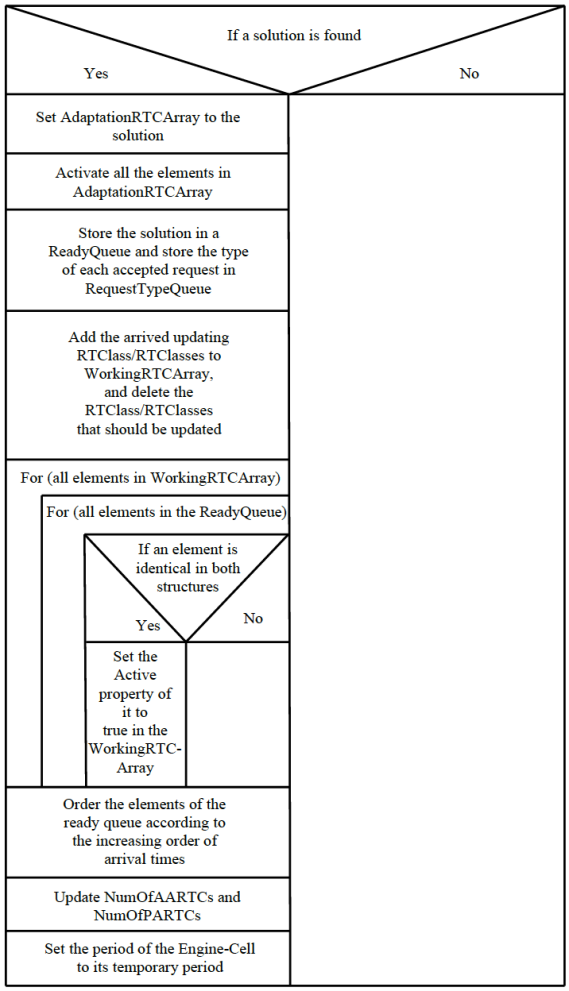

Figure 14. Nassi-Schneiderman Diagram for activating the accepted requests and updating the AEC [2].

```
For i = 1 to N do{      Time consumed in Step 1 is:  Const1 × N
Set x = i;              Time consumed in Step 2  is:  Const2 × N
Set y = 6;              Time consumed in Step 3  is:  Const3 × N
Set z = x + y;          Time consumed in Step 4 is:  Const4 × N
}

Time consumed by the Pseudo code is equal to  Const1 × N + Const2 × N +  Const3 × N + Const4 × N = (Const1 + Const2 + Const3 + Const4) × N.
The time complexity is O(N).
```

Figure 15. Example from extracting time complexity from a pseudo code.

$ExpPARTCs$ and $ExpAARTCs$ are reset in constant time.

- Resetting $WorkingRTCArray$ is done in constant time.

As shown above, the adaptation process takes place in bounded time. Part of this process, however, is the calculation of the $WCET_{EC}$ of the EC. This value depends on a couple of parameters that may vary by each application of the adaption. The following complexity function has been derived in [2] to express the influence of all relevant parameters on time complexity. It is assumed that based on this complexity function and an adequate model of the underlying hardware the resulting $WCET_{EC}$ can be estimated with sufficient precision. The reason is that the complexities, which construct the function can be derived from a pseudo code of the adaptation algorithm. [24] points out how time complexity can be derived from a pseudo code. In [2], the pseudo code was not described, but an estimation was done on Nassi-Schneiderman diagrams. Each diagram points out the step, which is specified by that diagram, by breaking it into smaller steps. One can estimate the complexity of these small steps, as if they were reflecting parts of a pseudo code. In Figure 15, we present an example for extracting the time complexity from a pseudo code.

We find that the algorithm is solved by a quadratic time complexity.

Complexity of the algorithm [2]: 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 = $O(b*h*f)+O(QB*h*f*SC+SC^2)+O(PN)+O(QB*SC*f*h+h^2*f+QB*n+h*n+f^2*h+f*m1+f*logGRP)+O(QB*SC*h*f+h^2*f)+O(h*f*QB*SC)+O(1) = O(b*h*f+PN+f*m1+f*logGRP+f^2*h+h^2*f+QB*n+h*n+h*f*QB*SC+SC^2)$

In Section V, we have first pointed out how to transform the traditional RT tasks into RT cells. For this purpose, we defined the new properties that have to be added to the structure of RT tasks in order to allow executing the tasks as cells. Cells can change their structure an behavior at runtime. In our approach, there is two kinds of cells. EC belongs to one kind, and RTCs belong to the other kind. In this section, we have listed the properties of EC and RTCs. Then we listed all parameters, which may play a role in time complexity of the adaptation algorithm. We defined the parameters. Afterwards, we went through the steps of the algorithm. In each step, we explained how the step is performed. Then, we presented the boundedness proof of the step. Finally, we presented the time complexity of the algorithm.

In the next section, we summarize the content of the paper, and introduce potential future work.

VI. Conclusion and Future Work

In this paper, we provided the details of the algorithmic solution described in [1]. We have showed the proof of boundedness for each step in the algorithm. The solution tries to evolve the system at run time. Each time the EC executes, and new requests exist, there is a possibility to change the RTCs, which construct the system. The EC executes a genetic algorithm, to solve a knapsack problem. The conditions of the problem aim to provide more processor capacity and to minimize the costs by choosing best combination of cells variants. Every individual that results from the genetic algorithm acts as a possible input for the knapsack. The genetic algorithm runs until a best individual is found, or a predefined time limit is reached. The time complexity of the algorithm has been deduced depending on abstract code. Code statements have been modelled by Nassi-Schneiderman diagrams [3]. In [2], time complexities are listed in the diagrams. In the future, we may apply different approaches including different genetic algorithms to solve the knapsack problem. This may provide different optimization output [2]. The problem might also be modelled by means different from the knapsack. Considering communication between cells is an additional aspect that may expand the scope of RT applications, where the algorithm can be applied [2]. The solution is designed for a local node and one remote node, where newly deployed cells can be installed. Later we may design a solution for more than one remote node. Each one is dedicated for a different type or sort of RT cells. By applying this enhancement, we can save costs because nodes can stay where appropriate developers exist [2].

VII. Acknowledgement

This work is based on a PhD thesis done at University of Paderborn, Germany [2].

References

[1] L. Khaluf and F. Rammig, “Organic Self-Adaptable Real-Time Applications,” In the fifteenth international conference on Autonomic and Autonomous Systems (ICAS), pp. 65-71, 2019.

[2] L. Khaluf, “Organic Programming of Dynamic Real-Time Applications,” a PhD thesis, University of Paderborn, 2019.

[3] I. Nassi and B. Schneiderman, “Flowchart Techniques for Structured Programming,” Technical Contributions, Sigplan Notices, pp. 12-26, 1973.

[4] L. Khaluf and F. Rammig, “Organic Programming of Real-Time Operating Systems,” In the ninth international conference on Autonomic and Autonomous Systems (ICAS), pp. 57-60, 2013.

[5] H. Ghetto, M. Silly, and T. Bouchentouf, “Dynamic scheduling of real-time tasks under precedence constraints,” Journal of Real-Time Systems, 2, pp. 181-194, 1990.

[6] M. Spuri and G. C. Buttazzo, “Efficient Aperiodic Service under Earliest Deadline Scheduling,” Real-Time Systems Symposium, pp. 2-11, 1994.

[7] S. Htiouech, S. Bouamama and R. Attia, “OSC: solving the multidimensional multi-choice knapsack problem with tight strategic Oscillation using Surrogate Constraints,” International Journal of Computer Applications (0975 8887), Volume 73 - No. pxc3889883, 2013.

[8] M. M. Akbar, M. S. Rahman, M. Kaykobad, E.G. Manning and G.C. Shoja, “Solving the Multidimensional Multiple-choice Knapsack Problem by constructing convex hulls,” Journal Computers and Operations Research archive, pp. 1259-1273, 2006.

[9] M. Hifi, M. Michrafy and A. Sbihi, “Algorithms for the Multiple-Choice Multi-Dimensional Knapsack Problem,” In: Les Cahiers de la M.S.E : série bleue, Vol. 31, 2003.

[10] Y. Chen and J-K Hoa, “A ”reduce and solve” approach for the multiple-choice multidimensional knapsack problem,” European Journal of Operational Research, pp. 313-322, 2014.

[11] A. Sbihi, M. Mustapha and M. Hifi, “A Reactive Local Search-Based Algorithm for the Multiple-Choice Multi-Dimensional Knapsack Problem,” Computational Optimization and Applications, pp. 271–285, 2006.

[12] Shubhashis K. Shil, A. B. M. Sarowar Sattar, Md. Waselul Haque Sadid, A. B. M. Nasiruzzaman and Md. ShamimAnower, "Solving Multidimensional Multiple Choice Knapsack Problem By Genetic Algorithm & Measuring Its Performance," International Conference on Electronics, Computer and Communication (ICECC), 2008.

[13] A. Duenas, C. D. Martinelly, and G. Tütüncü, "A Multidimensional Multiple-Choice Knapsack Model for Resource Allocation in a Construction Equipment Manufacturer Setting Using an Evolutionary Algorithm," APMS (1), IFIP AICT, Volume 438, pp. 539-546, 2014.

[14] IBM ILOG CPLEX Callable Library version 12.6.2.

[15] D. E. Goldberg, "Genetic Algorithms in Search, Optimization, and Machine Learning," 1989.

[16] G. C. Buttazzo, "Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and Applications," Third Edition, Springer, 2011.

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

[18] D. Pisinger, "Algorithms for knapsack problems," Dept of Computer Science, University of Kopenhagen, PhD thesis, February, 1995.

[19] F. Streichert, "Introduction to Evolutionary Algorithms," University of Tuebingen, 2002.

[20] http://math.feld.cvut.cz/mt/txtb/3/txe3ba3c.htm. Last visited on 30.05.2020.

[21] https://en.wikipedia.org/wiki/Monotonic˙function. Last visited on 30.05.2020.

[22] http://www.nlreg.com/asymptot.htm. Last visited on 30.05.2020.

[23] J. E. Speich and J. Rosen, "Medical Robotics, Encyclopedia of Biomaterials and Biomedical Engineering," 2004.

[24] https://en.wikipedia.org/wiki/Analysis˙of˙algorithms. Last visited on 31.05.2020.

[25] T. Weise, "Global Optimization Algorithms - Theory and Application", Self-Published, second edition, 2009.