# Organic Self-Adaptable Real-Time Applications

Lial Khaluf
Email: lial.khaluf@googlemail.com

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

***Abstract*—Nowadays, computing systems tend to find inspiration for their behavior in organic systems. Approaches have been published to develop a system behavior with the potential to react to environments. In the real-time domain, such approaches are still very rare and limited. In this paper, we provide an approach which is able to adapt at runtime and, at the same time, preserve all real-time constraints. In accordance to "Organic Programming", we make use of the concept of cells. A cell is an extension of a task allowing its adaptation. Cells exist by means of classes, which consist of a limited set of cell variants. All variants of a cell share the same fundamental functionality, however under different computing time demands and different costs. Our approach consists of an adaptation algorithm that behaves as a real-time cell. Under the assumption that the ecosystem of the real-time environment is given in the form of a set of real-time cells, each one with multiple variants, it provides a selection mechanism in the space of this ecosystem. The system goals aim to reduce system costs under the constraint of meeting all real-time requirements.**

***Keywords–Real-time cell; variant; organic programming; optimization; self-adaptability.***

## I. Introduction

Turning any physical process into an online process is a current trend in many kinds of businesses. This evolution is reflected by transforming the current physical systems into Cyber Physical Systems. In such systems, the correct functionality of the system is influenced by its reaction to internal and external events. Such adaptation capabilities apply in general to control processes as, for example, in the medical or energy sectors, etc. In most cases, the nature of such processes belongs to embedded systems where timing constraints have to be achieved. Cyber Physical Systems add several advantages over traditional systems, such as self-adaptability as reaction to failures as well as unexpected conditions [1]. In this sense, such a system is evaluated by its ability to adapt itself to environmental changes in real-time. Many approaches have been proposed to solve this challenge. However, most of the existing approaches have several limitations related to the ability of reacting to unexpected events, or reacting in an undefined way. In order to overcome these deficiencies, we introduce in this paper 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 that the system has the ability to grow at runtime. In other words, it is assumed that the system is able to have new resources, new events and reactions at runtime. Currently, we apply our algorithm on a single node system, with the ability to import the needed information from the outside which can be considered as a remote node. This information consists of the different reactions that the system may apply in response to specific events that may result from an internal or external environmental change. The reactions are developed by external sources, and added to the system at runtime. The solution we provide applies for all kinds of real-time systems. This is done by providing the system with organic properties at the level of real-time tasks. Such tasks, in our case, are transformed into cells, called real-time cells. A real-time cell is an extension of a real-time task by mechanisms empowering it to self-adaptation. Whenever an adaptation takes place, both the adaptation and the resulting adapted system have to respect real-time restrictions. For this purpose, a selection process is part of the adaption mechanism. Its search space is restricted to a current ecosystem given by a limited number of cell classes, each one with a limited number of variants. Under the constraint that all real-time restrictions have to be satisfied, this selection process aims to minimize the overall system costs. We assume a relatively low frequency of adaptation requests. Such requests react to requested improvements or slight environmental changes. In this paper, we mostly concentrate on the central essential question: how adaptation requests can be handled under real-time constraints. In Section 2, we present the related work. Section 3 describes the problem we are facing and provides a solution for it. In the last section, we conclude the achievements of the paper and present possible future work.

## II. Related Work

In [2], a new model for organic programming is introduced. It aims to overcome limitations of the traditional programming models such as the Object Oriented Programming (OOP) [28], Model Driven Architecture (MDA) [27] or Aspect Oriented Programming (AOP) [26], where abstract classes or models are difficult to change. The idea behind the approach in [2] is to have a system that is able to grow and evolve continuously. However, it was not made for real-time systems. In our approach, we concentrate on having a system consisting of cells with defined properties that enable self-adaptability in real-time.

The approach in [3] and [4] defines different profiles with different resource requirements for each task. It enables choosing the best combination of profiles at runtime to adapt the system to certain situations. However, these profiles are developed offline, and new ones cannot be added to the system at runtime, which decreases the system adaptation ability. Our approach applies the concept of organic programming by giving the ability to modify tasks online in a way that preserves

all real-time constraints. Cells can be developed and added online to the system.

In [5], 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: (1) Goals: Evolution, Flexibility, Duration, Multiplicity, Dependency, Change, Source, Type, Frequency, Anticipation, (2) Mechanisms: Type, Autonomy, Organization, Scope, Duration, Timeliness, Triggering, and (3) Effects: Criticality, Predictability, Overhead, Resilience [5]. In our approach, system goals may change according to adaptation scenarios. Events that trigger an adaptation depend on the system where we apply the developed algorithm. The type of change that causes an adaptation could be functional, non functional, or technological. In our approach, there is no restriction on this issue; changes are foreseeable, but can change over time.

Mechanisms of adaptability summarize how the system can react to changes, in terms of space and time required. The algorithm we provide may act by decisions taken automatically or by other parties. The adaptation is done by a central component. The scope of adaptation could be local or global. The duration of the adaptation is influenced by execution time of the central component.

The set of dimensions and effects deals with results of adaptation, such as the overhead. In our approach, missing a deadline may confirm the failure of the system.

In [6], a second roadmap for state of the art is presented. Challenges of a self-adaptive system are described.

The first challenge is to understand the different alternatives that may represent designer or developer decisions. In our approach, we have developed a general strategy that applies for different kinds of real-time systems. We have an abstract implementing component, which fits as a reusable component.

The second challenge is concerned with understanding the nature, goals, and lifecycle of the system. In [6], a comparison between the basics for traditional software processes, and self-adaptive processes is described. The first one is illustrated in [7] by the traditional approach to corrective maintenance, and the second in [8] and [9] by the automatic workaround approach. The traditional approach reports the problem to the developers. The automatic workaround approach moves the corrective actions to runtime by applying alternative procedures when a failure happens. In our approach, the alternative procedure might be a new request or an update request. Analyzing causes of the failure may be assigned to a human or a subsystem. In the workaround approach, recovering methods are developed at design phase. In our approach, this can be done at runtime. In the workaround approach, if a recovering method does not exist, a report is sent to the developers, which is the same action taken in our approach.

The third challenge is concerned with decentralization of control loops. Controlling a system could be done in a centralized [10]-[12] or decentralized manner [13]-[17]. The self-adapting component is central in our approach, as network reliability in terms of time and trustworthy is a main concern in real-time systems.

The fourth challenge is the verification and validation of the system. In our approach, verification is done for requirements of real-time systems, apart from the context of the system.

In our approach, we define the optimization constraints in a multi-dimensional multiple choice knapsack problem. Most common solutions can be found in [18] and [19]. In our approach, we use a genetic algorithm inspired from [18] to solve a knapsack problem. The reason is that it can provide the whole solution (individual consisting of best variants in terms of time and cost) at once if available. This allows to use required parameters of the individual elements in order to calculate the parameters of other elements. The most important fact for our application is 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.

## III. Problem Description and Solution Concept

In our assumption, we consider periodic, aperiodic tasks or if both then evidently together. Dependability may exist between aperiodic tasks. A request can be adding a task, deleting a task, updating a task, adding a set of dependent tasks, deleting a set of dependent tasks, updating a set of dependent tasks. We assume a mixed hard-deadline periodic and aperiodic task environment. Figure 1 shows an example of request types. In case 1, the algorithm should solve the case of Task 5 not being accepted by the underlying schedulability algorithm. In case 2, the algorithm should solve the case of Task 1 update not being accepted by the underlying schedulability algorithm, and the question of how to make an update of Task 1.

In this paper, we only consider the activities on one single local node. System tasks, and tasks that are triggered have to be executed on this node. We assume that task management is carried out by a Real-Time Operating System (RTOS) with Earliest Deadline First algorithm (EDF) as the principal scheduling method. Furthermore, we assume that aperiodic tasks are handled via a Total Bandwidth Server (TBS) [20] and that the underlying RTOS runs the Stack Resource Protocol [21] to avoid unlimited blocking and deadlocks. In order to be able to run the adaptation algorithm, we come up with the concept of real-time cells. A cell is a task that is able to change its structure and behavior at runtime, to allow adaptations in real-time. The change is decided by a central cell called “Engine-Cell”. Assuming that the system before update is correctly functioning, we strictly follow the concept of transactions. If a solution for an update request is found after applying the update operations, the system state is updated. If a solution is not found, the system goes back to the previous state.

The above mentioned Engine-Cell runs the adaptation algorithm using a two dimensional array as model of the underlying ecosystem. Each column stands for a class of cells which all share the same principal functionality. Each cell in the column is a variant, where these variants accomplish the same task, but with different costs and execution time demands. The Engine-Cell runs an adaptation algorithm, which intends to select over all cell classes the best combination of variants that allows to accept the newly arrived requests. The objective is to fulfill all real-time constraints and to provide a globally maximum quality of the adapted system. As this selection process takes place on the ecosystem defined by the mentioned two-dimensional array, the search space is restricted

to the bounded set of cells with bounded number of variants which is present at time of adaptation.

We also assume a remote node (as model of the environment) that is dedicated to install variant updates, and newly deployed cells. An update request means updating a cell according to a provided change of parameters without altering the principal behaviour. The remote node is used for providing external storage, and also to be uploaded in an appropriate place for developers. Modelling the current state of the system and cell classes and viewing these models by developers are not discussed in the scope of the paper. At each execution of the Engine-Cell, new requests may have arrived to the system. The adaptation algorithm is run by the Engine-Cell, trying to find a feasible solution by selecting variants over all cell types. A real-time cell becomes active when it is accepted by the system for execution. The Engine-Cell is called an Active Engine-Cell (AEC) once it is activated. Any other Real-Time Cell (RTC) is called an Active Real-Time Cell (ARTC) once it is accepted for execution. The Engine-Cell is treated here as a periodic cell and stays active as long as the system is running. We make the general assumption for all periodic cells (including AEC) that the relative deadline is equal to the period. As investigating the acceptance of newly arrived requests is part of the Engine-Cell algorithm, we ensure that the system state does not change during the execution of the Engine-Cell.

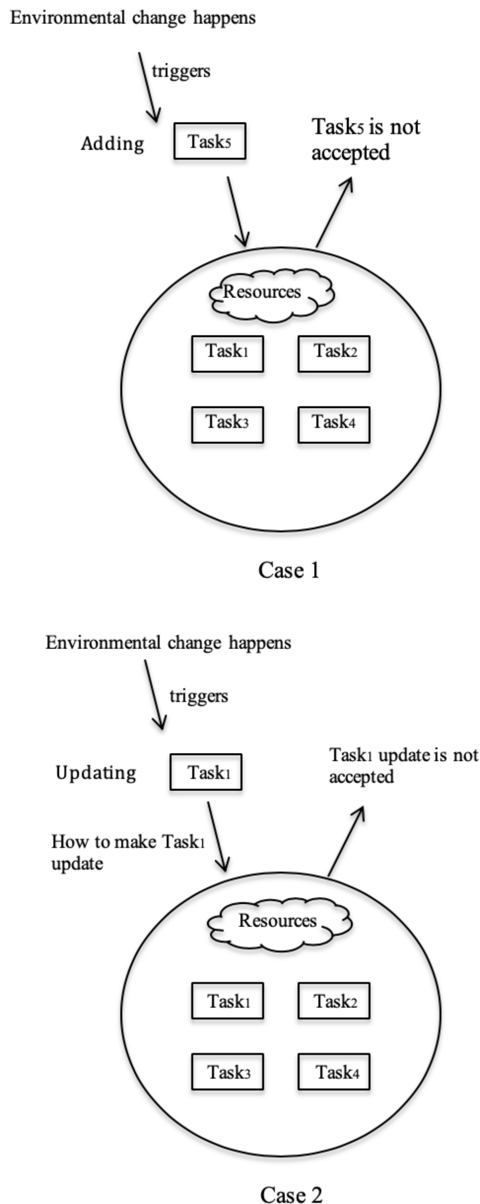


Figure 1. Request Types

The parameters controlling the Engine-Cell are defined as follows:

1) Hyperperiod: is the hyperperiod of the currently accepted periodic ARTCs. The next point in time where a hyperperiod completes execution is abbreviated as NHP (Next Hyperperiod). Adaptation takes place only once per hyperperiod. It becomes effective not earlier than 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. Calculating an initial NHP is carried out either offline or as part of the initialization when starting the system.
   Note: the response time on adaptation requests depends on the load of the system. A highly loaded system means a smaller fraction of the processing capacity to be dedicated for the AEC. At the same time, the execution time demand of the AEC tends to increase if some fixed upper bounds (such as dimensions of the RTCArray) change.
2) NumOfPARTCs: is the number of the current periodic ARTCs in the system.
3) NumOfAARTCs: is the number of the current aperiodic ARTCs in the system.
4) RTCArray: is the data structure that holds the different variants of RTCs in the system. Figure 2 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 execution time requirements. Switching between the different variants online enables to execute tasks in the best way regarding system resources. All periodic variants, that belong to the same class, have the same period. All aperiodic variants that belong to the same class, have the same deadline. The RTCArray is a dynamic component. RTCs can be added to it online. The upper bounds of its dimensions can grow online. Other parameters include the Worst-Case Execution

Time ($WCET_{EC}$), the worst case period ($WCT_{EC}$), and additional properties of the EC.

Another set of properties is defined for ordinary RTCs:

1) VariantsAllowed: is a Boolean property. When it is equal to true, all variants that belong to the class of the respective RTC variant should be examined to select the best variant in the adaptation algorithm. Otherwise, the respective RTC variant is considered mandatory to be processed by the algorithm.
2) 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 within the same class from the RTCArray. 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. UpdatingPoints is of relevance only in case of aperiodic tasks. Instances of such tasks may have a long execution time, exceeding the current hyperperiod. Therefore, just waiting for the next instance would not be appropriate. In case of periodic tasks, we restrict updates on the natural updating point, defined as the release time of the next instance of a periodic task [22].
3) $ET_{executed}$ : is the time that has been spent in executing an aperiodic RTC before the previous NHP.
4) NextUpdatingPoint: a variable that saves the next updating point which has not been yet reached by the executed code of the RTC.
5) 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.
6) TriggeringTime: is the time at which an RTC is triggered (chosen from the RTCArray).
7) TriggeringRange: is the range of time within which the arrival time of an RTC could be set. Our goal is to set the arrival time of requests greater or equal to NHP, because at this point, we assume that all accepted periodic requests are simultaneously activated (i.e., we assume all phases to be 0).
8) Deletion: a Boolean property, that is set to true if the request means deletion of a cell. It is set to false, otherwise.
9) Active: is a Boolean variable that is set to true when the cell is accepted for execution.

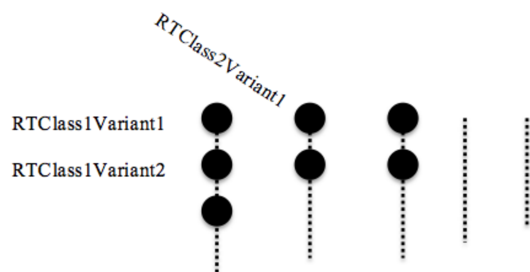


Figure 2. RTCArray

Other properties not described in the scope of this paper include the ID of the RTC (RTClassID/VariantID) inside RTCArray, the cost of an RTC, the importance factor, the factor of essentiality, the static parameters, and the updated cost, which should be calculated for an RTC, when it replaces another executing RTC.

In the following:

- We use the term ExpPARTCs to refer to the set of current periodic ARTCs excluding the RTCs, which belong to the deletion requests.

- We use the term ExpAARTCs to refer to the set of current aperiodic ARTCs excluding the RTCs, which belong to the deletion requests.

The Engine-Cell algorithm can be sketched as follows (See Figure 3):

**Step 1: Gathering and filtering the newly deployed RTCs:** The first step of the AEC is to collect the newly deployed RTCs, and store them in a WorkingRTCArray (a copy of RTCArray) following a procedure that ensures to keep the upper bound of the WorkingRTCArray dimensions preserved. As newly deployed RTCs enlarge the solution space RTC classes and/or variants may need to be dropped following some importance criteria.

**Step 2: Triggering and handling the newly arrived requests:** In this step, a TriggeredQueue is constructed from the WorkingRTCArray. Triggering a request from the WorkingRTCArray turns the Triggered property into true. Arrival times of requests are set greater or equal to NHP according to their TriggeringRange. The DeletionTime of requests that have to be deleted is set to the next updating point. If a request includes a set of dependent cells, we assume that their modified arrival times and deadlines are calculated offline following the rules of Modified Earliest Deadline First algorithm EDF* [23].

**Step 3: Calculating the cost of quality factors for the system:** The total cost of factors available by a node $Cost_{total}$ is calculated.

**Step 4: Adaptation algorithm:** In this step, we construct the lowest-cost feasible solution over the entire set of RTClasses stored in an AdaptationRTCArray which is constructed in the beginning of this step. This data structure further reduces the search space to be considered by excluding deleted cells and aperiodic cells that have their absolute deadline within the current hyperperiod. The reason for the latter exclusion is following the general assumption that adaptations become active not earlier than in the next hyperperiod.

To construct AdaptationRTCArray, we first copy variants of WorkingRTCArray into AdaptationRTCArray. We then reduce AdaptationRTCArray to contain only all classes of ExpPARTCs and such ExpAARTCs with absolute deadlines exceeding NHP. For each aperiodic ARTC that should be deleted and has an absolute deadline exceeding NHP, we add a column including the ARTC as the only variant. After that, we add a column that includes the AEC. We add the newly triggered requests, and finally the updating requests:

- 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. The precise algorithm to identify the set of aperiodic

ARTCs that can be updated and to calculate the time characteristics for the updates are omitted here. The number of those aperiodic ARTCs is denoted by NumOfANHP. The updated variant has been excluded when constructing ExpAARTCs.

- Adding a periodic update is done by adding the arrived RTClass, which includes the triggered updating variant to AdaptationRTCArray. The updated variant has been excluded when constructing ExpPARTCs.
- In case there is an update request for a set of aperiodic dependent RTCs the same rules as of updating a single (independent) variant are applied.

By the above operations, a reduced array is constructed that contains only those entries which are relevant for the adaptation algorithm. For technical reasons, the columns in the array are reordered, so that periodic columns come first, then AEC, and finally aperiodic columns.

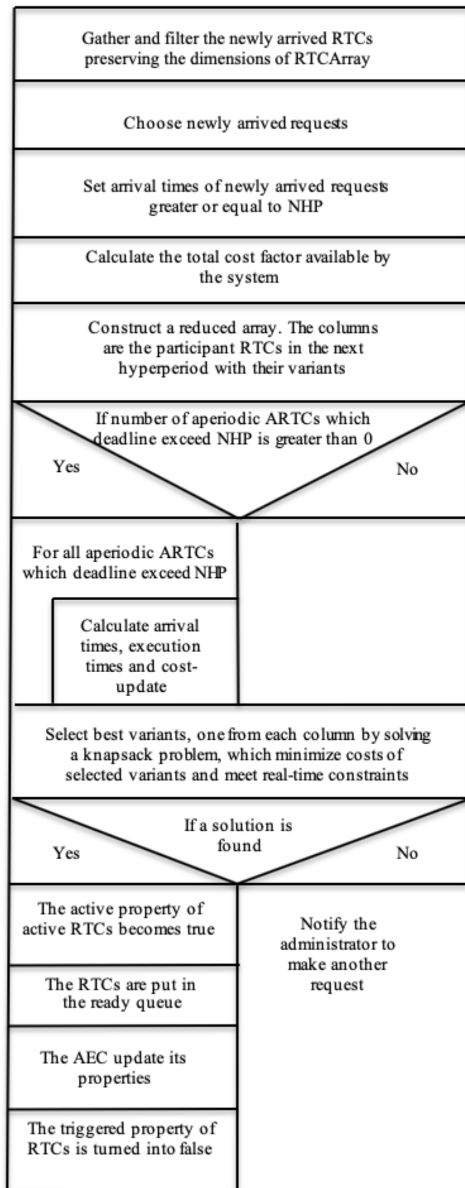


Figure 3. Nassi-Schneidermann Diagram for EC Algorithm

Let us assume that the number of columns in AdaptationRTCArray = Num. $\grave{N}$ denotes the number of columns, which represent the newly triggered aperiodic requests.

If (NumOfANHP $>$ 0) then we calculate arrival times, execution times, and Cost-Update for the running aperiodic ARTCs that are stored in AdaptationRTCArray, and deadlines exceed the NHP. The details of these calculations are omitted here.

The heart of the adaptation algorithm is to find a selection of variants for all RTC classes in the relevant ecosystem. This relevant ecosystem has been determined by the activities described above and stored in AdaptationRTCArray. The solution is a one dimensional array Solution that is assumed to contain one variant from each column in the AdaptationRTCArray. The chosen variants should pass the schedulability test of the Total Bandwidth Server (TBS) [20], and achieve the lowest possible accumulated cost.

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

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

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

Where:

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

By Num is denoted the number of columns (RTC classes) in AdaptationRTCArray while by $n_i$ is denoted the number of variants in the $i_{th}$ column. Note that three constraints are formulated for the three values of parameter k. Constraint 1 handles periodic tasks including the AEC, constraint 2 the aperiodic ones, and constraint 3 is an optional one limiting the total cost. For these three constraints, the weights $W^k$ and the constraining condition $R^k$ are defined differently.

Constraint 1: $W_{ij}^1 = Factor_1/Factor_2$

For any of the periodic RTCs: $Factor1 = C_{ij}$, $Factor2 = T_{ij}$

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

$WCT_{ECTemp}$ denotes the expected hyperperiod of the AEC. It is calculated the same way the initial hyperperiod is calculated. Here periodic cells are ExpPARTCs in AdaptationRTCArray, and newly triggered periodic requests in AdaptationRTCArray. Expected period of AEC is used instead of its current period. 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: Factor1 = 0, Factor2 = 1

Constraint 2: $W_{ij}^2 = Factor1 - Factor2$

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

For any of the aperiodic RTCs: $Factor1 = d_{Specified,ij}$, $Factor2 = d_{Calculated,ij}$.

Where:

Copyright (c) IARIA, 2019. ISBN: 978-1-61208-712-2

$d_{Specified,ij}$ : 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.

By $d_{Calculated,ij}$ we denote the deadline calculated by the TBS rule.

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

$Us = 1 - Up$.

Constraint 3: Depending on the different kinds of RTCs to be considered in solving the knapsack problem, the weights $W_{ij}$ for the optional third constraint are defined as follows:

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

$W_{ij}^3$ = Cost for aperiodic RTCs that are stored in AdaptationRTCArray

After defining the weights of the different variants, we can start discussing the conditions. The constraining conditions $R^k$ for the three constraints are defined as follows:

$R^1$ = 1(EDF constraint for periodic cells). $R^2$ = 0 (no aperiodic task missing its deadline). $R^3$ = $Cost_{total}$.

The limit $Cost_{total}$ is optional. If a solution is found, the newly arrived requests are accepted.

The algorithm which 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. In total, there exist up to $f^h$ individuals. Each of them is a potential solution of the knapsack problem. We select smaller subsets of individuals and call them Generations. Let us assume that the number of individuals in a generation $\leq$ upper bound of number of RTCs in a class in the WorkingRTCArray. In the initial generation, the first individual is given by selecting from each RTClass the variant with the lowest respective utilization. This individual allows a simple decision whether a solution exists, as if this individual does not fulfill the constraints then there cannot exist any solution. If the knapsack constraint $\sum_{i=1}^{Num} \sum_{j=1}^{n_i} W_{ij}^k x_{ij} \leq R^k$ has a solution for a set of individuals, we choose the individual which minimizes the accumulated cost of the chosen RTCs. The lowest-cost individual of a generation is a preliminary solution of the knapsack problem. The previous operations are bounded by upper bounds of RTCArray dimensions, and the given time bound for the iteration. 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. The latter termination condition guarantees boundedness.

**Step 5: Activate the accepted requests, and update the AEC:** The Active property of accepted RTCs becomes true. They are put into the ready queue as managed by the underlying RTOS. The AEC updates its properties. Updating requests take place in the WorkingRTCArray. After that, AdaptationRTCArray is set to empty. Cells are still enforced when having them replaced by other variants because, by definition, updating points are designed for this reason. Values of still to be used variables are transmitted to the updating variants, and accomplishing the same functionality must be ensured by the developer.

**Step 6: Turning the triggered requests into non-triggered:** The Triggered Property of requests RTCs is turned into false. After that, WorkingRTCArray is copied to RTCArray if the solution is accepted, and then it is set to empty.

**Step 7: Notify the system, in case the requests are not accepted. :** Algorithm variables are reset to their initial values.

In [24] we modelled each of the previous steps by a Nassi-Schneidermann diagram [25]. This helps to understand the specification of code structure and points out the calculation of time complexity.

Concerning the time complexity of the developed adaptation algorithm, we can show that per single execution (i.e., once per hyperperiod) the algorithm can be solved in quadratic time in the upper bounds of dimensions of RTCArray and upper bound of number of RTCs inside a dependent set request [24]. Parameters of time complexity are bounded. In case of solving the knapsack problem, this boundedness is enforced by setting an upper bound of execution time in the iterative genetic algorithm. Together with the fact that there are no unbounded blockings possible due to parameters not under control of the algorithm (assumption of Stack Resource Protocol included in the underlying RTOS) this implies the boundedness of the algorithm.

## IV. Conclusion

In this paper, we have developed an approach that enhances real-time operating systems by an organic adaptability feature. This implies building an infrastructure of the system, which can change its behavior at runtime. The basic unit in this infrastructure is a cell. A cell is a task that can change its structure and behavior by selecting a variant of it at runtime. The way variants are chosen at runtime follows resource and time limitations, in order to enhance the quality of the system. The boundedness of our algorithm has been proven. Many new trends can be developed in the context of the described problem, such as distributing the central algorithm that is run by the Engine-Cell on several nodes in order to save more processor utilization on one node, obtaining fault tolerance, dealing with the boundedness of the algorithm in case of a non- deterministic network, such as in a multi-agent system, measuring the optimization output by running the algorithm on a real-time operating system and observing the results, and having several controlling cells other than the Engine-Cell or having several variants of it, etc.

## V. ACKNOWLEDGEMENT

This work is based on a PhD thesis done at University of Paderborn, Germany [24].

## References


[1] E. A. Lee, "Cyber Physical Systems: Design Challenges," 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC), pp. 363-369, 2008.

[2] O. Imbusch, F. Langhammer, and G. von Walter, "Ercatons and Organic Programming: Say Good-Bye to Planned Economy," Dagstuhl Seminar Proceedings 2006.

[3] S. Oberthür, L. Zaremba, and H. Simon Lichte, "Flexible Resource Management for Self-X Systems: An Evaluation," in Proceedings of ISORCW2010.30, pp. 1-10, 2010.

[4] S. Oberthür, "Towards an RTOS for Self-Optimizing Mechatronic Systems, Dissertation," Paderborn, Germany, October 30, 2009.

[5] H. C. C. Betty at al. (Eds.), “Self-Adaptive Systems,” LNCS 5525, pp. 1-26, Springer Verlag, Berlin Heidelberg, 2009.

[6] R. de Lemos et al. (Eds.), “Self-Adaptive Systems,” LNCS 7475, pp. 1-32, Springer Verlag, Berlin Heidelberg, 2013.

[7] E. Burton 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.

[8] A. Carzaniga, A. Gorla, N. Perino, and 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.

[9] A. Carzaniga, A. Gorla, and 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.

[10] D. Garlan, S.W. Cheng, A.C. Huang, B. Schmerl, and P. Steenkiste, “Rainbow: Architecture-based self-adaptation with reusable infrastructure,” IEEE Computer 37, pp. 46-54, 2004.

[11] IBM: “An architectural blueprint for autonomic computing,” Tech. rep. IBM, January 2006.

[12] P. Oreizy et al., “An architecture- based approach to self-adaptive software,” IEEE Intelligent Systems 14, pp. 54-62, 1999.

[13] Y. Brun and 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.

[14] I. Georgiadis, J. Magee, and J. Kramer, “Self-Organizing Software Architectures for Distributed Systems,” In: 1st Workshop on Self-Healing Systems. ACM, New York, pp. 33-38, 2002.

[15] S. Malek, M. Mikic-Rakic, and 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.

[16] P. Vromant, D. Weyns, S. Malek, and J. Andersson, “On interacting Control loops in self-adaptive systems,” SEAMS 2011, Honolulu, Hawaii, pp. 202-207, 2011.

[17] D. Weyns, S. Malek, and 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.

[18] A. Duenas, C. Martinelly, and 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.

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

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

[21] T. P. Baker, “A Stack-Based Resource Allocation Policy for Realtime Processes,” In: Proceedings of the IEEE Real-Time Systems Symposium (RTSS), pp. 191-200, 1990.

[22] 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.

[23] 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.

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

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

[26] G. Kiczales et al., “Aspect Oriented Programming,” in ECOOP’97 — Object-Oriented Programming, pp. 220-242, 1997.

[27] R. Petrasch, O. Meimberg, “Model Driven Architecture,” ISBN 3-89864-343-3, 2006.

[28] T. Benaya and E. Zur, “Understanding Object Oriented Programming Concepts in an Advanced Programming Course,” in ISSEP 2008: Informatics Education - Supporting Computational Thinking pp. 161-170, 2008.