Organic Programming of Real-Time Operating Systems

Lial Khaluf[1] and Franz Josef Rammig[2]
Heinz Nixdorf Institute
University of Paderborn
Paderborn, Germany
Email[1]: klial@hni.uni-paderborn.de
Email[2]: franz@upb.de

***Abstract*—Self-adaptability and self-management have become nowadays challenging properties of real-time operating systems. Since the evolution of these systems as a response to environmental changes is restricted due to real-time constraints, it becomes more difficult for the system to adapt itself and manage its stability at run time. This paper introduces a new approach, which applies the organic programming concept to real-time operating systems. The approach enables to define a self-adaptable and self-management real-time operating system.**

Keywords-cell; task; scheduling; real-time

I. Introduction

Real-time operating systems serve a set of tasks with respect to real time constraints. However, this set may grow and change over time. To allow this evolution, the real-time operating system must be able to adapt itself to the new circumstances and to manage its data in order to preserve all real time constraints. To achieve this goal, this paper introduces a new approach, which allows the tasks in a real-time application to behave like objects do in our real world. Objects in the real world can be adapted to serve a specific goal. Also creatures can change their behavior according to a set of influential factors. Similarly, there are a lot of situations encountered by real-time applications, where a modification of structure or behavior is needed (as a result of a task arrival or update). E.g., in a rescue system, if a robot task is defined to run toward a burning building using its wheels, and at run time it is decided by the system developer to get use of the wind factor. This change aims to fasten the robot movement or save some energy. In this case, the system must be able to modify the task structure (adding the wind factor) and the task behavior (consider the wind factor in the task functionality). In addition, if this task modification results in violating real-time constraints when an acceptance test is made for the updated task together with the other system tasks, then the system must have the ability to modify the structure or behavior of the other tasks at run time to increase the time capacity of the system. The new capacity may in turn allow the adaptation of the new changes with respect to real-time constraints. This ability is called organic programming [1]. In this paper, a new approach is provided to turn a real-time operating system (RTOS) into a self-adaptable and self-managing system. Here, the Organic Reconfigurable Operating System (ORCOS) [2] is used as an example for an underlying RTOS. The approach introduces the concept of real-time cells, and defines their structure and behavior aiming at increasing the time capacity of the system for accepting new task arrivals and updates. The following section gives an overview of the related work. Section 3 describes the structure and behavior of real-time cells, and Section 4 shows the adapting algorithm of the system. Finally, a conclusion of the approach and a summary of possible future work are pointed out in Section 5.

II. Related Work

Self-adaptation and self-management are properties, which many approaches have aimed to realize, e.g., Ercatons in [1] have realized the concept of a true thing, which is able to adapt new changes at run time. However, this adaptation cannot be done under real-time constraints. In the real-time area, some systems [3][4] were developed to adapt themselves to a larger processing capacity at run time, but the used techniques may result in a large unbounded time overhead. The approach in [5], being by our knowledge one of the most recent and closest approaches to ours, has defined different profiles with different resource requirements for each task and allowed to choose the best combination of profiles at run time to adapt the system to certain situations. However, these profiles are developed offline, and new ones cannot be added to the systems at run time, which decreases the system adaptation ability. The approach in this paper applies the concept of organic programming on ORCOS to turn it into a self-adaptable and self-managing system. This is done by giving the ability to modify and develop the tasks online in a way that preserves all real-time constraints.

III. Structure and Behavior of Real-Time Cells

A real-time cell, referred to as RTC, is a system component mapped to a piece of memory to which a task is assigned. There are two kinds of RTCs, controlling RTCs and controlled RTCs. The first kind cannot change its behavior at run time and it is assigned a task before the system starts running. The second kind can change its structure and behavior at run time, and it is assigned a task online after the system starts running. At the moment, the presented approach is restricted only on periodic tasks scheduled by EDF [6]. However, the approach may be extended later to support in addition aperiodic tasks using an appropriate server like Total Bandwidth Server [7]. A task is usually defined by its data and functionality. The task in the proposed approach is similar to the usual task model but with an additional set of meta data.

 ISBN: 978-1-61208-257-8

The meta data differs between the tasks that can be assigned to controlling cells, called controlling tasks, and tasks that can be assigned to controlled cells, called controlled tasks. The meta data of both controlling and controlled tasks is used by the functionality of controlling tasks to be able to modify the controlled RTCs at run time. A controlled task is assumed to exist by mean of various versions (similar to profiles in [5]), called members in this paper to avoid confusion with the classical versioning model. All members of a task can accomplish the same basic functionality. However, the characteristics in various dimensions (memory requirements, precision, power dissipation, execution time,...) can vary. For simplicity reasons in this paper, it is assumed that all objectives other than execution time can be expressed by a single parameter, called *Cost*, so a cell is characterized by (cost, execution time). A new member may be added from the outside at any time. The adaptation process described in Section 4 is affecting controlled task members, which are currently in the system.

An RTC is active when its current task instance is executing (i.e., ready or running) and is non-active otherwise. The approach here defines only one controlling RTC, called the Engine-RTC. Its controlling task, called the Engine-Task, is responsible for accepting a new RTC or changing the structure and behavior of already existing controlled RTCs at run time. In this context, a controlled RTC is changed by changing its associated task. The new controlled task should accomplish the same basic functionality, which was expected from the old one. Whenever replacing a task member as part of the dynamic change test (see Section 4), this should also increase the time capacity of the system sufficiently, e.g., it may include functions or procedures with time characteristics different from the older ones. To allow this, other parameters usually have to be modified as well, e.g., the precision of calculations may be reduced. In other words, the system may have the ability to adapt itself to any new circumstances through choosing possible alternatives of the currently executing controlled tasks. As there might be a variety of choices that provide sufficient additional processor capacity, the goal is to find the solution, which came with the minimal costs concerning all other parameters. The meta data of each controlled task consists of the following information:

- ***ID/Member***

$$ID/Member; id \in \{1, 2, ...\}$$
$$and \quad Member \in \{0, 1, 2, ...\}$$

ID is a unique number to differentiate between the controlled tasks. Member is a number to point to a controlled task alternative. *ID/Member* is read by the Engine-RTC whenever a new controlled task arrives to realize if it is a new task (its *ID* does not exist in the system) or an update of an existing controlled task (its *ID* exists in the system, but the *Member* number does not exist for this *ID*).

- ***CriticalityDegree***

$$CriticalityDegree \in \{1, 2, ...\}$$

CriticalityDegree is a number to express how critical a controlled task is. It has the same value for all members of this task. Criticality increases whenever the *CriticalityDegree* decreases and vice versa.

- ***UpdatingPoints***

$$UpdatingPoints \in \{(i, RelativeTimePoint - i);$$
$$i \in \{0, 1, 2, ...\}\}$$

UpdatingPoints are certain points in the functionality code of the controlled task where replacing this task with another member of it is possible. Each updating point is defined with a number *i*, and the time *RelativeTimePoint-i* at which the replacement can take place. E.g., if the execution time equals 10 time units, and the controlled task has two updating points (0,0),(1,9), this means that the controlled task can be replaced before it starts executing or after 9 time units. The replacement of a controlled task at some updating point means modifying the controlled RTC to which this controlled task is assigned (switching to another task member). The release time of a next instance of a periodic task constitutes a natural updating point. In this paper, the adapting algorithm is first dedicated only for natural updating points and then refined for intermediate updating points.

- ***Location***

$$Location \in \{(x, y);$$
$$x \in \{ComputeNode_0, ComputeNode_1, ...\}$$
$$and \quad y \in \{MemoryAddress_0, MemoryAddress_1, ...\}$$

Location is defined by $ComputeNode_i$, the compute node on which the controlled task resides, and $MemoryAddress_i$, its memory address on that specific compute node. *Location* is used by the Engine-RTC to fetch the controlled task from its compute node to the compute node of ORCOS ($ComputeNode_0$).

- ***ArrivalTime*** is the time at which the current instance of the controlled task should start executing. *ArrivalTime* might not exist in the meta data of a controlled task, in which case, ORCOS is informed that this task might be needed in the future for replacing a certain controlled task for which it is an alternative.
- ***DeadlineTime*** is the period of time within which the controlled task execution should be completed.
- ***FetchingTime*** is the time required to fetch a controlled task from the compute node where it resides to the compute node where ORCOS resides.
- ***Dependency***

$$Dependency \in \{true, false\}$$

Dependency is true when the controlled task depends on the completion of other controlled tasks execution, otherwise *Dependency* is false. For simplicity, the adapting algorithm in this paper assumes an independent task set, which means that *Dependency* is false.

 ISBN: 978-1-61208-257-8

- ***RelatedTasks***

$$RelatedTasks \in \{\{(id_0/M_0, L_0, FT_0), (id_1/M_1, L_1, FT_1), ...\}; id_i/M_i \in \{ID/Member\}; L_i \in Location \quad and \quad FT_i \quad is \quad the \quad FetchingTime \quad of \quad the \quad controlled \quad taski\}$$

RelatedTasks determines the identity, location and fetching time of the controlled tasks on which the execution of the controlled task depends. *RelatedTasks* is used by the Engine-RTC to get information about these tasks, since the new controlled task is only accepted in the system if its *RelatedTasks* can also be accepted. Please note that for simplicity reasons within this paper, the set of *RelatedTasks* is assumed to be empty.

- ***ExecutionTime*** = time required to execute the controlled task + *ExecutionTime* of the *RelatedTasks* + time required to register the *UpdatingPoints* in the meta data of the Engine-RTC whenever an updating point is reached at run time.
  The previous definition assumes that all *RelatedTasks* have not started execution before the controlled task does. The *ExecutionTime* of any controlled task is bounded since all its factors are bounded. The reason is that the time required to execute any controlled task and the number of the *UpdatingPoints* for any controlled task are defined offline.

The Engine-Task is responsible for:

- registering the meta data of newly arrived controlled tasks or newly arrived members.
- making an acceptance test at new arrivals to the set of controlled tasks, the new arrival, and the Engine-task with respect to its worst case execution time. The approach here assumes the EDF scheduling algorithm, where all tasks are periodic and their period is equal to their *DeadlineTime*. Thus the acceptance test is $\sum_{i=1}^{n} C_i/T_i \leqslant 1$, where n is the number of tasks, C_i is the execution time, and T_i is the *DeadlineTime*.
- making a dynamic change test. This test is made in case the new arrival cannot be accepted to the currently existing system (a set of controlled tasks). The test verifies if changing the structure and behavior of RTCs according to certain rules (see the next section) can enable the acceptance of the new arrival.
- making a dynamic change, which means changing the structure and behavior of RTCs selected by the previous test. This includes fetching the controlled tasks, which are going to replace the controlled task of the selected RTCs from their compute nodes. In addition, it includes fetching the new controlled task or the new upgrading controlled task member, for which the dynamic change test is done.
- assigning the new controlled task or the new upgrading member (if it is accepted) to a controlled RTC.
- updating the meta data of the Engine-Task. This is important to evaluate the worst case execution time of the Engine-Task.

The meta data of the Engine-Task consists of the following:

- ***ID***=0, differentiates the Engine-Task from other tasks in the system.
- ***Location***

$$Location \in \{(ComputeNode_0, MemoryAddress_i); i \in \{0, 1, 2, ...\}\}$$

$Location$ defines the memory address of the Engine-Task on the compute node where ORCOS resides.

- ***DeadlineTime*** is the period of time within which the execution of the current instance of the Engine-Task has to be completed. This time is updated whenever the WorstCaseExecutionTime of the Engine-Task is updated.
- ***TaskArray***

$$TaskArray = \{v_{i,j}; v_{i,j} \quad is \quad a \quad member_i \quad of \quad a \quad controlled \quad task_j\}$$

TaskArray is an array dedicated to store controlled tasks members. Each column represents a controlled task, and includes elements representing the registered members of this controlled task. Each element includes the meta data of the member. Each column also includes three additional elements, the first one holds the last updating point registered by the controlled task, which is represented by this column. The second one indicates the number of elements currently stored for this task and the third one holds the identity *ID/Member* of the currently executed member of the controlled task represented by this column.

- ***NumberOfTasks*** equals the number of columns in *TaskArray*.
- ***WorstCaseExecutionTime (WCET)*** = registration time + acceptance test time + worst case dynamic change test time + worst case dynamic change time + worst case assigning time + time for updating the meta data.
  The *WorstCaseExecutionTime* is bounded since all its factors are bounded, even those, which depend on the number of controlled tasks and the number of their members. These numbers are updated with each execution of the Engine-Task, and since the *WCET* is calculated after each execution of the Engine-Task, this means that the numbers are updated and the *WCET* of these factors can be predicted.

IV. The Adapting Algorithm of the System

The approach assumes EDF to be the scheduling algorithm used. Whenever the Engine-RTC becomes active, the Engine-Task is assigned the highest priority. This is done by assigning a reserved fraction of the processor capacity to the Engine-Task, where this fraction can vary over time, but is always known. At the start of the system, the Engine-RTC is non-active. Whenever a new controlled task or a new member of

 ISBN: 978-1-61208-257-8

a controlled task is added to a compute node, the node sends the meta data of this task to ORCOS. If the Engine-RTC is non-active, a system call is made to make it active. If the meta data includes an arrival time, two cases are to be considered:

1) Arrival of a new controlled task:
   The Engine-task performs an acceptance test $\sum_{i=1}^{n} C_i/T_i \leqslant 1$ on all tasks including the newly arriving one;
   C_i = *ExecutionTime* of the ith task.
   T_i = *DeadlineTime* of the ith task.
   The Engine-task with its *WCET* is taken into consideration when making the acceptance test to ensure enough time capacity for its execution when a new arrival happens after this one. I.e., the adaptation of the processor bandwidth dedicated to the Engine-Task is considered as well.
   If this arrival can be accepted, the Engine-Task fetches the new controlled Task and assigns it to a Controlled RTC. Otherwise, a dynamic change test must be run. The goal is to replace some of the current controlled tasks with alternative members, so that the acceptance test succeeds. In other words, the Engine-task searches for the controlled tasks members of minimal *Cost*, which can substitute the current utilizations C_i/T_i with smaller ones. Thus, the dynamic change test is executed as follows:
   Let the columns of the *TaskArray* have the search order starting from the column, which represents the controlled task with the smallest remaining execution time to the controlled task with biggest remaining execution time. I.e., we want to provide the closest possible acceptance time. If several tasks have the same remaining execution time the task with highest CriticalityDegree is to be looked at first. For the next column j in the *TaskArray*{
   a) Let V_j be the currently executing controlled task member of the column j.
   b) Let $F_j = C_j/T_j$ be the processor utilization for V_j.
   c) For each member $V_{i,j}$\{
      - calculate the utilization $F_{i,j}$:
        $F_{i,j} = C_{i,j}/T_{i,j}$; where
        $C_{i,j}$ is the *Execution-Time* of $V_{i,j}$
        $T_{i,j}$ is the *Deadline-Time* of $V_{i,j}$
      - $Gain_{i,j} = F_j - F_{i,j}$
      - If $Gain_{i,j}$ is positive, extract the respective member's cost: $Cost_{i,j}$
      - Add $V_{i,j}$ with ($Gain_{i,j}$,$Cost_{i,j}$) to a set of candidate members.\}
   d) Select the subset of candidate members of minimal accumulated cost whose accumulated gain is sufficient to accept the new task.
   e) If the accumulated gain is sufficient: break; else inspect next task until all tasks have been visited.\}
   If the new task set is not schedulable, i.e., the maximal accumulated gain is not sufficient, then refuse the newly arrived controlled task.
   Otherwise, replace every selected V_j by the respective selected $V_{i,j}$.
2) Arrival of a new update member of a controlled task:
   The same previous procedure is followed, and the new update starts executing at the next natural updating point of the current executing instance of the controlled task.

The adapting algorithm could be refined to consider updating points other than natural ones. In this case, the previous two cases follow the same steps with one difference. The columns of the *TaskArray* must have the search order starting from the column, which represents the controlled task with the smallest execution time remaining to reach the next updating point to the controlled task with biggest execution time remaining to reach the next updating point. If two tasks have the same execution time remaining to reach the next updating point, the task with the highest CriticalityDegree is to be searched first.

V. Conclusion and Future Work

This paper has introduced an approach to develop a self-adapted and self-management RTOS. This was done by applying the concept of organic programming on RTOS (ORCOS in this case). Further work in this area could be done by, e.g., generalizing the approach to be applied to a larger set of scheduling algorithms, or considering other factors that may have influence on the time capacity, e.g., the non critical tasks, the communication between tasks, memory and other kinds of resources. Currently, we are pursuing to make an experimental evaluation of the principle approach on the RTOS ORCOS.

References

[1] F. Langhammer, O. Imbusch, and G. von Walter, "Ercatons and Organic Programming: Say Good-Bye to Planned Economy," Organic Computing - Controlled Emergence, Vol. 06031, 2006.
[2] <https://orcos.cs.uni-paderborn.de/orcos>, [retrieved: January, 2013].
[3] J. A. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real Time Operating Systems," Operating Systems Review (SIGOPS), Vol. 23, no. 3, February.1989, pp. 54-71.
[4] K. Ramamritham and J. A. Stankovic, "Scheduling Algorithms and Operating Systems Support for Real Time Systems," Proceedings of the IEEE, Vol. 82, no. 1, January.1994, pp. 55-67.
[5] S. Oberthür, L. Zaramba, and H. Lichte, "Flexible Resource Management for Self-X Systems: An Evaluation," in Proceedings of ISORCW'10, May.2010, pp. 1-10.
[6] W. A. Horn, "Some Simple Scheduling Algorithms," Naval Research Logistics Quaterly, Vol. 21, no. 1, March.1974, pp. 177-185.
[7] G. C. Butazzo, "Hard Real-Time Computing Systems. Predictable Scheduling Algorithms and Applications," Springer Science+Business Media, LLC, 2nd rev.ED, 2004.

 ISBN: 978-1-61208-257-8