The Role of Schedulers in Model-Driven Development of Real-Time Systems

Mehrdad Saadatmand, Mikael Sjödin
Mälardalen Real-Time Research Centre (MRTC)
Mälardalen University, Västerås, Sweden
{mehrdad.saadatmand, mikael.sjodin}@mdh.se

Naveed Ul Mustafa
School for Information and Communication Technology (ICT)
Royal Institute of Technology (KTH), Stockholm, Sweden
{naveedum}@kth.se

***Abstract*—Design of real-time embedded systems is a complex and challenging task. Model-driven development has the potential to reduce the design complexity of real-time embedded systems by increasing the abstraction level, enabling analysis at earlier phases of development, and automatic generation of code from the models. In this context, capabilities of schedulers as part of the underlying platform play an important role. They can affect the complexity of code generators and how the model is implemented on the platform. Also, the way a scheduler monitors timing behaviors of tasks and schedules them can facilitate extraction of runtime information. This information can then be used as feedback to the original model in order to identify parts of the model that may require to be re-designed and modified. In this paper, we describe our work in providing these features by introducing a second layer scheduler on top of OSE real-time operating system's scheduler. The approach can also contribute to the predictability of systems by bringing more awareness to the scheduler about the type of real-time tasks (i.e., periodic, sporadic, and aperiodic) that are to be scheduled, and the information that should be monitored and logged for each type.**

***Index Terms*—Real-time operating system, model-driven development, scheduler, runtime monitoring, Enea OSE**

I. Introduction

Model-Driven Development (MDD) is a promising approach to cope with the design complexity of real-time embedded systems. It helps to raise the abstraction level and also perform analysis at earlier phases of development. Therefore, problems in the design of a system can be identified before the implementation phase [1].

Automatic code generation is also one of the end goals in model-driven development. In the context of real-time systems, this means generation of periodic, sporadic and aperiodic tasks. However, most (industrial) Real-Time Operating Systems (RTOS) such as VxWorks, RTEMS, RT-Linux, Windows CE, OSE, and FreeRTOS do not explicitly support the definition of different types of real-time tasks (i.e., periodic, sporadic, and aperiodic) and specification of timing properties for them including period, deadline, Worst-Case Execution Time (WCET), etc. While in theory, a real-time task is simply specified by its timing parameters, in practice and when it comes to implementations, these parameters are introduced in the system in different ways. For example, a periodic task may be implemented in the form of an interrupt while its period is actually set by having a timer to trigger the interrupt periodically. For code generation, this means that for every model element defined as periodic, what is actually generated is an interrupt handler that *behaves* in a periodic way. Therefore, when we look at these systems at runtime, no tangible and single runnable entity as a real-time periodic task is actually observed. In other words, the semantic mapping of a periodic task at model level and such an entity at runtime becomes weak.

Moreover, other parameters of a real-time task, such as deadline, are lost and not present at the implementation level, or taken into account and defined in arbitrary and different ways in each implementation. This is because among all these parameters, what is usually supported explicitly by most real-time operating systems, is to specify only priority for a task. Other parameters are left to be defined and implemented by system designers in arbitrary ways, such as using timer interrupts and delays to enforce periodicity or Minimum Inter-Arrival Time (MIAT). The problem becomes even more evident when it comes to runtime monitoring of real-time systems and where a system needs to detect events such as deadline misses, execution time overruns, etc.

To cope with these problems, we propose a second layer scheduler which takes as input specification and implementation of real-time tasks, including all of their temporal parameters, and schedules and executes them using the underlying scheduler of the operating system. With this design, the code generators can then generate tangible real-time tasks according to a well-defined specification (e.g., definition of a task as: task(task type, period, deadline, execution time)), regardless of whether, for instance, they are going to be actually implemented as a timer interrupt or some other mechanism. This way, an actual real-time task along with its parameters will be present and identifiable at the code level. The top-level scheduler schedules the task and uses its parameters to manage and report events such as deadline misses or execution time overruns. In this approach, even if the underlying platform changes and implementation of real-time tasks (e.g., as timer interrupt) are modified, it will not require any changes in the code generators, and also specification of real-time tasks. This improves the portability of the generated code and transformation engines, making them as *platform-independent* [2] as possible.

In this work, we highlight the role of schedulers in model-driven development of real-time systems and how they fit and contribute to different phases of this approach. We also describe the suggested second layer scheduler we built on OSE real-time operating system [3], and demonstrate how it improves monitoring the timing behaviors of tasks at runtime and detecting events such as deadline misses which are critical in real-time systems.

The remainder of the paper is as follows. In Section II, we discuss the background context and motivation of the work. Section III describes the proposed approach along with its design and implementation details. In Section IV, an example is demonstrated and the implementation and behavior of the suggested approach is evaluated. In Section V we have a look at the related work, and finally in Section VI, we summarize the work and describe its future extensions and directions.

II. Background and Motivation

A. CHESS Project

This work has been done in the context of CHESS European project [4]. This project is about model-driven and component based development of real-time embedded systems for telecommunication, space, railway, and automotive domains which focuses on preservation and guarantee of extra-functional properties [5]. This is done by performing static analysis on design models and monitoring the behavior of the generated system at runtime. The idea is to back-annotate monitored results back to the model to inform the designer which modeled features have led to the violation of specified requirements and may need to be modified. The general structure of the approach is shown in Figure 1.

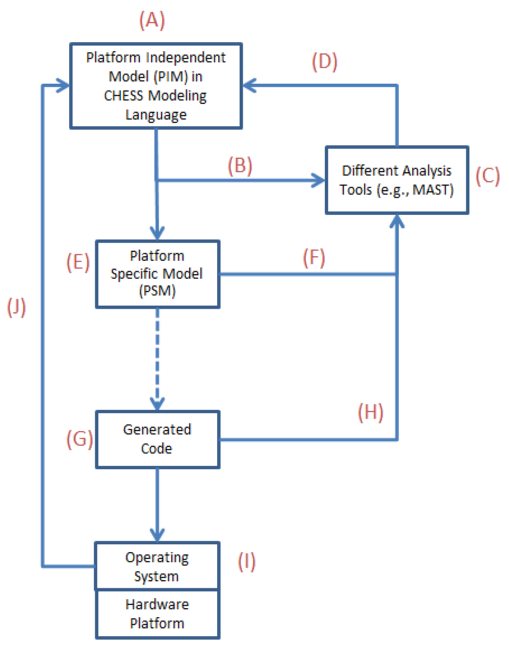
(A)
Platform Independent Model (PIM) in CHESS Modeling Language
(D)
(B)
Different Analysis Tools (e.g., MAST)
(C)
(E)
Platform Specific Model (PSM)
(F)
(J)
(H)
(G)
Generated Code
Operating System
Hardware Platform
(I)

Fig. 1. CHESS methodology

As can be seen from the above figure, different types of analysis are done at different abstraction levels (marked as B, F, and H), and the results are propagated back to the model (D). MAST [6] is one of the analysis tools that is used in CHESS to perform schedulability analysis on the model. The system model which is defined in the modeling language called CHESS ML, is transformed into an appropriate model as input for the MAST analysis tool.

To ensure that the assumptions based on which the analyses were done hold true, the system's execution is monitored and runtime information are collected. For example, the difference between the characteristics of the (ideal) execution environment of the system taken into account for analysis and the actual one when the system is implemented may lead to the violation of the assumptions that are used to perform analysis [7]. Therefore, monitoring the behavior of the system at runtime is important in preservation of system properties.

Timing properties are of utmost importance in real-time embedded systems. In order to extract and collect information about runtime timing behaviors for back-annotation to the model, the platform should be able to provide this information. Among the most important timing data for back-annotation that are of interest are deadline misses, actual response time, and execution time overruns. However, such a feature is not present explicitly in many commercial real-time operating systems today and needs to be implemented in different ways by system developers. In the scope of this work, we narrow our focus on Part I of Figure 1.

B. OSE Real-Time Operating System

OSE is a real-time operating system developed by Enea [8]. It has been designed from the ground up for use in fault-tolerant distributed systems that are commonly found in telecommunication domain, ranging from mobile phones to radio base stations and is embedded in millions of devices around the world [3]. It provides preemptive priority-based scheduling of tasks. OSE offers the concept of direct and asynchronous message passing for communication and synchronization between tasks, and OSE's natural programming model is based on this concept. Linx, which is the Interprocess Communication Protocol (IPC) in OSE, allows tasks to run on different processors or cores, utilizing the same message-based communication model as on a single processor. This programing model provides the advantage of not needing to use shared memory among tasks.

The runnable real-time entity equivalent to a task is called *process* in OSE, and the messages that are passed between processes are referred to as *signals* (thus, the terms process and task in this paper can be considered interchangeably). Processes can be created statically at system start-up, or dynamically at runtime by other processes. Static processes last for the whole life time of the system and cannot be terminated. Types of processes that can be created in OSE are: interrupt process, prioritized process, background process, and phantom process. One interesting feature of OSE is that the same programming model is used regardless of the type of process. Of the timing properties that we have been discussing so far, only priority can be assigned for prioritized processes. Periodic behavior can be implemented by using timer interrupt processes. Information such as task completion time, deadline misses and such, is not reported by default and it needs to be implemented using system level APIs for events such as process swap_in and swap_out which are triggered when a process starts and stops running (i.e., context switches).

C. Goal

Figure 2 shows the target system that is built in the scope of this work. As discussed before, most real-time operating systems, such as OSE, only allow specification of priority for real-time tasks. In other words, semantically, there is no parameter or kernel level value that represents deadline, execution time, or period of a task. Similarly, the monitoring information and logs that are generated by the system do not contain information about deadline misses, or execution time overruns, because these concepts are not actually understood by the kernel and have no meaning for it. Therefore, it is the job of a programmer to implement such features and collect information by also implementing event handlers that monitor when a task gets CPU time and when it is preempted, and then calculate deadline misses or execution time overruns through this information.

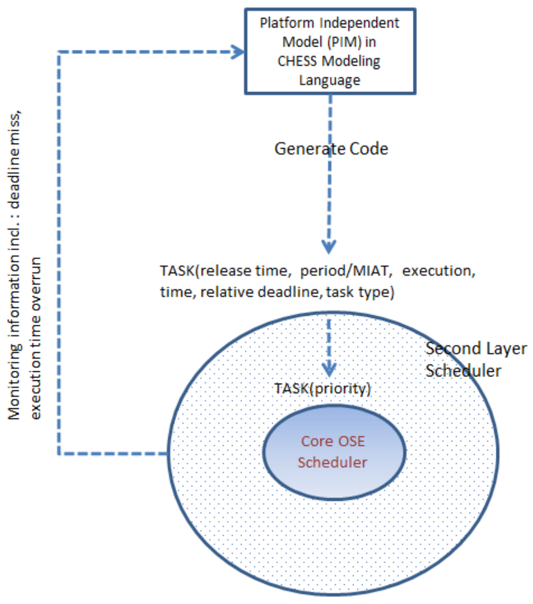
Platform Independent Model (PIM) in CHESS Modeling Language
Generate Code
TASK(release time, period/MIAT, execution, time, relative deadline, task type)
Second Layer Scheduler
TASK(priority)
Core OSE Scheduler
Monitoring information incl. : deadline miss, execution time overrun

Fig. 2. Interfaces between different parts in the suggested approach

The proposed solution that is shown in Figure 2 solves this situation by introducing a second layer scheduler. The interface to this scheduler layer representing the specification of a real-time task is in the form of: *Task(release time, period/MIAT, execution time, relative deadline, task type)*. Considering these parameters, the second layer scheduler then schedules the tasks using the priority-based scheduling mechanism of the core scheduler.

The code generator on the other hand, generates real-time tasks (from the model) according to this specification, and need not care how such a task is actually implemented on the core scheduler, hence more portability and re-usability of the analyzed model and generated code are obtained.

From the monitoring perspective, since the second layer scheduler is responsible for scheduling tasks according to the described specification, it is aware of concepts such as deadline, and can therefore, produce log information representing events such as deadline misses. This generated log information from the second layer scheduler can then be used to propagate necessary information back to the model.

We believe that the suggested added layer in our approach can also help with decreasing the gap between theoretical aspects of real-time systems and their actual implementations by providing more semantics to parameters and specifications of real-time tasks at implementation level and thus increasing the applicability of theoretical knowledge such as schedulability analysis techniques.

III. Scheduler Design and Implementation

In this section, the internal mechanism and design details of the second layer scheduler are described.

The second layer scheduler developed on top of OSE, schedules a given set of tasks (S) by releasing tasks to OSE core scheduler according to a selected scheduling policy. S can contain three kinds of tasks: Periodic, Sporadic, and Aperiodic tasks. Task parameters such as period and execution time are generated for the second layer scheduler from the model as input parameter files with .prm extension.

As shown in Figure 3, the system consists of a few other components besides the second layer scheduler process. At system startup, first process creator is started. Process creator creates an OSE process for each of the tasks that are specified as a set of input files. Initially, all these processes will be in the waiting mode (to receive a signal from the second layer scheduler process). From this point on, the second layer scheduler process, which has the highest priority in the system, controls the system. Based on a specified scheduling policy (e.g., EDF), the second layer scheduler selects an appropriate task from the queue of waiting tasks, and sends a *start* signal to it. It then enters a waiting state itself using OSE *receive_w_tmo* (receive with timeout) system call. This system call makes the caller process wait until it either receives a signal or the specified timeout expires. We make a specific use of this system call in our design by setting its timeout value equal to the time interval available before arrival of a new instance of a higher priority periodic task. Also, whenever a task finishes execution, it sends a completion signal back to the second layer scheduler process. Therefore, if the running task finishes its job before arrival of the next instance of a higher priority periodic task, the second layer scheduler will receive a completion signal (at the *receive_w_tmo* system call), and continues its job (scheduling next tasks). Otherwise, if the running task takes too much time, the timeout which is set in the *receive_w_tmo* command in the second layer scheduler will expire, and since the second layer scheduler process has the highest priority in the system, it preempts the running task, takes the CPU, checks the list of waiting tasks again, and selects the next appropriate task to run.

Right after receiving the completion signal by the second layer scheduler process, it generates log information about the behavior of the task which has just completed. Since the second layer scheduler has access to (and thus is aware of) all the real-time parameters of each task (e.g., periodic/MIAT, deadline, execution time), it can gracefully detect deadline misses, execution time overruns, and events of this kind, mark them in the log information and report them. This way, all this critical log information about the behavior of the system are also centralized, which can then be easily queried. This is an important feature which is absent in many real-time operating systems today.

Creation of monitoring log files and persistence of the collected information are done by the monitor process using the information that is sent to it by the second layer scheduler process in the form of signals. In this design, two separate log files are actually created: scheduling log file, and monitoring log file. Scheduling log file contains listing of schedules generated by the second layer scheduler by stating the time points at which a task in the task set is scheduled, completed or preempted. This log file is generated by the second layer scheduler. Events related to task deadlines can be investigated by examining the monitoring log file generated by the scheduler. Monitoring log file is updated with new information only when an instance of a task is completed, and scheduling log file is updated whenever a task is scheduled, preempted, resumed or completed.

The scheduling policy that the second layer scheduler uses for periodic tasks is selectable and not fixed. Same is the case with the scheduling mechanism for aperiodic and sporadic tasks. The selected policy is read as a configuration value at system startup. This makes the suggested approach flexible. Currently Rate Monotonic Scheduling (RMS) and Earliest Deadline First (EDF) policies are supported for periodic tasks while aperiodic and sporadic tasks can be scheduled using background [9] or polling server [9], [10] schemes. Other policies can also be added to design.

In the following sections, the role of different components in our design are described in detail.

A. System Components

1) Process Creator: Each task in a task set is specified by two files.

- **Parameter File:** A file with .prm extension provides task parameters including release time, period/MIAT, execution time, relative deadline and type of the task.

  Type of task can have four valid values. Different task types and the values by which, each task type is represented in the system are mentioned in Table I.

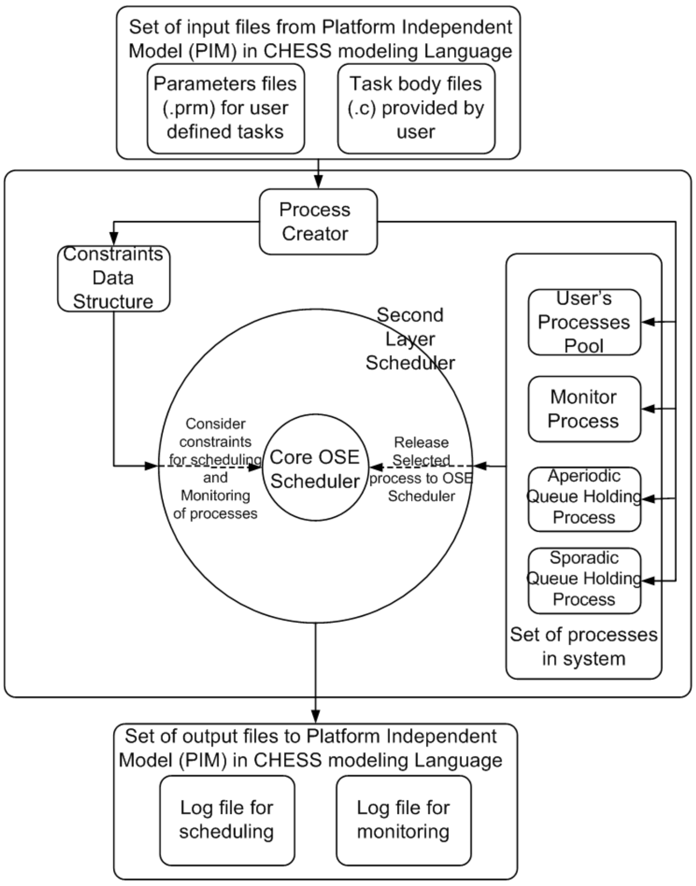
Set of input files from Platform Independent Model (PIM) in CHESS modeling Language
Parameters files (.prm) for user defined tasks
Task body files (.c) provided by user
Process Creator
Constraints Data Structure
Second Layer Scheduler
Consider constraints for scheduling and Monitoring of processes
Core OSE Scheduler
Release Selected process to OSE Scheduler
User's Processes Pool
Monitor Process
Aperiodic Queue Holding Process
Sporadic Queue Holding Process
Set of processes in system
Set of output files to Platform Independent Model (PIM) in CHESS modeling Language
Log file for scheduling
Log file for monitoring

Fig. 3. Components of Design

Task Type	Value
Periodic	0
Sporadic	1
Aperiodic	2
Polling Server	3

TABLE I
VALUES FOR DIFFERENT TASK TYPES

- **Body File:** A file with .c extension contains the body of the task. In other words, .prm file of a task contains its timing non-functional specification while .c file contains its functional implementation.

Process creator reads the parameters for each task from its .prm file into a data structure, called ”constraints”, and creates a prioritized OSE process against each user defined task. Moreover, It also creates following four OSE prioritized processes:

- Second layer scheduler process
- Sporadic queue holding process
- Aperiodic queue holding process
- Monitor process

None of the created OSE processes is started by process creator except the second layer scheduler process. Task parameters and Process Identifiers (PIDs) for all processes are then passed to the second layer scheduler in the form of “constraints” data structure.

2) Second Layer Scheduler: After receiving “constraints” structure and PIDs of all OSE processes created by process creator, the second layer scheduler schedules the tasks by releasing them to core OSE scheduler according to selected scheduling algorithm. The design provides options to select between RMS or EDF algorithm.

To schedule sporadic and aperiodic tasks, the second layer scheduler supports background scheduling and polling server for scheduling sporadic and aperiodic tasks.

3) Sporadic Queue Holder: Task set can contain periodic, aperiodic and sporadic tasks. Sporadic queue holder is a prioritized OSE process which maintains a list of sporadic tasks waiting for scheduling, by using a queue. Each element of queue contains two parameters for a sporadic task: PID corresponding to the given task, and release time of sporadic task.

To release a sporadic task, its PID and release time is to be placed in queue. An interrupt process (in case of hardware driven sporadic tasks) or a prioritized process (in case of software driven sporadic tasks) may initiate this placement by sending a signal to sporadic queue holder.

Upon receiving this signal, sporadic queue holder extracts the PID and release time of sporadic task from signal and updates the queue with extracted information.

The second layer scheduler checks if there is a sporadic task to be scheduled by making a query to sporadic queue holder process.

4) Aperiodic Queue Holder: Aperiodic queue holder has the same structure and mechanism as the sporadic queue holder, except that it maintains a list of aperiodic tasks. Also separate signals are defined for use with aperiodic and sporadic queue holders.

5) Monitor: Monitor process generates a log file to state whether specified timing constraints for each task in task set S are met or not. For example, if the specified MIAT parameter, in case of a sporadic task, is violated then monitor records this violation in a monitoring log file.
When a task is released to the core OSE scheduler for execution, the second layer scheduler observes its timing parameters. As soon as a task completes its execution, the second layer scheduler sends a signal to the monitor process. This signal contains start time, completion time, desired deadline, desired execution time, desired MIAT, actual execution time, and actual MIAT of completed task. These timing values are extracted from .prm file of the task under monitoring (i.e., desired deadline and execution time) and measured by the second layer scheduler (i.e., actual deadline and execution time). Monitor extracts these timing values from the received signal and saves the relevant monitoring statements in a monitoring log file.

B. Signals and Communications

To achieve the scheduling of tasks in a reliable way, several signals are defined and used by system. These signals play two important roles: carry required data from one component to another, and ensure synchronous execution of all components.
These signals are described below.

- **start_exe_sig:** Start execution signal. This signal is sent by the second layer scheduler to a process to be scheduled on core OSE scheduler. Target process can start execution only if it has received start_exe_sig signal.
- **comp_sig:** Completion signal. This signal is sent as an acknowledgment to the second layer scheduler upon completion by a process which is created against a user defined task.
- **aper_update_sig:** Update signal for aperiodic queue holder. To release an aperiodic task, an interrupt process or prioritized process sends the aper_update_sig signal to aperiodic queue holder process. This signal contains the PID and release time of an aperiodic task to be scheduled. This signal is also used as a response to the second layer scheduler by aperiodic queue holder on receiving start_exe_sig from scheduler.
- **spor_update_sig:** Update signal for sporadic queue holder. To release a sporadic task, an interrupt process or prioritized process sends the spor_update_sig signal to sporadic queue holder process. This signal contains the PID and release time of sporadic task to be scheduled.
This signal is also used as a response to the second layer scheduler by sporadic queue holder on receiving start_exe_sig from scheduler.
- **qupdate_confirm_sig:** Queue update confirmation signal. This confirmation signal is sent back to the sender of an update signal, after receiving aper_update_sig (in case of aperiodic queue holder) or spor_update_sig (in case of sporadic queue holder). Confirmation signal informs sender if queue is updated successfully. In case of failure, sender can send aper_update_sig again with same content after waiting for a finite amount of time.
- **monitor_info_sig:** Monitoring information signal. This signal is sent by the second layer scheduler to the monitor, every time a task completes its execution. Monitor uses the information contained in this signal to determine if a completed task has met its constraints, such as deadline and WCET.

C. Priority Assignment

In OSE, there are 32 priority levels. Priority 0 is considered the highest while 31 is considered as the lowest priority level. In our system, process creator creates one OSE process for each task in the input task set. All such processes are assigned priority level of 1. Similarly, sporadic queue holder process and aperiodic queue holder process have priority level of 1. However, the second layer scheduler process has priority level of 0 which is the highest possible priority level. The reason for assigning priority level 0 to the scheduler is to make it non preemptable by any other prioritized OSE process.

Monitor behaves as a background OSE process and hence has the lowest priority level. This ensures that monitoring is performed only when no task is ready and the scheduler is idle. This reduces the effect of monitoring on the scheduling of tasks.

D. Scheduling of Tasks

As described in the previous section, all OSE processes are created by Process creator but not started by it. Process creator starts only the second layer scheduler process and passes “constraints” structure along with PIDs of all OSE processes.

1) Scheduling of Periodic Tasks: The second layer scheduler examines the “type” parameter of all tasks to identify periodic tasks among the task set. Tasks are scheduled by releasing them to core OSE scheduler according to specified scheduling algorithm, for example RMS.

The second layer scheduler sends start_exe_sig to the the process representing the user defined task which has highest priority according to selected scheduling algorithm. Then scheduler waits for receiving comp_sig back from target OSE process but with a finite waiting time called “timeout”.

If comp_sig is received before the timeout is expired, it implies that the target process has completed. Hence the second layer scheduler releases to the core OSE scheduler the next ready OSE process representing the user defined periodic task. If comp_sig is not received within the timeout duration and a process representing a user defined task with higher priority is ready, then the former process is preempted and second layer scheduler releases to core OSE scheduler the process with higher priority.

2) Scheduling of Sporadic and Aperiodic Tasks: To schedule a sporadic task, it is necessary that its corresponding PID and release time are placed in the sporadic processes queue maintained by sporadic queue holder. This can be achieved by sending a spor_update_sig signal to the sporadic queue holder containing release time and PID of the OSE process corresponding to the task. Signal, spor_update_sig, can be sent to the sporadic queue holder either by an interrupt OSE process or a prioritized OSE process. In the first case, target sporadic task becomes interrupt driven while in second case it behaves as a program driven sporadic task. The above discussion is valid also for achieving interrupt and program driven behavior for aperiodic tasks.

Sporadic and aperiodic tasks can be scheduled by using one of following two approaches:

- Background Scheme: One approach to schedule sporadic and aperiodic tasks is to use time slots in which no periodic task is ready to run. In such case, the second layer scheduler first makes query to sporadic queue holder by sending start_exe_sig to find if there is any ready sporadic task. The spor_update_sig signal is sent back by sporadic queue holder to the second layer scheduler, indicating availability status of sporadic task.
  If there is a ready sporadic task, the second layer scheduler releases sporadic task to OSE core scheduler and waits until either it completes its execution or a periodic task becomes ready. If sporadic task is completed and no periodic task is ready to run, second layer scheduler again makes query to sporadic queue holder to find if there are any more sporadic tasks waiting in the queue.
  If sporadic task queue is empty and no periodic task is ready to run, the second layer scheduler makes query to aperiodic queue holder by sending start_exe_sig. Availability status of aperiodic task is communicated back to the second layer scheduler by sending aper_update_sig from aperiodic queue holder. If aperiodic queue is not empty and aperiodic task at the head of the queue is ready to run, the second layer scheduler releases that aperiodic task to core OSE scheduler.
  If there is no periodic, sporadic and aperiodic task to execute, Monitor process is released to core OSE scheduler by the second layer scheduler.
- Polling Server Scheme: An alternative approach to schedule sporadic and aperiodic tasks is to use polling server. Polling server is a periodic task like any other periodic task. It has a period P_s and execution time E_s. Execution time of polling server is known as its budget.
  Polling server is scheduled along with all other periodic tasks according to selected scheduling algorithm. However, when polling server gets the chance to execute, the second layer scheduler makes query to sporadic and aperiodic queue holding processes to find if there is any ready sporadic or aperiodic task. If sporadic or aperiodic task is ready to run, the second layer scheduler releases that task to OSE core scheduler and budget of polling server keeps declining per unit time.
  If sporadic or aperiodic task completes its execution before budget is expired, the second layer scheduler picks next ready sporadic or aperiodic task to release to OSE core scheduler. This sequence continues until either there is no sporadic or aperiodic task or budget of server is expired or a higher priority periodic task becomes ready to execute.
  At the start of each period of the polling server, its budget is set equal to its execution time. If at that time point, no sporadic or aperiodic task is ready to run then budget immediately declines to zero. Otherwise the budget decreases one level per time unit.

E. Monitoring of Tasks

On completion of a task, independent of its type, the second layer scheduler sends monitor_info_sig signal to Monitor. Monitor is implemented as a background OSE process. Hence, it can execute only when there is no periodic, sporadic or aperiodic task ready to run. Monitor continuously checks its input message queue for monitor_info_sig signal. This message carries following information to the monitor process regarding completed task:

- start time of the task

- completion time of the task
- specified deadline parameter for the task
- specified MIAT parameter for the task
- specified execution time for the task
- actual execution time for task
- actual deadline for task
- actual MIAT for the task

Monitor uses this information to make decision if a completed task has met its parameters or violated them. In any case, monitor records the information in a monitoring log file.

Operation of the scheduler is summarized by the sequence diagram of Figure 4. In this diagram, the task set consists of two periodic tasks T_1 and T_2, one sporadic task T_3 and an aperiodic task T_4. Timing parameters of the tasks defined in this task set are listed below using the specification convention: Task (Release Time, Period, WCET, Relative deadline, Task type).

Periodic task: $T_1(0, 12, 3, 8, 0)$
Periodic task: $T_2(0, 4, 1, 3, 0)$
Sporadic task: $T_3(0, 15, 2, 6, 1)$
Aperiodic task: $T_4(0, 0, 1, 6, 2)$

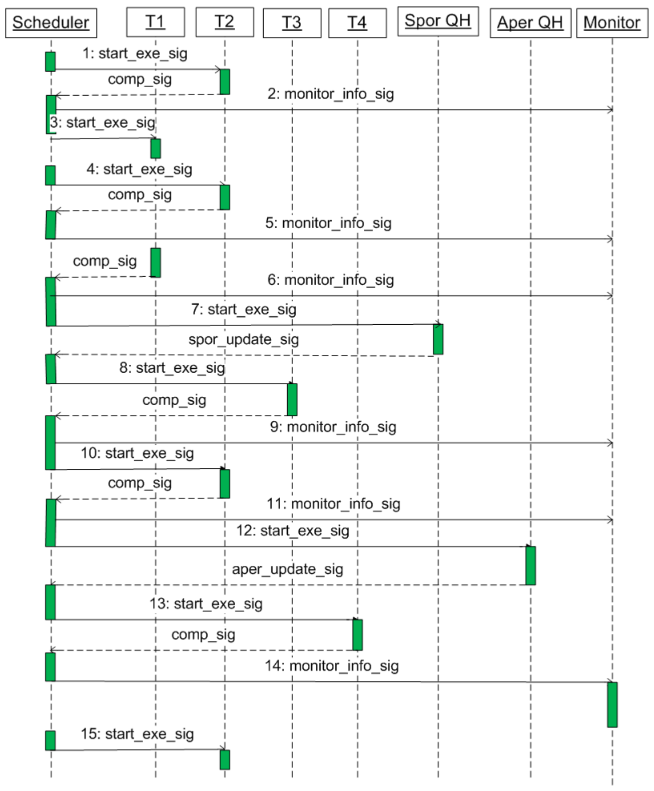

Fig. 4. Sequence diagram to demonstrate operation of the scheduler

This figure shows a valid sequence of execution when RMS is used as the scheduling policy for periodic tasks, while sporadic and aperiodic tasks are scheduled using background scheme. As is evident from the sequence diagram, monitor gets the chance to execute only when no other process is in ready state.

IV. Experiment and Monitoring Results

The described approach has been implemented and tested on OSE SoftKernel (SFK) version 5.5.1 [8]. In this section, we show an example of a task set which is implemented based on the specification of the proposed second layer scheduler. The way the task set is scheduled and the log information that is generated for the behaviour of tasks are illustrated.

A task set consisting of four tasks is created. Periodic tasks in the task set are configured to be scheduled using RMS while aperiodic and sporadic tasks are to be scheduled using polling server scheme (this can be changed by changing configuration parameters).

Timing constraints for tasks are given below (last parameter identifies the type of task):

- $T_1(0, 10, 2, 5, 3)$: Polling server with release time=0, period=10, WCET/Budget=2, Relative Deadline=5, Task type=3.
- $T_2(0, 5, 2, 4, 0)$: Periodic Task with release time=0, period=5, WCET=2, Relative Deadline=4, Task type=0.
- $T_3(0, 5, 2, 4, 1)$: Sporadic Task with release time=0, MIAT=5, WCET=2, Relative Deadline=4, Task type=1. Two instances of Sporadic task T_3 are released at time 0;
- $T_4(0, 0, 2, 7, 2)$: Aperiodic Task with release time=0, period= 0 (Not Applicable), WCET=2, Relative Deadline=7, Task type=2.

As mentioned before, these timing parameters are actually specified in the .prm file of each task (i.e., t1.prm, ..., t4.prm). Process creator opens these files and populates "constraints" data structure with these data.

Using the implemented second layer scheduler to schedule this task set, the following log files are automatically generated:

- **Scheduling log file:** Scheduling log file provides the time points for each task at which it is scheduled, preempted/not completed, resumed or completed. Parts of the scheduling log information generated for the task set are shown in Listing 1. To make it easier to follow and understand the information in the log file, the PIDs that are assigned to each task by the system are also mentioned below:
  - PID of process representing T_1 = 65595.
  - PID of process representing T_2 = 65596.
  - PID of process representing T_3 = 65597.
  - PID of process representing T_4 = 65598.

Listing 1. Scheduling log file

```
task PID=65596
Scheduled for 5 ticks at ticks=1115
task PID=65596
Completed at ticks=1117
task PID=65597
Scheduled with budget= 2 ticks at ticks=1117
task PID=65597
Not completed at ticks=1119
Remaining Execution Time in ticks=1
task PID=65596
Scheduled for 5 ticks at ticks=1120
task PID=65596
Completed at ticks=1122
```

```
task PID=65596
Scheduled for 5 ticks at ticks=1125
task PID=65596
Completed at ticks=1127
task PID=65597
Resumed with budget= 2 ticks at ticks=1127
task PID=65597
Completed at ticks=1128
task PID=65597
Scheduled with budget= 1 ticks at ticks=1128
task PID=65597
Not completed at ticks=1129
Remaining Execution Time in ticks=1
task PID=65596
Scheduled for 5 ticks at ticks=1130
task PID=65596
Completed at ticks=1132
task PID=65596
Scheduled for 5 ticks at ticks=1135
task PID=65596
Completed at ticks=1137
task PID=65597
Resumed with budget= 2 ticks at ticks=1137
task PID=65597
Completed at ticks=1138
task PID=65598
Scheduled with budget= 1 ticks at ticks=1138
task PID=65598
Not completed at ticks=1139
Remaining Execution Time in ticks=1
task PID=65596
Scheduled for 5 ticks at ticks=1140
task PID=65596
Completed at ticks=1142
```

- **Monitoring log file:** On completion of each instance of a task, monitoring log file lists type of task, PID of process representing that task in system, start time of the task, specified deadline, completion time of the task, specified WCET for the task, actual execution time consumed by the task, response time of the task, specified MIAT/period and actual interval between two consecutive invocations of the task. Listing 2 shows parts of the monitoring log information generated for the task set.

Listing 2. Monitoring log file

```
PID =65596
Type of task =0
 start time in ticks=1115
 specified deadline in ticks=1119
 completion time in ticks =1117
 specified WCET in ticks=2
 actual execution time in ticks=2
 Response time in ticks =2
 specified Period/MIAT in ticks =5
 Interval between two consecutive invocations in ticks=5
PID =65596
Type of task =0
 start time in ticks=1120
 specified deadline in ticks=1124
 completion time in ticks =1122
 specified WCET in ticks=2
 actual execution time in ticks=2
 Response time in ticks =2
 specified Period/MIAT in ticks =5
 Interval between two consecutive invocations in ticks=5
PID =65596
Type of task =0
 start time in ticks=1125
 specified deadline in ticks=1129
 completion time in ticks =1127
 specified WCET in ticks=2
 actual execution time in ticks=2
 Response time in ticks =2
 specified Period/MIAT in ticks =5
 Interval between two consecutive invocations in ticks=5
PID =65597
Type of task =1
 start time in ticks=1117
 specified deadline in ticks=1121
 completion time in ticks =1128
 specified WCET in ticks=2
 actual execution time in ticks=3
 Response time in ticks =11
 specified Period/MIAT in ticks =5
 Interval between two consecutive invocations in ticks=10
PID =65596
Type of task =0
 start time in ticks=1130
 specified deadline in ticks=1134
 completion time in ticks =1132
 specified WCET in ticks=2
 actual execution time in ticks=2
 Response time in ticks =2
 specified Period/MIAT in ticks =5
 Interval between two consecutive invocations in ticks=5
PID =65596
Type of task =0
 start time in ticks=1135
 specified deadline in ticks=1139
 completion time in ticks =1137
 specified WCET in ticks=2
 actual execution time in ticks=2
 Response time in ticks =2
 specified Period/MIAT in ticks =5
 Interval between two consecutive invocations in ticks=5
PID =65597
Type of task =1
 start time in ticks=1128
 specified deadline in ticks=1132
 completion time in ticks =1138
 specified WCET in ticks=2
 actual execution time in ticks=2
 Response time in ticks =10
 specified Period/MIAT in ticks =5
 Interval between two consecutive invocations in ticks=9
```

The first four lines in the generated scheduling log information shown in Listing 1 indicates that the periodic task T_2 with PID of 65596, which is started at tick time 1115, has completed at 1117. The next task which is scheduled is T_3 with PID of 65597. The polling server has the capacity of two time unit at this time instance, therefore, the sporadic task T_3 can run until tick time 1119, and at 1120 another instance of T_2 arrives which causes the second layer scheduler to preempt T_3. However, at 1119, T_3 has not managed to complete its job, and therefore, it is marked as 'Not completed'.

On the other hand, the monitoring information in Listing 2, among other things, can be used to check whether any deadline miss has occurred or not. For example, it shows that the first two instances of T_2 (PID=65596) have met their deadlines. The first instance has finished its job at 1117 and finished before its deadline which is 1119. The deadline for the second instance is at 1124, and it has managed to complete its job at 1122, and therefore, meet its deadline. However, the deadline of the sporadic task T_3, with PID of 65597, has been 1121 while it has managed to finish its job at 1128. Its actual execution time has also been three time units which is one time unit more than its specified WCET. This shows that there has been execution time overrun for this task and there is something wrong with the specified WCET value of it, and it needs to be re-considered. Such information are hardly provided by default in any real-time operating system.

Figure 5 visualizes the schedule generated by the scheduler. This figure is created (manually) using the information available in the scheduling log file generated by the scheduler.

The scheduling log file shows that the first task is released at 1115 system ticks. To make this schedule easier to understand in the figure, subtraction of 1115 ticks is performed at every time point.

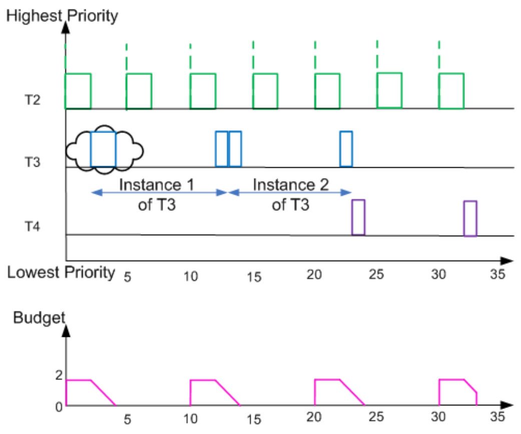

Fig. 5. Schedule generated by the design using second layer scheduler

As is indicated by a cloud symbol in Figure 5, actual execution time consumed by first instance of sporadic task is 3 ticks instead of 2 ticks as specified in timing constraint of WCET=2. Therefore, it misses its deadline of 5 ticks and is completed at 13 ticks. Second instance of sporadic task is scheduled immediately after completion of the first instance. This is because MIAT of 10 ticks is already elapsed (10+2=12). The diagram in the lower part of Figure 5 indicates the replenishment and decrease of budget with passage of time as is defined for the behavior of polling servers.

Now that the necessary information about the runtime behavior of tasks is provided in the log files generated by the system, a user can easily query them, extract desired parts, and draw conclusions. For example, it is very easy to find out the number of deadline misses, execution time overruns, the task with maximum number of deadline misses, etc. by using the log files as the data source. Similarly, at any time point, the number of periodic, sporadic, and aperiodic tasks in the system can easily be requested from the second layer scheduler; a simple but important feature which is not provided by default in many RTOSes today. Also, it is now possible to identify and report the time period during which maximum number of deadline misses have occurred, and examine as well how the system has been behaving in terms of context switches and preemptions during that period. These are features whose implementations can be very hard and complex without having the necessary monitoring information and using the suggested approach.

V. Related Work

Many of the operating systems and also programming languages today provide support for measuring the CPU time a runnable entity (i.e., thread, etc.) consumes to perform its function. However, the monitoring facilities and event handling mechanisms provided by these platforms are not usually integrated with their scheduling facilities [11]. As a result, the platform cannot enforce and monitor real-time properties of threads such as their allowed execution times and deadlines. Real-Time Specification for Java (RTSJ) [11] is introduced to integrate scheduling of threads with the execution time monitoring facilities and enforce execution budgets on them in Java.

The work described in [12] is an attempt to implement sporadic servers in Ada. It also uses the concept of queues to manage sporadic and aperiodic tasks as we did here. However, it does not discuss real-time applications built on this design and using these servers; i.e., how sporadic and aperiodic tasks are defined and introduced to the system. [13], which also targets sporadic tasks in Ada, highlights the problem that we mentioned in this paper which many real-time platforms suffer from. It states the issue by claiming that the underlying kernel needs to provide complex execution time monitoring mechanisms at runtime and that ”such mechanisms are not generally supported by Ada 9X”.

One of the closest works to our approach is the implementation of a new scheduling class called *SCHED_DEADLINE* for the Linux kernel that adds EDF scheduling policy support to Linux [14]. It is also motivated by acknowledging the fact that due to limited support for specifying timing constraints for real-time tasks (e.g., deadline) and lack of control over them, feasibility study of the system under development and guaranteeing the timing requirements of tasks are not possible. There are however several differences between this work and ours. It focuses only on mechanisms for adding EDF scheduling policy to the Linux kernel, while we target the problem in a more general manner and allowing the scheduling policy to be configurable. Our focus is mainly on improving the monitoring of real-time events by providing more control over real-time tasks and more knowledge about their timing constraints to the scheduler regardless of the scheduling policy. Moreover, we try to provide an abstraction layer around the core scheduler to hide platform-dependent implementation details from the user, while SCHED_DEADLINE tries to solve a different problem and is basically added as a separate module to the system.

One concept which also introduces different levels of abstraction around a core scheduler is Hierarchical Scheduling Framework (HSF) [10], [15]. There are fundamental differences between what we introduced here and HSF. HSF is a modular approach in which a system is divided into several subsystems. The subsystems are scheduled by a global (core) scheduler, while the tasks in each subsystem are scheduled by local (sub-system) level schedulers. The structure that we introduced here does not try to divide a system into different subsystems where each of these subsystems may be scheduled differently by a different scheduler.

There are also studies that focus on execution monitoring of real-time systems. Many of these studies, such as [16], try to predict timing violations in the system in different ways, for example, using statistical models. Our suggested approach does not to try to predict violations and produces precise monitoring information for behavior of real-time tasks and violation of timing constraints. The monitoring part in our approach is coupled with the scheduler and by bringing awareness to the scheduler about the type of tasks it is scheduling, monitoring such information becomes a natural and straightforward part of the scheduler.

VI. Discussion and Conclusion

In this paper, we introduced the concept of the second layer scheduler as an approach to bring semantics and awareness for different types of real-time tasks and their parameters to the scheduler. It was shown how this awareness improves the monitoring capabilities of the system to help with the detection of critical events such as deadline misses, and execution time overruns. While the approach was motived and described in the context of model-driven development of real-time systems, and how it can contribute to ease code generation and enable back-annotation of data, it does not necessarily need to be used in this context and the concept of the second layer scheduler is applicable and practical per se.

Considering a larger set of timing parameters for scheduling of tasks and generating detailed log information in the second layer scheduler can bring along their own overheads. Measurement of these overheads and evaluation of the price of these added features are left to be done as a future work. Especially we plan to perform two overhead measurements: startup overhead (reading configurations and initializing tasks), and context switch and scheduling decisions overheads. It should however be noted that the actual logging is done by the monitor process in our design which behaves as a background process. The second layer scheduler only sends out a signal (including needed information) to the monitor process and continues its job (asynchronously) without using any critical section for data sharing by using message passing mechanisms of OSE. This way, the overhead of creating log information in the second layer scheduler process is tried to be mitigated.

Generation of such detailed monitoring information can also help with the predictability of real-time systems at runtime. For instance, even in cases where no deadline misses occur in the system, it becomes possible to observe how close tasks are to missing their deadlines and whether this gap is decreasing or increasing. Based on such analysis of monitoring information, the system can also adapt itself in order to prevent deadline misses.

One issue that we did not discuss in this paper is the priority inversion problem. This problem is handled automatically by OSE for communication among periodic tasks, but for sporadic and aperiodic tasks, the priority inversion issue should be more investigated. Also the way the system is designed for the background scheme, periodic tasks will have higher priority over sporadic tasks, and the priority of sporadic tasks will be higher than aperiodic ones. When the polling server scheme is used, the priority of sporadic tasks will be dependent on the priority of their periodic server, but still higher than aperiodic ones. This can also be extended to be configurable by the user. Moreover, in this work, since the tasks were assumed to be generated from a model, the task set was considered to be known and static. We leave the extension of the implementation to accept new tasks dynamically as a future work. Also, the possibility to have different numbers and types of servers for sporadic and aperiodic tasks could be another future work.

Since the suggested approach is designed to be flexible in terms of the used scheduling algorithms, it would be interesting as a future direction of this work, to investigate the possibility to let the system intelligently select an appropriate/optimal scheduling algorithm based on the requirements at the model level, and generate code accordingly, especially that the back-annotation mechanism can also be used as a feedback loop.

VII. Acknowledgements

This work has been partially supported by the European Project CHESS (ARTEMIS-JU100022) [4] and Enea Services Stockholm AB [8].

References

[1] B. Selic, “The pragmatics of model-driven development,” *IEEE Software*, vol. 20, pp. 19–25, September 2003.

[2] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture - Practice and Promise*, 2003.

[3] Enea, “The architectural advantages of enea ose in telecom applications,” http://www.enea.com/software/products/rtos/ose/, Last Accessed: February 2012.

[4] CHESS Project: Composition with Guarantees for High-integrity Embedded Software Components Assembly, http://chess-project.ning.com/, Last Accessed: February 2012.

[5] L. M. Cysneiros and J. C. S. do Prado Leite, “Non-functional requirements: From elicitation to conceptual models,” in *IEEE Transactions on Software Engineering*, vol. 30, no. 5, 2004, pp. 328–350.

[6] Modeling and Analysis Suite for Real-Time Applications (MAST), http://mast.unican.es/, Last Accessed: February 2012.

[7] S. Chodrow, F. Jahanian, and M. Donner, “Run-time monitoring of real-time systems,” in *Real-Time Systems Symposium (RTSS). Proceedings., Twelfth*, dec 1991, pp. 74 –83.

[8] Enea, http://www.enea.com, Last Accessed: February 2012.

[9] B. Sprunt, “Aperiodic task scheduling for real-time systems,” Ph.D. thesis, Carnegie Mellon Univ, Tech. Rep., 1990.

[10] R. Davis and A. Burns, “Hierarchical fixed priority pre-emptive scheduling,” in *In Proceedings of the 26 th IEEE International Real-Time Systems Symposium (RTSS’05)*, 2005, pp. 389–398.

[11] A. J. Wellings, G. Bollella, P. C. Dibble, and D. Holmes, “Cost enforcement and deadline monitoring in the real-time specification for java.” in *7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*. IEEE Computer Society, 12-14 May 2004, pp. 78–85.

[12] B. Sprunt and L. Sha, *Implementing sporadic servers in Ada*, ser. Technical report. Carnegie Mellon University, Software Engineering Institute, 1990.

[13] A. Burns and A. J. Wellings, “Implementing analysable hard real-time sporadic tasks in ada 9x,” *Ada Letters.*, vol. XIV, pp. 38–49, January 1994.

[14] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino, “An EDF scheduling class for the Linux kernel,” in *Proceedings of the Eleventh Real-Time Linux Workshop*, Dresden, Germany, Sep. 2009.

[15] T. Nolte, M. Behnam, M. Åsberg, R. J. Bril, and I. Shin, “Hierarchical scheduling of complex embedded real-time systems,” in *École d’Éte Temps-Réel (ETR’09)*, August 2009, pp. 129–142.

[16] Y. Yu, S. Ren, and O. Frieder, “Prediction of timing constraint violation for real-time embedded systems with known transient hardware failure distribution model,” in *Real-Time Systems Symposium, 2006. RTSS ’06. 27th IEEE International*, dec. 2006, pp. 454 –466.