UNIVERSITÄT AUGSBURG

`tms-sim` – Timing Models Scheduling Simulation Framework Release 2016-07

Florian Kluge

Report 2016-04 Juli 2016

INSTITUT FÜR INFORMATIK

D-86135 AUGSBURG

Copyright © Florian Kluge
Institut für Informatik
Universität Augsburg
D–86135 Augsburg, Germany
http://www.informatik.uni-augsburg.de
— all rights reserved —

`tms-sim` is a framework for the simulation and evaluation of scheduling algorithms. It is being developed to support our work on real-time task scheduling based on time-utility and history-cognisant utility functions. We publish `tms-sim` under the conditions of the GNU GPL to make our results reproducible and in the hope that it may be useful for others. This report describes the usage of the TMS framework libraries and how they can be used to build further simulation environments. It is not intended as a documentation of the single classes, which can be found in the Doxygen documentation.

Contents

Contents 5

1 Introduction 7

2 Core Package 9
2.1 Scheduling Objects . . . 9
2.1.1 Job . . . 9
2.1.2 Scheduler . . . 9
2.1.3 Task . . . 10
2.1.4 Time Utility Functions . . . 10
2.1.5 History-Cognisant Utility Functions . . . 11
2.1.6 General Remarks on Tasks and Utility Functions . . . 11
2.2 Simulation . . . 11
2.3 Helper Classes . . . 12

3 Implementations 15
3.1 Schedulers . . . 15
3.1.1 ALD . . . 15
3.1.2 FPP . . . 15
3.1.3 EDF . . . 16
3.1.4 OEDF . . . 16
3.1.5 GDPA & GDPA-S . . . 17
3.2 Task Models . . . 17
3.2.1 PeriodicTask . . . 17
3.2.2 MKTask . . . 18
3.2.3 SporadicTask . . . 18
3.2.4 SPTask . . . 18
3.3 Utility Functions . . . 18
3.3.1 Time-Utility Functions – `UtilityCalculator` . . . 18
3.3.2 History-Cognisant Utility Functions - `UtilityAggregator` . . . 19

4 Task Set Generator 21
4.1 MKGenerator . . . 21
4.2 AbstractMkTaskset . . . 22
4.3 Tasksets with Dependencies . . . 22

5 Utility Classes 25
5.1 Logging 25
5.2 Key/Value Files 25

6 Executables 27
6.1 TMS-Sim 27
6.2 MKEval 27
6.3 RunTS 29
6.4 tseval 29
6.5 MkDSE 30
6.6 Tms-Vis 31
6.7 Further Programs 31

7 Installation 33
7.1 Dependencies 33
7.2 Building & Installing 33
7.3 Build Parameters 34

Bibliography 35

1 Introduction

One outcome of our work on a *Generic Timing Model (GTM)* [11] are *History-Cognisant Utility Functions (HCUFs)*. HCUFs extend the well-known concept of *Time-Utility Functions (TUFs)* [7]. A TUF assigns a certain utility to a job execution depending on its completion time. A HCUF provides a utility value for the execution multiple consecutive jobs and thus allows to rate the behaviour the associated task. Both TUFs and HCUFs can be used for real-time scheduling. They are especially useful if temporal overload situations have to be resolved [7, 12].

The *Timing Models Scheduling Simulation (`tms-sim`)* framework is being developed to assist the evaluation of TUF- and HCUF-based scheduling algorithms. `tms-sim` is written in C++ and provides a number of classes and functionalities that assist the implementation of scheduling simulations. Its core package provides base classes that can be used to implement various task models, schedulers, and utility functions, as well as a container that manages the actual scheduling simulation. For all abstract base classes a number of implementations is provided in `tms-sim`. An XML interface can be used to load/store task sets persistently.

This report is intended as a high-level manual for `tms-sim`. It does not replace the documentation that can be generated from the source code. Instead, it is aimed to provide a general view of the framework and to describe how `tms-sim` can be used and extended. In chapter 2, we describe the core package of `tms-sim` and how simulations are performed. Actual implementations of schedulers, task models, and utility functions are presented in chapter 3. Random generation of task sets is described in chapter 4. The executables that are currently part of `tms-sim` are described in chapter 6. In chapter 5, we describe utility classes that are part `tms-sim` and handle e.g. logging. Installation of `tms-sim` is described in chapter 7.

License Information

The source code of the `tms-sim` framework is available[1] under the conditions of the GNU General Public Licence Version 3[2].

Version Information

This report is an update of an earlier version [9]. Extensions can mainly be found in chapters 4 and 6.

[1] `https://github.com/unia-sik/tms-sim`
[2] `http://www.gnu.org/copyleft/gpl.html`

Acknowledgements

I would like to thank my students Peter Ittner and Markus Neuerburg for their work on and with the `tms-sim` framework. Furthermore, I would like to thank my colleague Christian Bradatsch for his comments on a earlier version of this report [9].

2 Core Package

The core package provides the main functionalities of `tms-sim`. On the one hand, it contains the base classes that are used to represent scheduler, tasks etc., on the other hand, it provides the functionality to execute scheduling simulations. The base classes that represent concepts from scheduling are described in the following section (2.1). The actual simulation is discussed in section 2.2. Section 2.3 introduces some helper classes that can be used to ease the implementation of further evaluations not yet contained in `tms-sim`.

2.1 Scheduling Objects

2.1.1 Job

A `Job` represents an instance of a task that is executed by a scheduler. It is characterised mainly by its execution time and an absolute deadline. These and other relevant attributes are obtained from the task that creates a job. The job class is not intended for inheritance, instead relevant specialisations should be in implementations of the `Task` class. The implementation of `Job` assumes that the execution time is constant and known a priori.

2.1.2 Scheduler

Scheduling and dispatching are performed in implementations of the abstract `Scheduler` class. A simulation (see sect 2.2) uses a scheduler object to execute `Jobs`. Therefore, an implementation of the `Scheduler` class has to provide the following methods:

`Scheduler::enqueueJob(job)` adds a new job that must be executed to the scheduler. You may use this method also to create a schedule as is done by the FPP and EDF scheduler implementations (see sect. 3.1).

`Scheduler::removeJob(job)` tries to remove an active job from the scheduler.

`Scheduler::schedule(now, scheduleStat)` is called by the simulation after all task activations for the current cycle (`now`) have been performed. Use this method especially to check if all jobs can still keep their deadlines. If jobs need to be cancelled, they must be returned to the simulation via the `scheduleStat` parameter.

`Scheduler::dispatch(now, dispatchStat)` dispatches a job and executes it for one time step. If the job finishes in this time step, it must be returned to the simulation through the `dispatchStat` parameter. `dispatchStat` is also used to store information, e.g. about the job currently being executed, whether a deadline miss occurred in the current time step, or if no job could be dispatched and the scheduler thus was idle.

`Scheduler::hasPendingJobs()` returns `true` if there are still jobs enqueued for execution.

`Scheduler::getId()` shall return a human-readable unique ID of the class. This ID is used in several logging outputs.

2.1.3 Task

A `Task` is the abstract representation of some work that shall be performed. Any time the `Task` is activated, it generates a `Job` that must be executed by the scheduler. However, there is no direct interaction between a `Task` and a `Scheduler`. The `Task` class provided in the core package contains only basic attributes (e.g. execution time, deadline, utility functions, etc.). Additionally, this base class comprises a number of attributes and methods to collect statistical data during the simulations. Here, among others, the number of activations and successful completions is counted. The activation policy and generation of new jobs must be provided by subclass implementations.

The basic mechanism for spawning new jobs is already implemented in the `Task` class. Concrete implementations have to provide the following hook functions:

`Task::startHook(now)` is called to initialise the task for execution. The method must return the absolute time when the task will release the first job.

`Task::getNextActivationOffset(now)` is used to determine relative time when the next job of a task is released. It returns a value that is added to the task's last absolute activation time.

`Task::spawnHook(now)` is called when a the time for the next activation of a task expires. It returns a new job that is passed to the scheduler.

`Task::completionHook(job, now)` is called when a job finished execution. This method has to free any memory that is related to the job.

`Task::cancelHook(job)` is called whenever a job was cancelled by the scheduler. Like `Task::completionHook(job)`, this method has to free any memory that is related to the cancelled job

In the current implementation, the release time of a job is determined when the previous job is released. This method does not allow to implement activation dependencies between tasks. We will accommodate for this in future versions of `tms-sim`.

2.1.4 Time Utility Functions

`tms-sim` supports TUFs and HCUFs. TUFs rating single job executions are implemented using the abstract `UtilityCalculator` class:

`UtilityCalculator::calcUtility(job, complTime)` shall return a utility value that is achieved if `job` is completed until `complTime`.

2.1.5 History-Cognisant Utility Functions

The `UtilityAggregator` class represents a HCUF that rates multiple consecutive jobs of a task. When inheriting from this class, the following methods must be implemented:

`UtilityAggregator::getCurrentUtility()` returns the current HCUF value of the associated task.

`UtilityAggregator::predictUitlity(u)` returns the HCUF value that would be achieved, if the TUF value `u` is added to the execution history.

`UtilityAggregator::addHook(u)` is called any time a new TUF value is added to the execution history. Use this method to record TUF values in your implementation.

2.1.6 General Remarks on Tasks and Utility Functions

The class definitions of the abstract `Task`, `UtilityCalculator` and `UtilityAggregator` classes add two interface classes whose methods must be implemented in subclasses. The `ICloneable<class T>` interface defines a `clone()` method that is used for replication of objects.

The `WriteableToXML` class defines methods that are used to serialise objects to an XML file. It shall be used as follows:

`WriteableToXML::getClassElement()` must be overwritten in any subclass implementation. It shall return a subclass-specific element name (e.g. "periodictask").

`WriteableToXML::writeData(writer)` is called by the `WriteableToXML::writeToXML()` method to write an object's data to the XML file. If your subclass (e.g. of `Task`) defines additional fields, you need to overwrite this method. In your implementation, make sure to first call the superclass' `writeToXML()` method. After that, write your class' data to the XML file.

Deserialisation is performed by dedicated builder classes. Create a new builder class that inherits from `IElementBuilder<X>` from the `xmlio` package of `tms-sim`, where `X` is one of `Task`, `UtilityCalculator`, or `UtilityAggregator`. Implement the `IElementBuilder<X>::build()` method that reads the object from XML. Finally, register your builder class in the corresponding factory, i.e. one of the `TaskFactory`, `UtilityCalculatorFactory`, or `UtilityAggregatorFactory` classes.

2.2 Simulation

The `Simulation` class manages the actual scheduling simulation. It executes a task set with a given scheduler for a number of (discrete) time steps.

Figure 2.1 outlines the execution a single simulation (using one task set and one scheduler). In each time step, the `Simulation::doActivations()` method checks each task for new activations. If a task's `nextActivationOffset` time since the last activation has elapsed, the `Task::spawnHook()` method is invoked. The returned job then is enqueued in the `Scheduler`. Then, the `Simulation::doExecutions()` method calls the `Scheduler`. First, the

`Scheduler::schedule()` method is invoked to create and/or check the current scheduler. This method may cancel jobs that e.g. will definitely miss their deadlines. The cancelled jobs are returned via the `scStat` parameter, and the `Simulation` notifies the associated tasks about the cancellation. Second, the `Scheduler::dispatch()` method executes one job for one time step. If a job finished execution, it is returned via the `dispStat` parameter and the associated task is notified.

After the given number of time steps has been simulated, the simulation cleans up by executing all currently active jobs until they are finished (not shown in fig. 2.1). No new jobs are generated during this time. Results of each simulation are returned to the calling program.

2.3 Helper Classes

The core package provides some further classes to ease extensions of `tms-sim`:

Deadlinemonitor keeps a list of active jobs ordered by their deadline. Thus, pending or actual deadline misses can be detected.

Stat holds the overall simulation statistics.

WriteableToXML defines a general interface that must be implemented by any class that should be written to XML files.

ICloneable<class T> defines an interface for classes that can be cloned (used in the task and utility function classes).

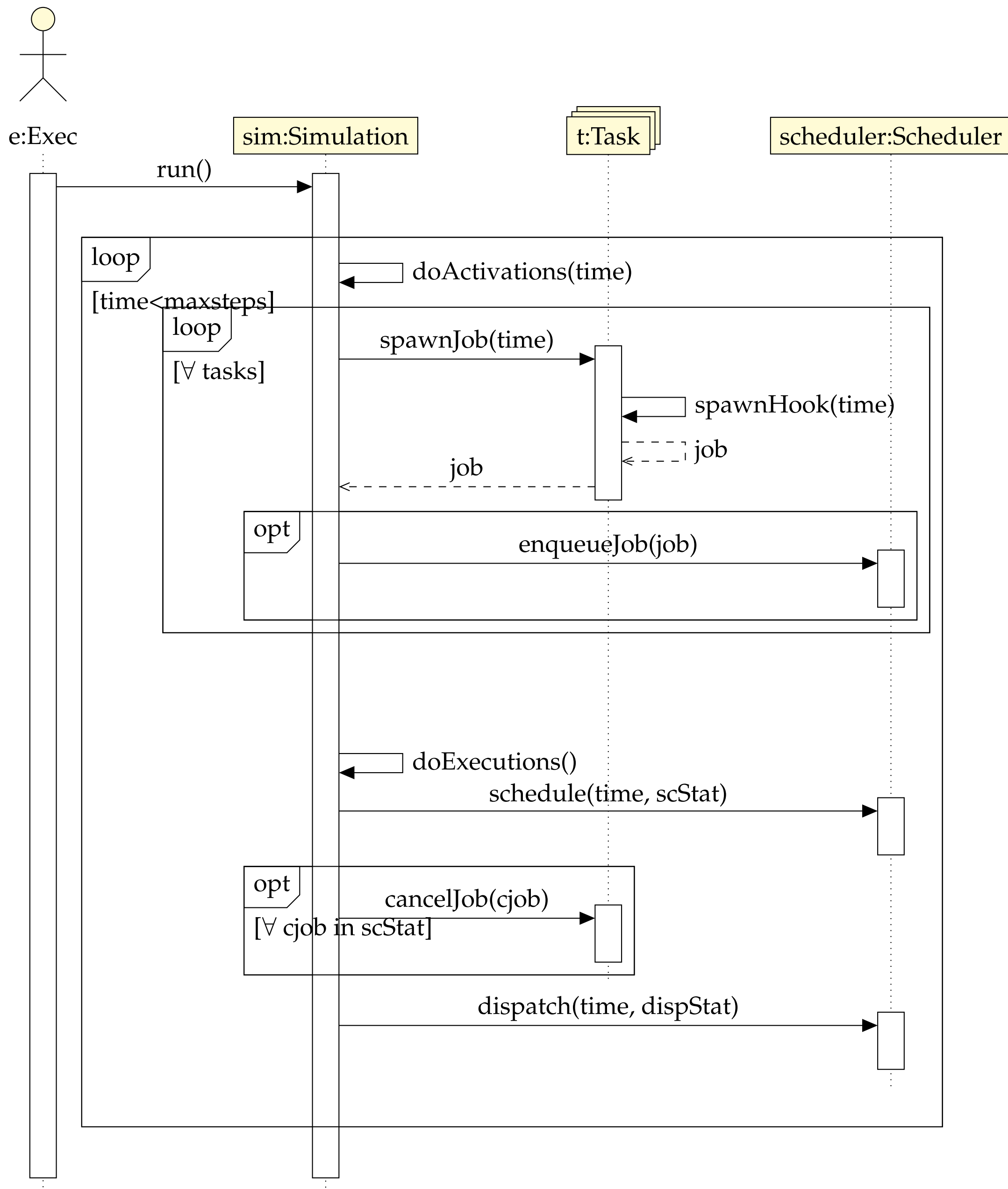

Figure 2.1: Execution of simulation

3 Implementations

3.1 Schedulers

Figure 3.1 gives an overview of the schedulers that are currently implemented in `tms-sim`.

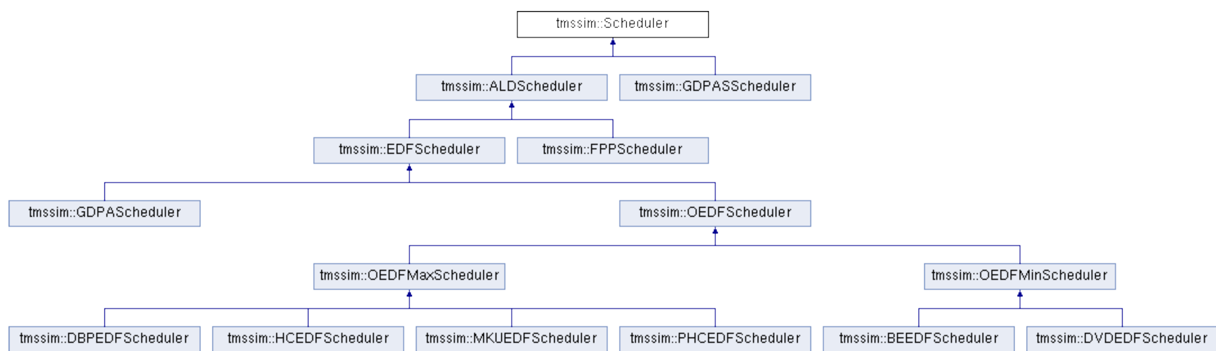

Figure 3.1: Class hierarchy of scheduler implementations

3.1.1 ALD

The *Abstract List Dispatch Scheduler* (`ALDScheduler`) is intended to ease the implementation of schedulers that can hold the execution order of active jobs in a list. The `ALDScheduler` relieves the programmer from having to implement the same dispatching mechanism again and again any time he writes a new scheduler.

When inheriting from `ALDScheduler`, your implementation has to place active jobs in the `ALDScheduler::mySchedule` list. Dispatching is always performed from the front of the list. If your implementation decides to cancel some job, make sure to call the `ALDScheduler::notifyJobRemoved()` method to keep the scheduler's internal state consistent. This is not necessary, if the `ALDScheduler::removeJob()` method was called.

Most current scheduler implementations (except for the GDPA-S scheduler) are derived from this class.

3.1.2 FPP

The `FPPScheduler` class implements the *Fixed Priority Preemptive* policy. Jobs are kept in a list ordered by their priority (lower priority value $\leftrightarrow$ higher priority). The task with the highest priority is dispatched. If a task arrives with the priority higher than the one of the currently executing task, it takes over the processor. `FPPScheduler::schedule` checks whether all active tasks can keep their deadlines. Tasks that will miss their deadline are cancelled.

3.1.3 EDF

The `EDFScheduler` class implements the *Earliest Deadline First* policy [15]. The Jobs are ordered increasingly by their deadlines, and the job with lowest deadline is dispatched. However, if this job is pending to miss its deadline, it is immediately cancelled. The class provides the static `EDFScheduler::checkEDFSchedule()` method to check an EDF schedule for feasibility. This method is intended to be used in subclasses.

3.1.4 OEDF

The OEDFMax/Min schedulers are abstract schedulers. They extend the EDF scheduler by a mechanism to cancel jobs if a overload situations must be resolved. Thereby, the whole schedule of jobs currently active is analysed. Cancellation is performed by maximising/minimising some value for each active job. The jobs with the largest/smallest values are cancelled until the overload situation is resolved. Actual calculation of the value is performed in subclass implementations. Therefor, both the `OEDFMaxScheduler` and `OEDFMinScheduler` provide two methods that must/can be overwritten:

`::calcValue(time, job)` shall return the utility value (e.g. based on a HCUF) of `job`, if it is finished until `time`. This method must be overwritten in any case.

`::isCancelCanditat(value)` allows to apply additional limits for cancellation of jobs. The OEDFMax/Min schedulers simply try to maximise/minimise a value. Using this method, an implementation can impose bounds on the value that prohibit cancellation. The default implementation returns always `true`, meaning any value is acceptable for a job to be cancelled.

OEDFMax Schedulers

The `OEDFMaxScheduler` selects the job(s) to be cancelled based on a maximisation policy. The following implementations of the policy are available in `tms-sim`:

`PHCEDFScheduler` uses the *Pure History-Cognisant* utility value of a task [12]. Tasks with a higher value representing more successful executions in the past have a higher probability for cancellation.

`HCEDFScheduler` works similar to the `PHCEDFScheduler`, but additionally takes the job's remaining execution time into account [12]. This approach is inspired by the work of Aldarmi and Burns [1] on the dynamic value-density approach (see below).

`DBPEDFScheduler` This scheduler is intended to be used for scheduling `DBPTasks` (see 3.2.2). It uses the *distance-based priority* [6] of a task as cancellation criterion.

`MKUEDFScheduler` is used for HCUF-based scheduling of (m, k)-firm real-time tasks as introduced in [13].

OEDFMin Schedulers

If a `OEDFMinScheduler` detects a pending deadline miss, it bases its decision for job cancellation on the minimisation of a value.

`BEEDFScheduler` implements the Best-Effort approach from the original work on time-value scheduling by Jensen et al. [7]. The relevant value is the value-density of the job, i.e. its possible value/time-utility divided by the job's execution time: u/C.

`DVDEDFScheduler` is an optimisation of the best-effort approach. It was proposed by Aldarmi and Burns [1] to heed the time that was already spent for the execution of a job. The dynamic value density of a job is calculated as $u/\bar{C}$ with $\bar{C}$ being the remaining execution time of a job.

3.1.5 GDPA & GDPA-S

The `GDPAScheduler` and `GDPASScheduler` implement the *Guaranteed Dynamic Priority Assignment* policies by Cho et al. [3]. They are used for the scheduling (m,k)-firm real-time tasks. The GDPA Schedulers are aimed at providing a bounded probability of failure. Basically, they work like EDF, but come with additional mechanisms to resolve pending deadline misses while still guaranteeing (m,k)-firm real-time constraints.

3.2 Task Models

`tms-sim` comes with a number of task models that are already implemented. An overview of these is given in figure 3.2.

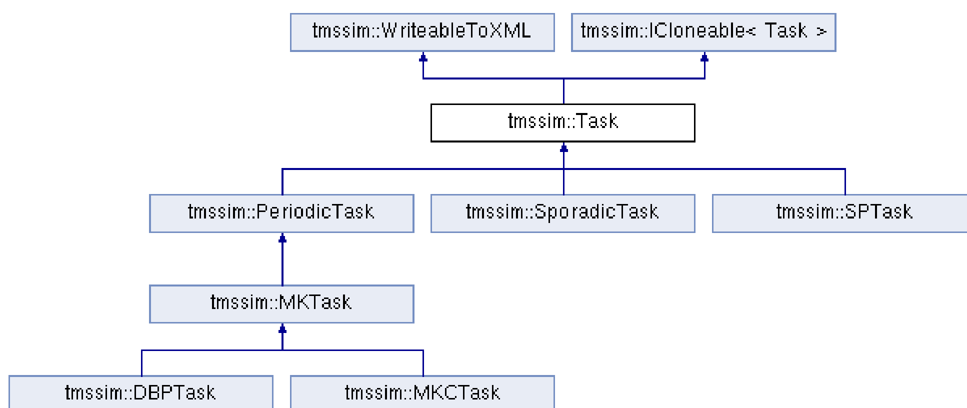

Figure 3.2: Class hierarchy of task implementations

3.2.1 PeriodicTask

The `PeriodicTask` implements the well-known model of a task that is activated in a strictly periodical manner (see e.g. [15]).

3.2.2 MKTask

The `MKTask` and its subclasses are derived from the `PeriodicTask` class. The `MKTask` class represents a periodic task with (m, k)-firm real-time constraints, meaning that at least m instances within k consecutive jobs must be executed successfully. The following specialisations of the `MKTask` are available in `tms-sim`:

`DBPTask` The first work on (m, k)-firm real-time tasks defines a *Distance-Based Priority* assignment [6]. The priority of a task is chosen based on the task's distance from a failing state. The task is in a failing state, if less than m jobs within the k-window were executed successfully.

`MKPTask` implements a static assignment of (m, k)-patterns [17]. The pattern determines which instances of a task are mandatory or optional. Mandatory jobs are scheduled with their original, e.g. rate monotonic priority [15]. Optional jobs are assigned the lowest possible priority and thus are only executed if the system has time to spare. In the original work [17], any first instance of a task is classified as mandatory. If the task set is synchronous, i.e. all task offsets are zero, $t = 0$ is a critical instance. This leads to a rather pessimistic schedulability test. It can be relieved by introducing rotation values that shift the (m, k)-pattern of single tasks [16]. The `MKPTask` is prepared for such pattern rotation, however the actual rotation values must be calculated separately. In `tms-sim` this is performed by the `MKGenerator::calcRotationValues()` method. The method includes the corrections to the original algorithm [16] which are described in [10].

3.2.3 SporadicTask

The `SporadicTask` class implements a task that is activated with a minimum inter-arrival time $t_{\min}$. Actual activations may also happen later. Currently, the following activation policy is implemented: The task is assigned a basic activation probability p_A. In each time step after $t_{\min}$ has elapsed, a random number is generated and evaluated against p_A. Based on the result, either the task is activated, or the process is repeated for the next time step.

3.2.4 SPTask

The `SPTask` class was introduced to generate a certain but very controlled random behaviour. Basically, a `SPTask` behaves like a periodic task, but is only activated with a certain probability p_P. It can be used to introduce sporadic overloads at very specific points in time.

3.3 Utility Functions

3.3.1 Time-Utility Functions – `UtilityCalculator`

At the moment, we provide only a utility calculator for the firm real-time model, the `UCFirmRT` class (see figure 3.3). This calculator yields a utility of 1, if a job finishes until its deadline, and a utility of 0 else.

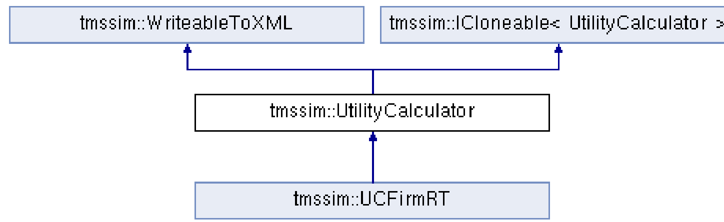

Figure 3.3: Class hierarchy of `UtilityCalculator` implementations

3.3.2 History-Cognisant Utility Functions - `UtilityAggregator`

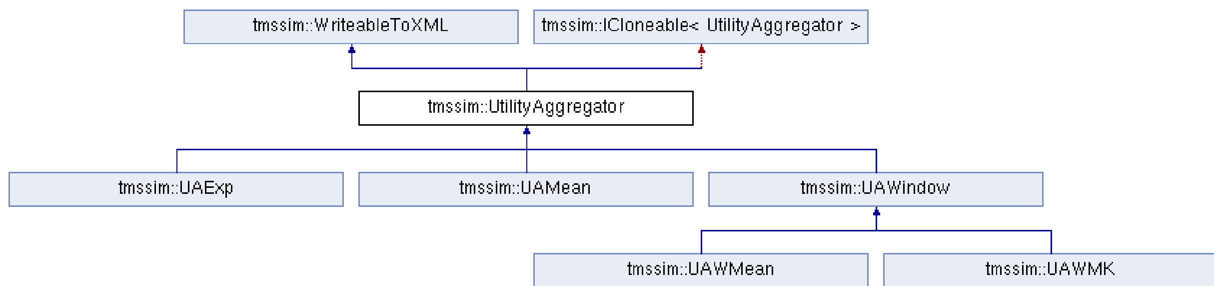

Figure 3.4: Class hierarchy of `UtilityAggregator` implementations

A number of utility aggregators is provided in `tms-sim` (see figure 3.4 for an overview):

`UAExp` aggregates all TUF values that a task collects into a single value U_i. Jobs that date back longer have less influence on the actual value of U_i than those that were performed shortly before. We calculate $U_i(k)$ after the execution of the k-th job as:

$$\begin{aligned} U_i(0) &= 1 \\ U_i(k) &= (1-w)U_i(k-1) + wu_k \end{aligned}$$

w represents some weight $0 < w < 1$ that allows to weigh between the newest utility u_k and the historical evolution of U_i. Newer values have a higher influence on the actual HCUF values.

`UAMean` calculates the mean value of all TUF values that a task experienced. No weighting is performed by this aggregator.

`UAWindow` is an abstract super class for the implementation of utility aggregators that operate on a sliding window of TUF values. This class provides management of the window, actual HCUF calculations must be implemented in subclasses.

`UAWMean` calculates the mean value over a sliding window.

`UAWMK` is an aggregator that was developed to represent (m,k)-firm real-time constraints. It calculates the HCUF value H_m based on the specific (m,k)constraints and the window entries w_i [13]:

$$\hat{H}_m(\tau_i) = \frac{k}{m} H_m(\tau_i) = \frac{1}{m} \sum_{j=1}^{k_i} w_i^j \tag{3.1}$$

4 Task Set Generator

`tms-sim` comprises several techniques to randomly generate task sets. The first generator is the `MkGenerator` that creates independent sets of (m, k)-firm real-time tasks. The `AbstractMkTaskset` (sect. 4.2) is an advancement of the `MkGenerator`. The third generator is the `TsTaskset` (sect. 4.3) that generates task sets with producer and consumer tasks that are used to evaluate communication timing behaviour.

4.1 MKGenerator

The `MKGenerator` class was the first task set generator in `tms-sim`. It generates (m, k)-firm real-time task sets, and it was used for the simulations presented in [13]. An instance of the MKGenerator uses the following fixed parameters for task set generation:

- A seed for the random number generator,
- the size of the task set,
- a period interval $[T_{\min}, T_{\max}]$,
- an interval $[k_{\min}, k_{\max}]$ for the k parameter,
- a target utilisation U_t and the maximum allowed deviation d_U,
- a granularity g_C for the calculation of the task execution times.

Task sets are generated in the following manner: The period of the tasks is randomly selected between $T_{\min}$ and $T_{\max}$. The k value of each task is randomly selected from $[k_{\min}, k_{\max}]$, and the m value from $[1, k]$. The execution times of the tasks then are generated such that the task set's utilisation lies inside the interval $U_t \pm d_u$: For each task τ_i, an execution time weight w_i is randomly selected from $[1, g_c]$. Based on the weights w_i and the target utilisation U, a value C'_i is calculated:

$$C'_i = \frac{U}{\sum_{j=1}^{n} w_j} T_i w_i \qquad (4.1)$$

A task's actual execution time is retrieved by rounding C'_i to the nearest integer value. If the rounding results in 0, $C_i = 1$ is set. This forced rounding up of C_i leads to a slight upward shift of the mean utilisation away from U of the generated task sets within any interval. Task sets with a utilisation outside $U \pm d_u$ are discarded immediately.

Generated task sets are stored in a special `MKTaskSet` container. The `MKTaskSet` comprises some additional information about the task set. For example, the `MKTaskSet::seed` field can be used to re-generate the task set later again. Thus, there is no need to store the whole task set, instead it is sufficient to save the seed value of the task set. Additionally, the generator checks whether a task set passes the sufficient schedulability test for (m, k)-firm real-time task sets by Jia et al. [8] and stores the result in the `MKTaskSet::suffMKSched` field.

4.2 AbstractMkTaskset

For a more detailed analysis, the class `AbstractMkTaskset` was developed. It represents a set of `AbstraktMkTasks` that have no execution time, but only execution time weights (see above). Concrete task sets can be derived for any given processor utilisation. Insofar, the `AbstractMkTaskset` generalises the `MkGenerator` described above.

4.3 Tasksets with Dependencies

Simulation results presented in [2] are based on task sets with producer and consumer tasks. The task set itself is represented by the `TsTaskSet` class. Task parameters T_i, C_i are generated as described above. Additionally, period generation can be restricted using the approach described by Goossens and Macq [5]. The number of producers and consumers must be fixed through a parameter.

The dependencies between m producers and n consumers are represented by a `DependencyMatrix` $D = (d_{i,j}) \in \{0,1\}^{m \times n}$, where each row stands for a producer, and each column for a consumer task. An entry $d_{i,j} = 1$ means that consumer task j reads data from produces task i.

Generation of the matrix is performed row-wise using a parameter `maxRowSum` as additional constraint. The sum of each row i, $s_i = \sum_{j=1}^{n} d_{i,j}$, is chooses the number of connections randomly from an interval $I_R = [1, \texttt{maxRowSum}]$. Now the problem is to distribute s_i 1s over n places. The number of possible (distinct) assignments is given by the binomial coefficient $\binom{n}{s_i}$. To find an actual assignment, `tms-sim` implements a recursive algorithm that enumerates all possible assignments deterministically. Thus, each assignment can be identified by a number $a \in [1, \binom{n}{s_i}]$. a is chosen randomly from the interval, and using the recursive algorithm, the actual assignment for the row is determined.

Algorithm 4.1: The `getAssignment` algorithm

```
1 Function getAssignment(n, s_i, a)
2     A ← (0,0,...,0);                    /* initialise bitfield with 0 */
3     c ← a;
4     fillBuckets(A, n, 0, s_i, c);
5     return A;
```

The algorithm works as follows: The `getAssignment` function (algorithm 4.1) shall return a bit field A of size n, with s_i entries being 1. These entries shall be assigned according to the number a. Actual assignment is performed by the `fillBuckets` function (algorithm 4.2). The parameters A and c are passed by reference, as they must be manipulated during recursion. The `fillBuckets` function takes the following parameters: A is a reference to the bitfield that shall contain the final assignment (indexing starts with 0), n is the size of the bitfield, o denotes the offset from which on filling should start, k is the number of remaining 1s that must be distributed, and c references the counter variable that identifies the assignment. The function implements a depth-first search of possible assignments, until the s_ith assignment is found. Each instance of `fillBuckets` tries assignments of a 1 to the places $i \in [o, n-k]$ of A (line 3).

For each assignment, it then calls itself recursively to assign the remaining $k-1$ 1s to places $i+1$ to $n-1$ (line 5). If an instance has placed the last 1 in A, a full assignment has been found. Depending on the counter value c, different actions are taken: If $c > 0$, c is decreased by 1 (line 10), meaning that further assignments have to be checked. Else, if $c = 0$, the final assignment has been found, which is signalled to parent instances of `fillBuckets` by setting $c = -2$ (line 12), such that these terminate the recursion too.

Algorithm 4.2: The `fillBuckets` algorithm

```
1  Procedure fillBuckets(&A, n, o, k, &c)
2      for i = o to n − k do
3          A_i ← 1;
4          if k > 1 then
5              fillBuckets(A, n, i+1, k - 1, c);
6              if c = −2 then
7                  return;
8          else
9              if c > 0 then
10                 c ← c − 1;
11             else if c = 0 then
12                 c ← −2;
13                 return;
14         A_i ← 0;
```

5 Utility Classes

The utils package of tms-sim contains some general classes to support the implementation of tms-sim, though they are not tms-sim-specific. At the moment, these are two kinds of loggers and support for key/value files.

5.1 Logging

tms-sim comes with two types of loggers: The TLogger classes are intended for logging of implementation-related information. They support five log levels from TLL_NONE without any output to TLL_DEBUG with very fine-grained information. The log level can either be set globally using the CPPFLAGS of the compiler, or by defining the level specifically in each .cpp file before the inclusion of <utils/tlogger.h>.

The Logger class should be used to output information about actual job execution. It supports a number of log levels only limited by sizeof(int). A log output is only written to clog, if the output's log level is smaller than the global log level.

5.2 Key/Value Files

The KVFile class can be used to parse data from files written in the typical KEY=VALUE style. Lines starting with # are ignored. KVFile implements a std::map<std::string,std::string> with some additional functions for reading the input file and retrieving specific primitive data types like integers from a key/value pair.

6 Executables

The `tms-sim` package comes along with several executables that can be used for some basic evaluations.

6.1 TMS-Sim

This tool is used to simulate of a single task set (read from XML) with multiple schedulers. At the end of the simulation, a number of statistical data is output that allows to compare the overall performance of the schedulers with respect to the task set. Depending on the verbosity level, additional runtime information is displayed that allows for examining the behaviour of the schedulers in more detail. Command line parameters are described in table 6.1.

Table 6.1: Command line parameters for `tms-sim`

Parameter	**Description**
`-i xmlfile`	Read task set from `xmlfile`
`-n steps`	Execute simulations for `steps` time steps

`-s SCHEDULERS`	Simulate with the schedulers specified by the `SCHEDULERS` string.
	Valid entries are:
	`b:BEEDFScheduler`
	`d:DVDEDFScheduler`
	`e:EDFScheduler`
	`f:FPPScheduler`
	`g:GDPAScheduler`
	`h:HCEDFScheduler`
	`m:MKUEDFScheduler`
	`p:PHCEDFScheduler`
	`s:GDPASSScheduler`

`-v[v[v[v]]]`	set verbosity level
`-help`	Print a short help text and exit

6.2 MKEval

The `mkeval` tool is intended for an automated simulation of large number of (m,k)-firm real-time task sets. The used schedulers and task models are currently hard-coded in the executable. An overview of the schedulers and task models is provided in table 6.2. `mkeval` is designed to run multiple simulations in parallel to speed up the overall simulation. Basic settings for

task set generation are provided via a configuration file. This file is organised in the typical `key = value` style. The following parameters for the generator must be provided in the file: `minPeriod, maxPeriod, minK, maxK, maxWC`. Command line parameters accepted by `mkeval` are described in table 6.3. So far, all parameters are mandatory.

Table 6.2: Task models and schedulers used in the experimental evaluation

Model	**Abbr.**	**Scheduler**	**Reference**
Distance-based priority	DBP	FPP	[6]
(m, k)-firm control tasks	MKP	FPP	[17]
MKP with pattern rotation	MKP-R	FPP	[16]
Guaranteed Dynamic Priority Assignment	GDPA	EDF (modified)	[3]
Guaranteed Dynamic Priority Assignment-S	GDPA-S	EDF (modified)	[3]
Utility-based (m, k)-tasks	MKU	OEDFmax	sect. 3.1.4
OEDFmax with DBP	DMU	OEDFmax	sect. 3.1.4

Table 6.3: Command line parameters for `mkeval`

Parameter	**Description**
`-c cfg_file`	Read task set generation parameters from `cfg_file`
`-u U`	Generate task sets with a target utilisation `U`
`-d dU`	Allow a maximum deviation of `dU` from the target utilisation
`-s seed`	Use `seed` as seed for task set generation
`-t size`	Specifies the `size` of each generated task set
`-T task sets`	Specifies how many `task sets` are generated
`-n steps`	Execute simulation for `steps` time steps
`-m nthreads`	Simulate using `nthreads` threads
`-p prefix`	Set the `prefix` of the log files
`-help`	Print a short help text and exit

During simulation, `mkeval` generates a number of log files and outputs. Logfiles are written for each scheduler. Their name is composed from the log file prefix (see table 6.3) and the scheduler abbreviation (see table 6.2). Each line in such a file contains the overall statistical data for one task set identified by the seed at the beginning of the line. Additionally, a `$PREFIX-success.log` file is created that records for each task set which schedulers have successfully executed the task set. Some additional information is written to `stdout`. The `==RSB==` lines contain information, how many task sets were successful with a certain combination of schedulers. `==STAT==` lines show the overall execution statistics for the different schedulers. `==SUCC==` lines give the same statistics only for task sets that were executed successfully with all schedulers.

6.3 RunTS

The outputs of the `mkeval` tool give a very high-level view about the behaviour of the task sets. Sometimes, one might be interested in examining a single task set in more detail. This can be achieved by using the `runts` tool. `runts` simulates a single (m, k)-firm real-time task set with a number of schedulers. Depending on the chosen verbosity level, more or less detailed runtime information is output. Command line parameters of `runts` are documented in table 6.4.

Table 6.4: Command line parameters for `runts`

Parameter	Description
`-c cfg_file`	Read task set generation parameters from `cfg_file`
`-u U`	Generate task sets with a target utilisation `U`
`-d dU`	Allow a maximum deviation of `dU` from the target utilisation
`-s seed`	Use `seed` as seed for task set generation – use the seed from the `mkeval` logfile
`-t size`	Specifies the `size` of the generated task set
`-n steps`	Execute simulation for `steps` time steps
`-p prefix`	Set the `prefix` of the log files
`-v[v[v]]`	set verbosity level (optional)
`-help`	Print a short help text and exit

6.4 tseval

The `tseval` program was used to evaluate the communication timing and data age between producer and consumer tasks under different communication semantics [2]. Command line parameters are documented in table 6.5.

Table 6.5: Command line parameters for `tseval`

Parameter	Description
`-c cfg_file`	Key/value-file with settings for the task set generator (mandatory)
`-s seed`	Use `seed` as seed for task set generation [default=time(NULL)]
`-m simulationthreads`	number of threads that are used for simulation
`-S`	run single simulation and output to stdout
`-T N`	Simulate `n` tasksets
`-p prefix`	Set the `prefix` of the log files

`tseval` implements the following three semantics for communication between tasks: (1) In the *bounded execution time (BET)* model, data from a producer can be used as soon as the producer task has finished execution. (2) With *logical execution time (LET)*, data is logically written and read only at task period boundaries. (3) The LET-RTA uses a *response time analysis* to prepone output times but still keep the predictable behaviour of LET. More on the approaches can be found in [2].

6.5 MkDSE

`mkdse` implements a design space exploration for (m,k)-firm real-time task sets. Its basic idea is to derive multiple concrete task sets with increasing processor utilisation from a single abstract task set (that only has execution time weights, but no actual execution time). Each concrete task set is simulated with a number of schedulers. Thus, characteristics of an abstract task sets can be evaluated, especially its breakdown utilisation [14]. Also, the performance of the different schedulers can be compared. Further uses of `mkdse` include the search for breakdown anomalies.

Table 6.6: Command line parameters for `tseval`

Parameter	**Description**
`-c cfg_file`	Read task set generation parameters from `cfg_file`
`-e ecfg_file`	Fined-grained configuration of execution
`-S seed`	Use `seed` as seed for task set generation – use the seed from the `mkeval` logfile
`-seed-file file`	read seeds from file (one per line; will ignore `-T` parameter)
`-t size`	Specifies the `size` of the generated task set
`-T task sets`	Specifies how many `task sets` are generated
`-u U`	Generate task sets with a target utilisation `U`
`-d dU`	Allow a maximum deviation of `dU` from the target utilisation
`-n steps`	Execute simulation for `steps` time steps
`-m nthreads`	Simulate using `nthreads` threads
`-a sched`	Scheduler/Task allocators, simulate scheduler `sched`; may be specified multiple times; for a list of valid schedulers, see thet actual `-help` output.
`-p prefix`	Set the `prefix` of the log files
`-x prefix`	Write successful tasksets to xml file; default prefix is log prefix
`-l log`	Activate execution logging for class `log`; for a list of valid options, see thet actual `-help` output.
`-restrict-periods`	Use period generator by Goossens and Macq [5]
`-help`	Print a short help text and exit

`mkdse` uses several helper classes to organise simulations as efficiently as possible. For each `AbstractMkTaskset`, all associated concrete task sets are created and held by a MkDseSimulationSet. The concrete tasksets themselves allocate and manage the actual simulation objects (`MkSimulation`). In this binary, only schedulability of task sets is evaluated. For some schedulers/taskmodels, no actual simulations are performed, if a more efficient schedulability test (implemented in the `MkpSimulation` class) yields a positive result. Actual simulations are performed by the `GstSimulation` class which implements Goossens' schedulability test [4] for (m,k)-firm real-time tasks.

6.6 Tms-Vis

The `tms-vis` tool provides a simple Qt GUI to visualise and compare actual task schedules. It requires log files from the simulation of single task sets as input. The simulations must be run with the log options `-l SIM` and `-l EXEC` activated. Further options may be used, their corresponding output is ignored by `tms-vis`.

6.7 Further Programs

genseeds can be used to generate one or more seed files, e.g. for `mkdse`.

mkbdexcsearch old search for breakdown utilisation.

mkcexcrun run (m, k)-firm real-time task set with fixed (m, k)-patterns and exact schedulability test.

mkcs-check checks the calculation of spin parameters for MKP-S.

mkexcrun old simulation with exact schedulability test, deprecated.

mkextwrite writes a single taskset from an extended mkeval (`AbstractMkTaskset`) to an XML file.

mkrun simulates a single (m, k)-firm task set (given by generation parameters).

mkrun-xml simulates a single (m, k)-firm task set from XML file.

mktsprint converts a (m, k)-firm task set from XML to a more readable format.

mkwrite writes a (m, k)-firm task set specified by its generation parameters to XML. This program should be used with care, as its output is different from the task sets that are generated by `mkrun` for the same parameters.

prseedsearch find seeds for good `AbstractMkTasksets`, such that the first generated task set meets the maximum allowed utilisation deviation.

tsrun executes single producer/consumer task set from XML.

tswrite writes a single producer/consumer task set to XML.

7 Installation

7.1 Dependencies

`tms-sim` has the following dependencies on other packages:

`dev-libs/libxml2` for de-/serialisation of task sets (mandatory).

`>=dev-util/cmake-2.8.12` for building (`tms-sim` requires the `OBJECT` option for the `add_library` command).

`>=sys-devel/gcc-4.7` mandatory for `-std=gnu++11`

`>=sys-devel/gcc-4.8.1` (optional) only required if you need to track bugs in the memory management using the `-fsanitize=address` compiler option.

`app-doc/doxygen` (optional) required if you want to generate the HTML documentation from the source code.

Multithreading support e.g. `pthreads`, for building the `mkeval` binary.

7.2 Building & Installing

Currently, we recommend to install and run `tms-sim` only from a local, non system-wide installation. To build `tms-sim`, perform the following steps:

1. retrieve the code from `https://github.com/unia-sik/tms-sim`
2. Change to the default build directory

   ```
   # cd ${BASE}/tms-sim/build}
   ```

3. Configure the build system. For installation into `${BASE}/tms-sim/build` you can use:

   ```
   # ../build-local.sh ..
   ```

   If you also want to build the API documentation, use:

   ```
   # ../build-local-with-doc.sh ..
   ```

   Otherwise, directly call `cmake` with any additional options (see sect. 7.3):

   ```
   # cmake [options] ..
   ```

The default installation path is `/usr/local/`.

If you want to build the visualisation GUI, specify `-DBUILD_GUI=1` (the variable must be set) on your call to `cmake` or the build scripts.

4. Build:

```
# make [-j<threads>]
```

5. Install the binaries.

```
# make install
```

You may skip this step and directly run the binaries from the directories where `make` puts them.

7.3 Build Parameters

The build process can be controlled with the following parameters to `cmake`:

-DCMAKE_INSTALL_PREFIX=<*prefix*> Install to directory <*prefix*>. The `build-local.sh` script uses `./` as default target.

-DBUILD_DOCUMENTATION=on Build HTML documentation for the source. The documentation will be installed to <*prefix*>`/share/doc/tms-sim/`.

Bibliography

[1] S. A. Aldarmi and A. Burns. Dynamic value-density for scheduling real-time systems. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems, 1999.*, pages 270–277, 1999.

[2] C. Bradatsch, F. Kluge, and T. Ungerer. Data age diminution in the logical execution time model. In *29th GI/ITG International Conference on Architecture of Computing Systems (ARCS 2016), Nuremberg, Germany, April 4-7, 2016, Proceedings*, Apr. 2016.

[3] H. Cho, Y. Chung, and D. Park. Guaranteed dynamic priority assignment scheme for streams with (m, k)-firm deadlines. *ETRI Journal*, 32(3):500–502, June 2010.

[4] J. Goossens. (m, k)-firm constraints and dbp scheduling: Impact of the initial k-sequence and exact feasibility test. In *16th International Conference on Real-Time and Network Systems (RTNS'08)*, pages 61–66, Oct. 2008.

[5] J. Goossens and C. Macq. Limitation of the hyper-period in real-time periodic task set generation. In *Proceedings of the 9th International Conference on Real-Time Systems (RTS'01)*, pages 133–148, Mar. 2001.

[6] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m, k)-firm deadlines. *IEEE Transactions on Computers*, 44(12):1443–1451, 1995.

[7] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *6th Real-Time Systems Symposium (RTSS '85), December 3-6, 1985, San Diego, California, USA*, pages 112–122, Dec. 1985.

[8] N. Jia, Y.-Q. Song, and F. Simonot-Lion. Task Handler Based on (m,k)-firm Constraint Model for Managing a Set of Real-Time Controllers. In N. Navet, F. Simonot-Lion, and I. Puaut, editors, *15th International Conference on Real-Time and Network Systems - RTNS 2007*, pages 183–194, Nancy, France, 2007. URL : http://rtns07.irisa.fr.

[9] F. Kluge. `tms-sim` – timing models scheduling simulation framework – release 2014-12. Technical Report 2014-07, Department of Computer Science, University of Augsburg, Dec. 2014.

[10] F. Kluge. Notes on the generation of spin-values for fixed (m, k)-patterns. Technical Report 2016-01, Department of Computer Science, University of Augsburg, Jan. 2016.

[11] F. Kluge, M. Gerdes, F. Haas, and T. Ungerer. A generic timing model for cyber-physical systems. In *Workshop Reconciling Performance and Predictability (RePP'14)*, Grenoble, France, Apr. 2014.

[12] F. Kluge, F. Haas, M. Gerdes, and T. Ungerer. History-cognisant time-utility-functions for scheduling overloaded real-time control systems. In *Proceedings of 7th Junior Researcher Workshop on Real-Time Computing (JRWRTC 2013)*, Sophia Antipolis, France, October 16, 2013, Oct. 2013.

[13] F. Kluge, M. Neuerburg, and T. Ungerer. Utility-based scheduling of (m, k)-firm real-time task sets. In *Accepted at 28th GI/ITG International Conference on Architecture of Computing Systems (ARCS 2015)*, Mar. 2015.

[14] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *10th Real Time Systems Symposium (RTSS 1989)*, pages 166 –171, Dec. 1989.

[15] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[16] G. Quan and X. Hu. Enhanced fixed-priority scheduling with (m,k)-firm guarantee. In *The 21st IEEE Real-Time Systems Symposium, 2000. Proceedings.*, pages 79–88, 2000.

[17] P. Ramanathan. Overload management in real-time control applications using (m, k)-firm guarantee. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):549–559, 1999.