DATooR:
A DAG Design and Analysis Tool

Morteza Mohaqeqi
Wang Yi
Uppsala University

February 2022

DATooR is a Design and Analysis Tool for Real-time systems that are specified as a set of reoccurring DAG tasks deployed on a heterogeneous multiprocessor platform. First, it allows designers to evaluate – by simulation and analytical methods – different scheduling policies using user-specified system configuration parameters such as the number of processor cores, memory access latency, and task parameters etc., and randomly generated task sets. Second (under development), for a system design with a fixed hardware configuration, a task set and a scheduling policy, the tool will generate a run-time schedule providing performance and real-time guarantees.

Contents

Chapter 1

Introduction

Today, parallel architectures such as multi- and many-core processors have become ubiquitous in computing. In embedded domains, there is an increasing trend towards the usage of heterogeneous and parallel architectures for performance-demanding and real-time applications. Paradoxically, the introduction of performance enhancing architectural solutions, such as memory hierarchies, heterogeneous processor cores of different processing capabilities and multi-threading introduce a large degree of uncertainty and make it extremely hard to provide performance and real-time guarantees. This brought a significant challenge (and also a great opportunity) for embedded systems designers to explore the hardware parallelism.

1.1 Applications Areas

This tool is designed (and under development) for intended applications in safety-critical domains where high-performance and real-time requirements must be ensured. Typical application areas include automotive systems involving self-driving and 5G/6G networks, that are computationally demanding real-time systems deployed on heterogeneous multi-core and many-core platforms.

The tool offers two features. First, it allows the designers to simulate the timing behavior and performance of a system with (a large number of randomly generated) possible configurations of hardware and software components, and scheduling policies. The goal is to select and validate the potentially best system configuration and run-time scheduling policy by simulating a large number of randomly generated system configurations. Second, for a given system configuration (a design), it shall provide a run-time schedule, with real-time and performance guarantees such as worst-case response times and throughput.[1]

The tool uses directed acyclic graphs (DAG) to describe software components. The inherent parallelism of embedded software can be truly modeled using the branching structures of DAG's. A DAG [12] is a directed graph where nodes represent parts of a program and edges represent precedence constraints. In summary, DATooR provides the following utilities with respects to timing analysis of DAG tasks running on multicore systems.

- Simulating the execution of a set of periodic real-time DAGs on a specified hardware platform,
- Analytical computation of an upper-bound on the worst-case response-time of the tasks.

[1]In the current version, the configurations are specified by the user; then the tool evaluates the selected policy over a number of randomly generated task sets.

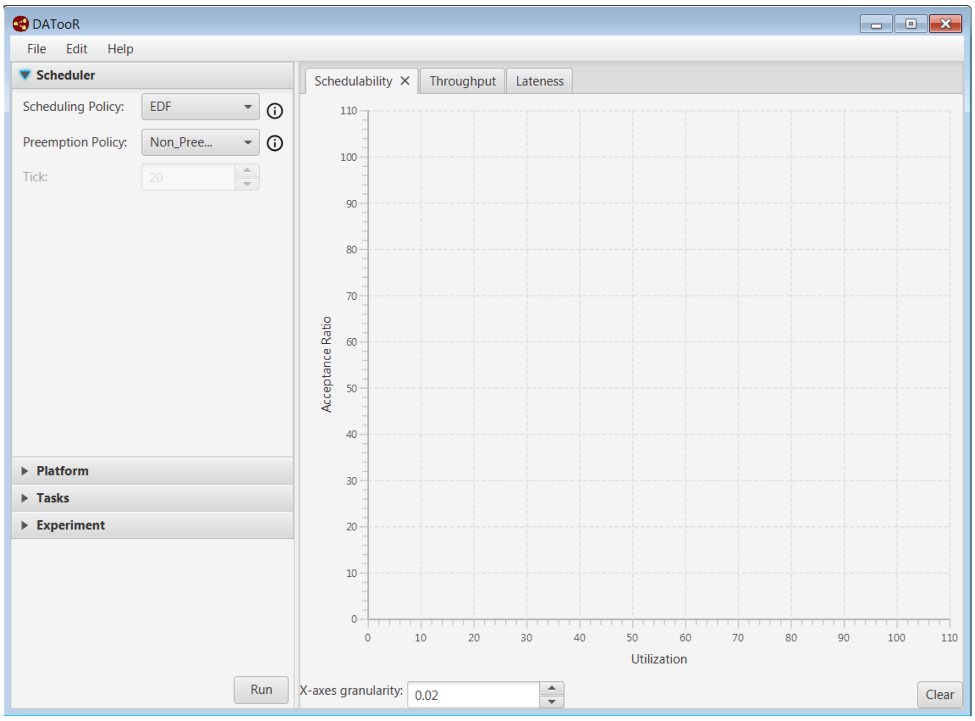


Figure 1.1: A snapshot of the tool.

A user can specify the target hardware platforms with configuration parameters such as number and type of processor cores, as well as memory overheads. In addition, the mapping between software components onto hardware resources as well as the intended run-time scheduling policy among a set of pre-specified policies can be specified.

1.2 An Overview of the Tool

Figure 1.1 shows the main window of the tool, which contains two main parts: configuration (the left-hand side), and results (the right-hand side). In the configuration part, the user can specify software and hardware configuration for which the timing analysis (simulation or analytical) is to be done. The details of this part is presented in Chapter 5.

The result of the analysis is obtained in terms of performance and time-related measures that are provided to the user through a number of charts. The formal definition of the calculated measures as well as the meaning of the presented charts is given in Chapter 6.

Before describing the tool components, the related preliminaries are reviewed. Specifically, the assumed system model is specified in Chapter 2. The supported scheduling algorithms are described in Chapter 3. The methods for random task set generation are specified in Chapter 4.

1.3 How to Run

The tool is self-contained, and no additional software/package is required to be installed. To run the tool,

- in Windows, one needs to double click on the executable file,
- in Linux, one needs to run the file from terminal.

Chapter 2

System Models

System model consists of two parts: (1) task model, and (2) processor model.

2.1 Task Model

DATooR adopts the reoccurring task model where each task is represented by a period and a directed acyclic graph (DAG) of nodes and directed edges. We use the following notations.

Predecessor/Successor: Consider an edge in a DAG from a node u to a node v. Then, u is a *predecessor* of v, and v is a *successor* of u. Nodes with no successor are called *sink*.

Critical Path: The critical path of a node u, denoted by $cp(u)$, is the longest path from u to a sink node [8]. Formally,

$$cp(u) \doteq \begin{cases} wcet(u), & \text{if } u \text{ is a sink,} \\ wcet(u) + \max\{cp(v) | v \in Succ(u)\}, & \text{otherwise.} \end{cases}$$

Here, $wcet(u)$ and $Succ(u)$ are the worst-case execution time and successors of u, respectively.

Example 1. *Figure 2.1 shows a DAG with seven nodes. The number next to each node is its WCET. The node* `J_0` *has three successors:* `J_1`*,* `J_4`*, and* `J_6`*. The DAG has two sink nodes:* `J_3` *and* `J_5`*. The longest path from* `J_1` *to a sink node is* `J_1` $\rightarrow$ `J_5` $\rightarrow$ `J_3`*, which implies* $cp(\texttt{J\_1}) = wcet(\texttt{J\_1}) + wcet(\texttt{J\_5}) + wcet(\texttt{J\_3}) = 50$.

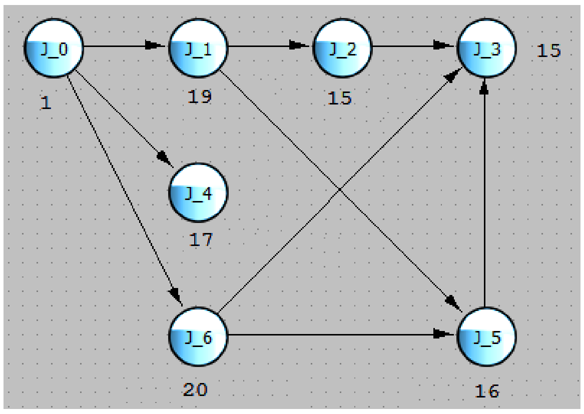


Figure 2.1: A sample DAG structure.

A sporadic release model is assumed where each task releases instances with a minimum inter-release separation time. In addition, tasks have implicit deadline, that is, for each task: deadline = period.

The actual execution time of a node in a task instance is between 0 and its WCET. Each task instance has an *absolute* deadline which is computed by adding the task's (relative) deadline to the release time of that instance.

For each node of a task instance its *laxity* is defined as below.

Laxity: Let d be the absolute deadline of a task instance. The laxity of a node u of the task at time t is

$$laxity(u) \doteq d - (t + cp(u)), \tag{2.1}$$

where $cp(u)$ is the critical path of u defined above.

2.2 Processor Model

In the current version, a homogeneous multicore platform is assumed. The overheads related to memory access can be expressed in terms of three parameters, i.e., memory time (for initial data loading), preemption time (when a preemtive scheduling is used), and communication time (for inter-core migration). The details are provided in Sec. 5.3.1.

Chapter 3

Scheduling Algorithms

Real-time scheduling for multiprocessors (in particular heterogeneous platforms) providing performance- and real-time guarantees is probably one of the most difficult challenges in embedded systems design. There are various scheduling algorithms proposed in the literature for single processors. These algorithms may be directly adopted to the multiprocessor setting. Unfortunately these algorithms perform often badly with low resource utilization; many of them suffer also from anomalies due to the uncertainty introduced by complex hardware features and software components deployed.

3.1 Preemption Strategies

To fulfill timing or QoS requirements, running tasks or nodes may be preempted by newly released instances. Preemption may be (dis-)allowed at node- or task-level. Here only node-level preemption shall be considered.

Non-preemptive scheduling: A running node should run until it is completed. One may consider the case of task-level non-preemption: when a node is completed, only nodes from the same task can be executed.

Preemptive scheduling: A running node can be preempted at any time by any eligible node. One may consider the case of task-level preemption: a node can be preempted by only nodes from the same task.

Restricted preemptive scheduling: It is preemptive scheduling, but preemption is allowed to take place only at predefined time points. A special case is tick-based scheduling where the time line is divided into ticks or time slots. Preemption is allowed to take place only by the end of a time slot.

3.2 Scheduling Policies

At run time, whenever there is a processor core available, eligible nodes will be selected according to their priorities. The priority of a node can be assigned statically or dynamically at node- or task-level according to static task parameters or parameters representing the current status of task executions.

Selecting and prioritizing the nodes for execution is accomplished by the *scheduler*. The scheduler considers *eligible* nodes to dispatch. Formally, at a time instant t, a node is *eligible* if by t (1) all of its predecessors have been finished, and (2) it is not finished.

There are a number of basic scheduling approaches that are used (directly or a modified version) in the tool.

EDF: Earliest-Deadline First assigns the highest priority to nodes with the earliest (absolute) deadline.

RM: Rate-Monotonic assigns the highest priority to nodes of the tasks with the shorter period.

LLF: Least-Laxity First (LLF) prioritizes eligible nodes according to their laxity. Laxity of a node in a DAG is the difference between the deadline and the longest path from that node to any sink node [8], as defined in Eq. 2.1.

FIFO: In this approach, nodes are prioritized based on the arrival time (of the corresponding task instance). Earlier arrival means higher priority.

Federated: In the Federated scheduling approach [6], tasks are partitioned based on the minimum number of required cores (e.g., m). For tasks with $m > 1$, exactly m cores are exclusively dedicated. Other tasks (i.e., those with $m \leq 1$) can share cores to execute.

Partitioned: The nodes are accommodated to the cores so that each node is always run on the same core. Assigning the nodes to the cores can be done according to different criteria. The work in [2] presents some heuristics for that.

Chapter 4

Random Task Generation

In order to assess a timing/schedulability analysis approach, a set of randomly generated task sets are used. To generate a random (DAG) task, one needs to determine the graph structure, the WCET of each node, and the period.

4.1 Random Graph Generation

Graph generation methods mainly specify a method to create graph structures. In DATooR a number of well-known graph-generation methods have been implemented.

4.1.1 Layer-by-layer

In the *layer-by-layer* approach [3], the graph nodes are placed into some layers and then edges are added between layers. Formally, the set of n nodes are first randomly distributed among k layers. Then, for any pair of nodes a and b with $layer(a) < layer(b)$, an edge is added from a to b with the probability p_{edge}.

Algorithm 1 layer_by_layer [3]

Input: n: number of nodes, k: number of layers, p_{edge}: edge probability.

```
1: for $i$ = 1 ... $n$ do
2:     $layer(i) \leftarrow \mathsf{unif}(1, k)$
3: end for
4: for $i$ = 1 ... $n$ do
5:     for $j$ = 1 ... $n$ do
6:         if $layer(i) < layer(j)$ then
7:             if $\mathsf{unif}() < p_{edge}$ then
8:                 Add an edge from $i$ to $j$
9:             end if
10:        end if
11:    end for
12: end for
```

$\mathsf{unif}()$ returns a uniformly distributed random real value between 0 and 1. As well, $\mathsf{unif}(i_1, i_2)$ returns a uniformly distributed random *integer* between (and including) i_1 and i_2.

4.1.2 Series-parallel graphs

This method comprises a recursive procedure. The procedure gets a source and a sink node, as well as the number of outgoing branches from the source node. For each branch, either a single node is inserted between source and sink, or the procedure is recursively called to insert a new subgraph between source and sink.

This method properly considers fork and join structures, and provides an appropriate model to specify many programming constructs in embedded software [10]. The generated graph structure can be tuned through a number of parameters:

- n_{par}: the maximum branching degree,
- p_{par}: the branching probability,
- p_{edge}: the probably of adding an edge. (see Algorithm 4).

The pseudo-code of the process is seen in Algorithm 2.

Algorithm 2 series-parallel

```
1: Create two nodes $src$ and $sink$
2: $depth(src) \leftarrow n_{depth}$
3: $depth(sink) \leftarrow -n_{depth}$
4: expand$(src, sink, n_{depth} - 1, \text{unif}(2, n_{par}))$
5: addEdges()
```

Algorithm 3 expand

```
Input: $src, sink$: source and sink nodes, $dep$: current depth, $n_{br}$: number of branches.
1: for $i$ = 1 $\ldots n_{br}$ do
2:     if $dep = 0$ or unif() $< p_{par}$ then
3:         Create a new node $v$ with $depth(v) = dep$
4:         Add edges $(src, v)$ and $(v, sink)$
5:     else
6:         Create new nodes $v_1, v_2$ with $depth(v_1) = dep$ and $depth(v_2) = -dep$
7:         Add edges $(src, v_1)$ and $(v_2, sink)$
8:         expand$(v_1, v_2, dep - 1, \text{unif}(2, n_{par}))$
9:     end if
10: end for
```

Algorithm 4 addEdges()

```
1: {$n$: num. of ndoes}
2: for all $1 \leq i, j \leq n$ do
3:     if there is no edge from $i$ to $j$ then
4:         if $depth(i) < depth(j)$ and unif() $< p_{edge}$ then
5:             Add edge $(i, j)$
6:         end if
7:     end if
8: end for
```

4.1.3 The STR2RTS benchmark

The STR2RTS benchmark [11] contains a number of benchmark programs from streaming applications. Each program is described as a DAG and the WCET of each node.

4.2 Selecting Periods

In designing the tool, a problematic approach have been taken. The focus is on two application areas: 5G networks [7] and automotive systems designed according to the AUTOSAR reference model AUTOSAR [5] (even though other task models with arbitrary task parameters and release patterns will also be considered). These systems are typical real-time applications but restrict the release and execution patterns of software components to harmonic periodic tasks. For 5G networks, tasks may be released and activated according to their TTI's, that is, the task periods. According to the 5G protocol, the TTIs in a task system must be $0.125ms$ or a multiple of $0.125ms$ up to $1ms$. Similarly, a task period in AUTOSAR must also be a multiple of a basic task period specified. In this way, the hyper-period of a task system is reduced significantly to the scale, allowing for the construction of a feasible schedule by simulation. The set of periods in each case is given in the following table.

Application	**Period Values (milliseconds)**
5G	0.125, 0.25, 0.5, 1
AUTOSAR	1, 2, 5, 10, 20, 50, 100, 200, 1000

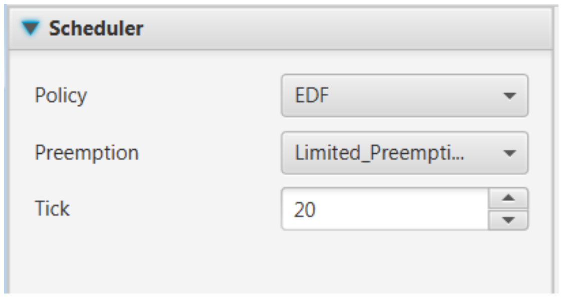
Chapter 5

Tool Manual: Configuration

The user can specify the desired parameters (i.e., experiment settings). For this, four types of configurations should be set, discussed in the following.

5.1 Scheduler

Figure 5.1 shows the scheduling configuration panel. Using this, the user determines the desired scheduling policy, and the respective properties.


Scheduler

Policy: EDF

Preemption: Limited_Preempti...

Tick: 20

Figure 5.1: Scheduler configuration.

From an abstract point of view, system execution consists of a set scheduling instant (points) where the scheduler is invoked, and determines which nodes should be run on which cores. At each scheduling point, selecting the most eligible nodes to execute is determined by the scheduling *policy*. In turn, scheduling points are determined according to the *preemption* method.

5.1.1 Scheduling Policy

Given a set of eligible nodes, the policy determines which one(s) has (have) the higher priority to execute. Scheduling policies implemented in the tool are described in the following table.

Policy	**Description**
RM	RM (Rate monotonic) scheduling. See Section 3.2.
EDF	EDF (Earliest Deadline First) prioritizes the eligible nodes according to the absolute deadline of the corresponding tasks; the earlier (i.e., the closer) the deadline, the higher the priority. Ties (between nodes with the same deadline) are broken according to a static priority.
LLED	LLED (Least-Laxity Earliest-Deadline) is a version of EDF where ties are broken by the laxity; the less the laxity, the higher the priority.
EDLL	In EDLL (Earliest-Deadline Least-Laxity), the less the laxity, the higher the priority. Ties are broken by deadline: if two nodes with the same laxity, the one with the closer deadline gets higher priority.
Random	Eligible nodes are randomly selected to execute.
FP_FIFO	Jobs with earlier arrival time get higher priority. Ties are broken by a fixed priority.
Federated	For each task, the minimum number of required cores is computed and those cores are exclusively dedicated to the task.
PReserved	Once a new instance of a task is released, all required resources (based on WCETs) are reserved for that. During time instants where a core is not used by the task, this can be used to run other tasks.
Dynamic Federated [4]	This is similar to the Federated scheduling, but once a core is for sure not required by the task instance anymore (according to an LL offline schedule), the core is used to run other tasks. This helps better resource utilization.
Dynamic Federated BFS [4]	This is similar to Dynamic Federated, but the offline schedule is built by BFS traversal of the nodes.
FP_Harmonic	The same as RM, but the analysis optimized for harmonic tasks, It does not apply to the general case.

5.1.2 Preemption

The preemption method determines if and when the scheduler is allowed to preempt (pause) an executing job and give the resource to another one. Preemption approach can be *tick* based. Roughly, in a tick-based approach, scheduling is done only at certain time points, called ticks.

Method	**Description**
Non-Preemptive	Node-level non-preemptive scheduling (some times called “limited preemptive” approach in, e.g., [13]). Once a node is started, it is continued until finished.
Preemptive	There is no restriction on preempting the jobs.
Ticked-Preemptive	This is similar to Non-Preemptive, but jobs can be preempted at ticks.
NW-Ticked-Preemptive	This is a *Non-Work-Conserving* version of Ticked-Preemptive. The scheduler is invoked, and allowed to preempt the jobs, only at ticks, where ticks are equally-distant time points (specified by the user). If a job is finished before a tick, the core is idled until the next tick.

Figures 5.2 and 5.3 illustrate Ticked-Preemptive and NW-Ticked-Preemptive approaches, respectively.

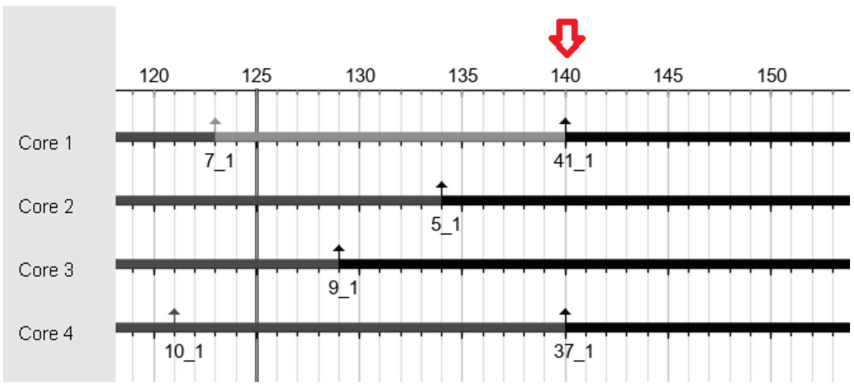


Figure 5.2: Ticked-Preemptive scheduling. Preemption is allowed only at ticks (Here, tick=20).

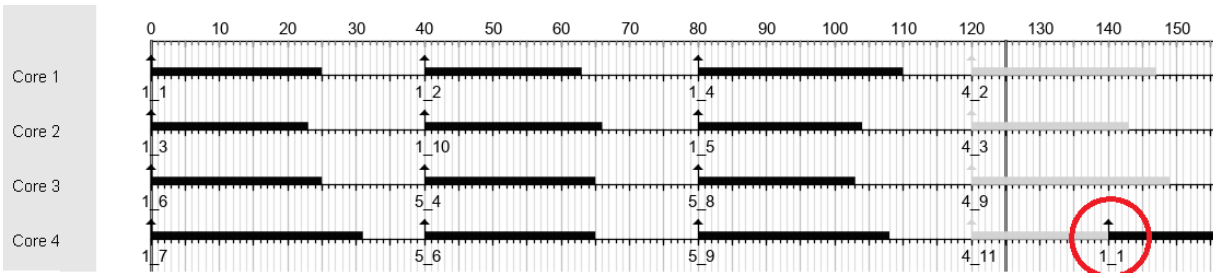
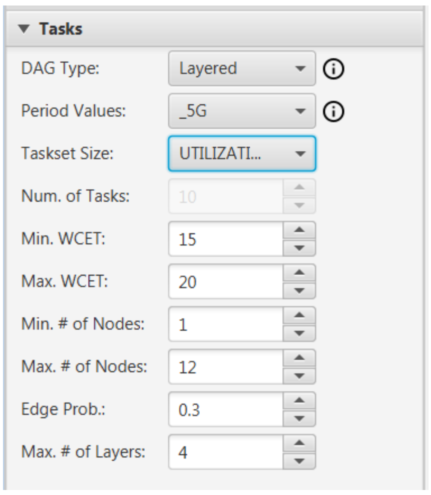


Figure 5.3: NW-Ticked-Preemptive scheduling. Scheduling is done only at ticks (Here, tick=20). At $t = 140$, there is a tick where a job is preempted by a higher priority one.

5.2 Task Set

Real-time task parameters that can be set by the user are seen in Fig. 5.4. These parameters are used for random task set generation. In particular, the following aspects will be specified.



Tasks

DAG Type:	Layered
Period Values:	_5G
Taskset Size:	UTILIZATI...
Num. of Tasks:	10
Min. WCET:	15
Max. WCET:	20
Min. # of Nodes:	1
Max. # of Nodes:	12
Edge Prob.:	0.3
Max. # of Layers:	4

Figure 5.4: DAG tasks parameters.

- **DAG type (graph structure)**: Three types of DAG structures are included in the tool, i.e., Layered, SeqParallel, and STR2RTS, which are described in Sec. 4.1.
- **Period**: The approach for assigning periods to the tasks. The period sets used for period assignment are described in the following table.

Period type	**Description**
RELAXED	In this approach, tasks utilization are first generated using the UUnifast algorithm [1] such that the total utilization equals the desired value. Then, the period of each task is computed through dividing its total WCET by its utilization, i.e., $T_i = C_i/U_i$.
5G	The 5G periods (see Sec. 4.2).
AUTOSAR	Periods from AUTOSAR (see Sec. 4.2).
AUTOSAR_EXT	AUTOSAR periods extended to cover more diverse values (obtained from [9]). The period values come from the set $\{x \times 10^y \| 1 \leq x \leq 9, 3 \leq y \leq 5\} \cap [500, 100000]$
AUTOSAR HARMONIC	A subset of AUTOSAR periods that build a harmonic set; that is, $\{1, 2, 10, 20, 100, 200, 1000\}$ ms.

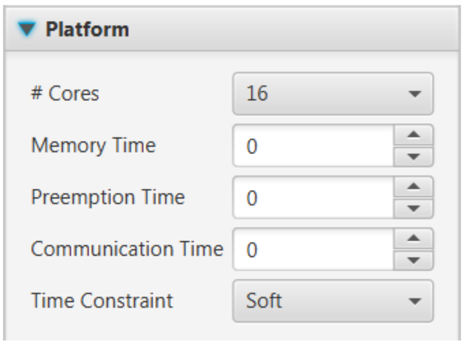
- **Number of Tasks**: There are two methods to determine the number of tasks in each task set (i.e., task set size): FIXED, where all ask sets are of the same size, specified by the user, and UTILIZATION_BASED, where randomly generated tasks are added to the task set until reaching the desired utilization.

- **Nodes WCET**: The criteria for assigning WCET to the nodes is specific to the selected DAG Type. For the *Layered* type, the range for WCET is obtained from the user, the WCET of each node will be randomly selected with a uniform distribution from the range. For the STR2RTS type, the WCETS come from the benchmark data. For SeqParallel DAG tasks, the WCET is randomly selected from the range $[1, 50]$.

The meaning of the other task parameters in the figure is described in Sec. 2.1.

5.3 Platform

Figure 5.5 shows the parameters from the hardware platform that the user can specify. Currently, a homogeneous multicore processor is assumed. Number of cores is determined by the parameter labeled as # Cores, as seen in the figure. Other parameters are described below.



# Cores	16
Memory Time	0
Preemption Time	0
Communication Time	0
Time Constraint	Soft

Figure 5.5: Platform parameters.

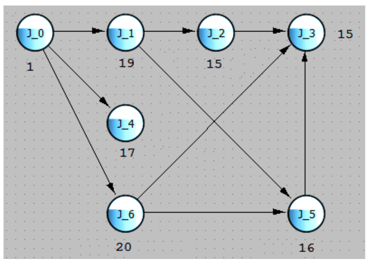
5.3.1 Memory-related timings

The execution time of a job not only depends on the time CPU is executing the respective code, but also on the time it takes to access the memory. A task instance needs memory access to load the required data upon initialization. The memory is supposed to be accessed only by one task at each time instant. So, if more than one task instances are released at a time, they will compete. This can add delays to the start of a task instance.

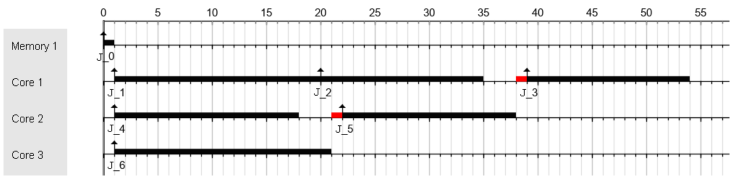
Additionally, the context-switch caused by preemption, and also, transferring data between CPU cores should be considered as a part of the total job execution times. These platform-related timings are captured by three parameters described in the following table.

Parameter	Description
Memory Time	The time it takes for each task instance to load initialization data from memory (once it is granted the memory access).
Preemption Time	Whenever a preempted job is dispatched to be executed again, this (constant) delay is added to account the context-switch overheads.
Communication Time	Once a node is dispatched to be executed for the first time, if at least one of its predecessors have been run on a different core, a communication delay is added before this start.

As an illustrating example, consider Fig. 5.6a where a DAG with seven nodes is shown. In this example, the initialization phase, i.e., reading from memory, has been represented by an individual node `J_0`. A sample schedule of this task on three cores is depicted in Fig 5.6b. During time interval $[0, 1]$, the task is reading from memory.



(a) DAG structure (values next to the nodes represent WCET.)



(b) A sample schedule. Initialization (reading from memory) takes one time unit. Start of J_3 is delayed one time unit (communication delay) since it has a predecessor (J_6) executed on a different core (i.e., Core 3).

Figure 5.6: Scheduling of a DAG where Memory Time = Communication Time = 1

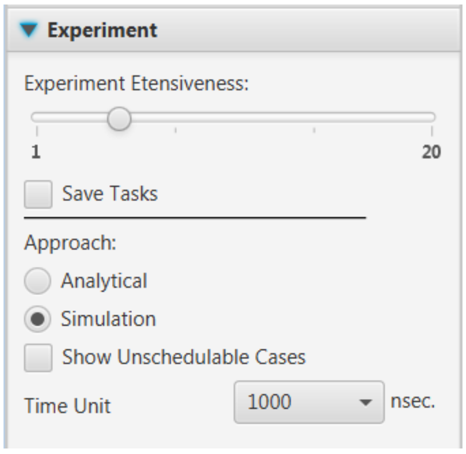
5.3.2 Time constraint

Time constraints on the tasks is either Soft or Firm. The meaning of each is described below.

Constraint	Meaning	Scheduler Behavior
Soft	Completion of jobs after deadline is of some value.	Jobs are dispatched and executed even after deadline.
Firm	Completion of jobs after deadline is of no value.	A job is dropped out of the system as soon as its deadline is passed.

5.4 Experiment Parameters

An *experiment* consists of generating a set of random DAG tasks, performing schedulability and timing analysis, and presenting the results. Figure 5.7 shows the general experiment configuration.



Experiment Etensiveness:

Save Tasks

Approach:

Analytical

[✓ Simulation]

Show Unschedulable Cases

Time Unit 1000 nsec.

Figure 5.7: Experiment settings.

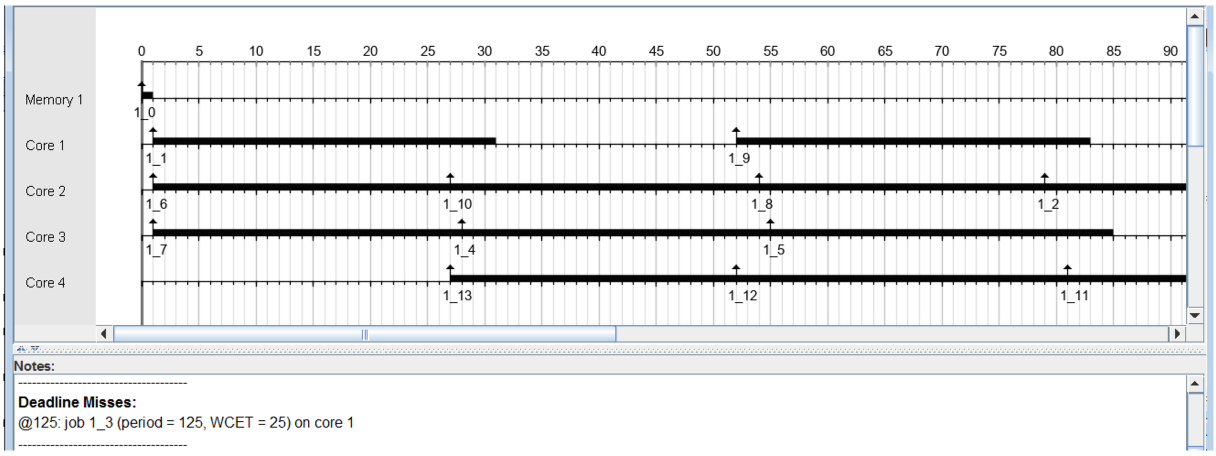
The experiment parameters are described in the following table.

Parameter	Description
Experiment Extensiveness	It can be a number between 1 and 20. A value of n means the experiment will generate and evaluate $n \times 1000$ random DAG tasks.
Save Tasks	If checked, those generated task sets that are unschedulable will be saved in an XML file readable by the TimesPro tool[1].
Approach	Determines whether evaluation of task sets is done through simulation of one hyper-period, or through analytical methods.
Show Unschedulable Cases	If checked, for each unschedulable task set, the simulated schedule is graphically represented (see Fig. 5.8 as an example).
Time Unit	Determines the time unit of the specified parameters (e.g., task parameters). Smaller time unis allow more resolution in specifying the parameters, but more time-consuming simulations.

As seen, the approach employed for evaluation of task sets can be set by the Approach parameter, which can be either Analytical or Simulation. Analytical methods usually provide pessimistic but safe results. In contrast, simulation is done only for one possible scenario, and it does not guarantee the result for all possible situations. More precisely, in the simulation, the worst-case execution time (WCET) of the nodes is used as the actual execution time. The obtained result does not necessary hold if the actual execution time is less than WCET.

As mentioned in the table above, the scheduling trace of unschedulable task sets is visualized once the `Show Unschedulable Cases` parameter is checked. Such a sample trace is seen in Fig. 5.8.

[1]http://www.it.uu.se/research/group/darts/timespro



Deadline Misses:
@125: job 1_3 (period = 125, WCET = 25) on core 1

Figure 5.8: Simulation trace of an unschedulable task set.

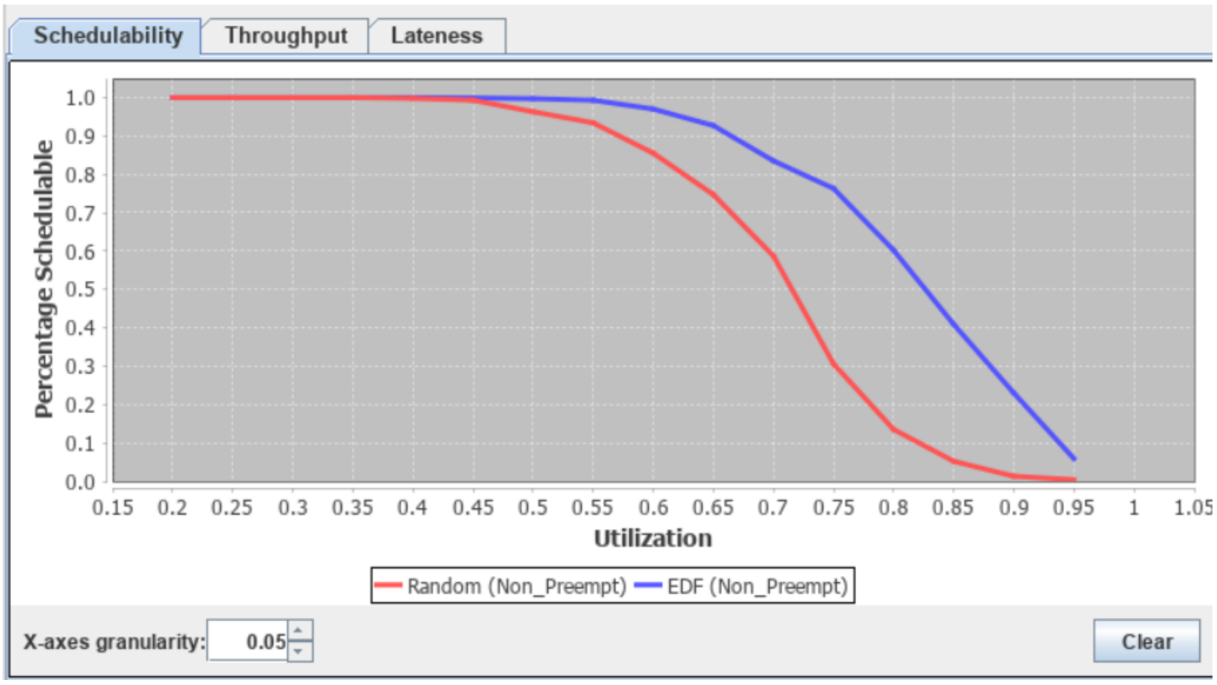
Chapter 6

Tool Manual: Analysis Results

The result of an experiment is a set of measures computed by the tool. These measures and the representation method are described in this chapter.

6.1 Schedulability Ratio

During system execution, each DAG releases several DAG instances. A DAG instance meets its deadline if all of the nodes inside the DAG are done by the deadline. A DAG is said to be schedulable if all of its instances meet their deadline. In turn, a task set is schedulable if all of its DAGs are schedulable. Given a set of task sets, a favorite measure is the percentage of schedulable task sets.



X-axes granularity: 0.05

Figure 6.1: Schedulability ratio.

Figure 6.1 shows the average percentage of task sets that are schedulable for each *utilization*. The utilization of a DAG task is defined as the total WCET of its nodes divided by the task's period. Formally, the utilization of a DAG task G with period T_G is computed by

$$U(G) = \frac{\sum_{n \in nodes(G)} wcet(n)}{T_G}$$

Let τ_U be the set of all task sets, generated in an experiment, whose utilization is U. Further,

$$\tau_s = \{\tau \in \tau_U | \text{ task set } \tau \text{ is schedulable}\}.$$

The percentage of schedulable task sets, which is plotted by the diagram, is computed by $|\tau_s|/|\tau_U|$, where $|\tau_s|$ and $|\tau_U|$ denote the size of τ_s and τ_U, respectively.

6.2 Throughput

In a system execution, a task set may be unschedulable while many of the task instances meet their deadline. To account for this, the *throughput* measure is used. The throughput of a task set is the number of DAG instances that meet their deadline divided by the total number of DAG instances released in one hyper-period. The latter is computed by $\sum_{i=1}^{n}(HP/T_i)$, where HP is the hyper-period and T_i is the period of the i-th DAG task. Figure 6.2 shows the average throughput of two scheduling policies. The average throughput of a set of n task sets is obtained by dividing the sum of their throughput by n.

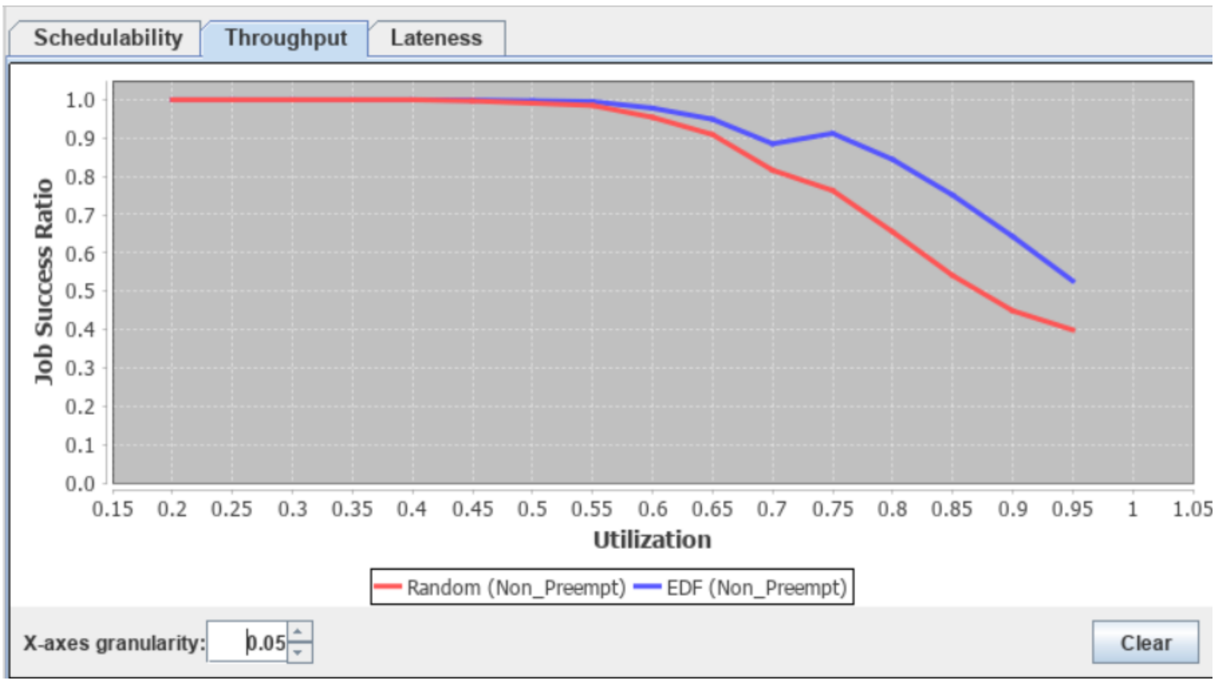


Figure 6.2: Throughput.

It is worth noting that for a set of task sets, throughput is always larger than or equal to the schedulable percentage.

6.3 Lateness

Schedulability ratio and Throughput measures do not provide any information on how late/how early DAG instances are finished. The lateness measure reflects this. The lateness of a DAG instance with an absolute deadline of d finished at time f is defined by $f - d$. A positive value of lateness means a deadline miss. Note that lateness may also be negative.

To represent the lateness data of an experiment, DATooRuses *frequency plots*. Consider an execution of a task set τ in one hyper-period. Let n be the total number of DAG instances released, and $n(l)$ be the number of those DAG instances whose lateness equals l. The relative lateness frequency is a function defined as

$$lf_{\tau}(\ell) = \frac{n(\ell)}{n}.$$

Based on this definition, it holds $\int_{-\infty}^{\infty} lf_{\tau}(\ell) = 1$. In fact, $lf_{\tau}(.)$ can be seen as a *probability mass function (PMF)* [14], where $lf_{\tau}(\ell)$ denotes the probability that a randomly selected DAG instance have a lateness of l. The curve plotted by the tool is an average over all functions $lf_{\tau}(.)$ obtained for each task set. That is, if the experiment is done for a set of task sets $\mathcal{T}$, the value of the curve for a value of ℓ in x-axis is obtained by

$$\sum_{\tau \in \mathcal{T}} lf_{\tau}(\ell)/|\mathcal{T}|$$

. Figure 6.3 shows the frequency plot of lateness for a simulation run.

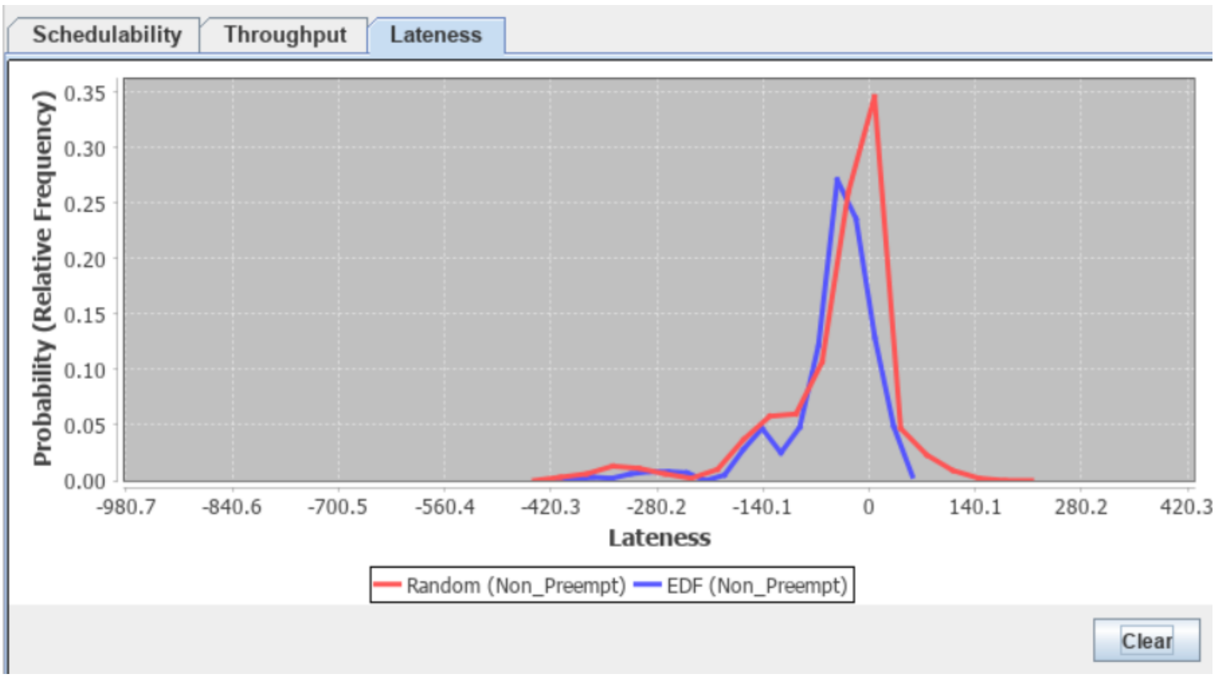


Figure 6.3: Lateness frequency plot.

Chapter 7

Developers Guide

This section describes the code structure, and serves as a guide for those who want to extend the tool. DATooR is developed by the Java programming language. The current version is developed by OpenJDK 11.0.2.

7.1 Code Structure

The overal structure of the code is seen below.

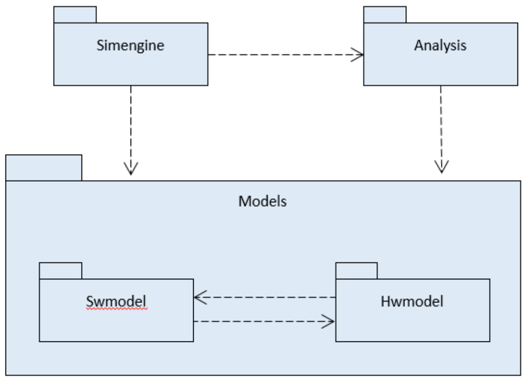


Figure 7.1: The overall code structure.

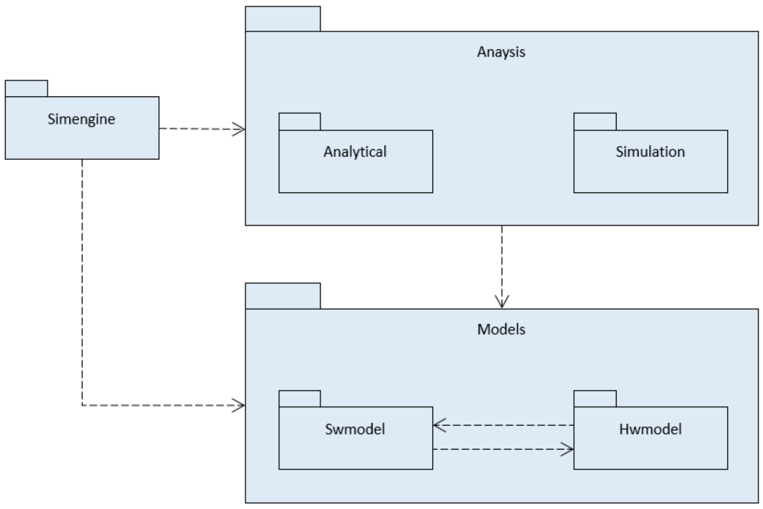


Figure 7.2: A more detailed view on the code structure.

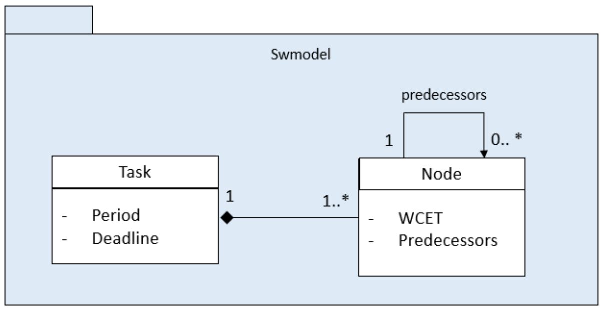


Figure 7.3: The software model.

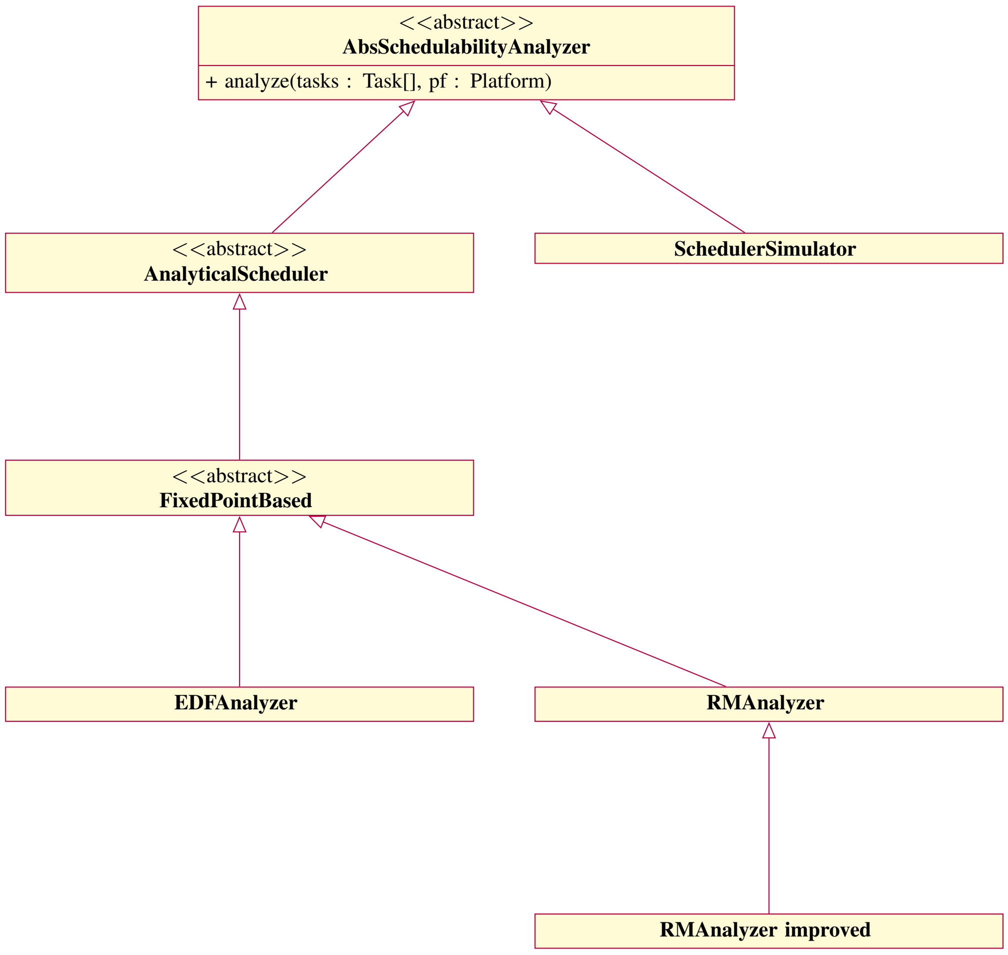


Figure 7.4: The class hierarchy of analytical methods.

Bibliography

[1] Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[2] Daniel Casini, Alessandro Biondi, Geoffrey Nelissen, and Giorgio Buttazzo. Partitioned fixed-priority scheduling of parallel tasks without preemptions. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 421–433, 2018.

[3] Daniel Cordeiro, Grégory Mounié, Swann Perarnau, Denis Trystram, Jean-Marc Vincent, and Frédéric Wagner. Random graph generation for scheduling simulations. 2010.

[4] Gaoyang Dai, Morteza Mohaqeqi, and Wang Yi. Timing-anomaly free dynamic scheduling of periodic DAG tasks with non-preemptive nodes. In *IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 119–128, 2021.

[5] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

[6] Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 85–96. IEEE, 2014.

[7] Shao-Yu Lien, Shin-Lin Shieh, Yenming Huang, Borching Su, Yung-Lin Hsu, and Hung-Yu Wei. 5g new radio: Waveform, frame structure, multiple access, and initial access. *IEEE communications magazine*, 55(6):64–71, 2017.

[8] Roberto Medina, Etienne Borde, and Laurent Pautet. Scheduling multi-periodic mixed-criticality dags on multi-core architectures. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 254–264. IEEE, 2018.

[9] Mitra Nasri, Geoffrey Nelissen, and Björn B Brandenburg. Response-time analysis of limited-preemptive parallel dag tasks under global scheduling. In *31st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 21–1, 2019.

[10] Bo Peng, Nathan Fisher, and Marko Bertogna. Explicit preemption placement for real-time conditional code. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 177–188. IEEE, 2014.

[11] Benjamin Rouxel and Isabelle Puaut. Str2rts: Refactored streamit benchmarks into statically analyzable parallel benchmarks for wcet estimation & real-time scheduling. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[12] Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D Gill. Parallel real-time scheduling of DAGs. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3242–3252, 2014.

[13] Maria A Serrano, Alessandra Melani, Sebastian Kehr, Marko Bertogna, and Eduardo Quiñones. An analysis of lazy and eager limited preemption approaches under dag-based global fixed priority scheduling. In *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 193–202. IEEE, 2017.

[14] Kishor Shridharbhai Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*, volume 13. Wiley Online Library, 1982.