ABSTRACT

Within the context of hard real-time systems, the schedulability analysis of a task set is a major issue. The problem consists in proving that the tasks always satisfy their temporal constraints for a given scheduling policy and a given platform. Extensive work has been done in the last decades for defining sufficient criteria and exact algorithms. Sufficient criteria usually have an excellent complexity but often lead to an over-dimension of the system. On the opposite, exact algorithms, especially in the case of multiprocessor platform, suffer from an exponential complexity.

In this paper, we study an exact technique: we apply a brute force search combined with a model checker (UP-PAAAL) that determines whether the exploration is complete. We consider periodic tasks which execute on parallel platforms composed of homogeneous processors. Under these hypotheses, we have encoded four policies: fixed task priority, gEDF, gLLF and LLREF. The analyser is user friendly and provides promising performances.

1. INTRODUCTION

The use of digital embedded systems becomes natural for implementing highly critical systems. In the transport sector, numerous steps have been made, leading to the development of critical functions-by-wire and unmanned vehicles. With the increasing cost of energy, manufacturers generally try to optimize the power consumption, for instance by improving the aerodynamics and the weight of aircraft. This leads to flexible and unstable vehicles which need more accurate and complex control functions requiring more and more computing resources. Thus, in the next years, it will become unreasonable to expect to implement such control functions on uniprocessor platforms. Additionally, next generation of embedded chips will evolve toward multicore architectures. Thus, this architectural evolution, combined with stronger functional and timing requirements, needs efficient tools and methods for carefully managing key resources, such as core capacity, and for proving that the implementation is correct.

1.1 Real-time constraints

Embedded systems must satisfy hard timing constraints. A periodic real-time task is a functional code which executes periodically and which must satisfy some timing constraints. A task set is composed of several tasks \{\tau_1, \tau_2, \ldots, \tau_n\} and each task is characterised by four real-time attributes \(T, C, O, D\) where \(T\) is the period of repetition, \(C\) is the worst case execution time estimation, \(O\) is the first arrival time and \(D\) is the relative deadline which is assumed to always be less or equal to the period (we say that the task is constrained-deadline). Each repetition of the task is usually called job or instance. This model of task is a standard adaptation of the Liu and Layland model \[15\] and is more restrictive than other models that can be found in the literature such as the arbitrary deadline periodic or the sporadic tasks model \[7\].

Usually, scheduling strategies are classified as preemptive versus non-preemptive, and off-line versus on-line policies. In non-preemptive case, each task instance, when started, completes its execution without interruptions. Conversely, in preemptive case, the scheduling unit can suspend a running task instance if a higher priority task asks for the processor. Off-line scheduling is based on a schedule which is computed before run-time and stored in a table executed by a dispatcher. Conversely, the idea of on-line scheduling is that scheduling decisions are taken at run-time whenever a running task instance terminates or a new task instance asks for the computing resources. On-line preemptive scheduling strategies allow powerful and flexible implementations in the price of complexity and non determinism.

We consider in the following real-time embedded systems implemented on multicore architectures with on-line preemptive scheduling, i.e., scheduling strategies where an executing instance may be interrupted at any instant and will be resumed later with no cost. Moreover, we assume that a task can migrate from one core to another, but that a task cannot execute on two different cores at the same instant (there is no intra-task parallelism).

For a given set of real-time multiperiodic tasks, the question is twofold: firstly selection of a scheduling policy that will order the execution of the tasks according to their real-time attributes; secondly the schedulability analysis, i.e. the verification that all the timing constraints are met. This paper focuses on the second question.

1.2 Scheduling strategies for multiprocessors architectures

For multiprocessor platform, several impressive results have been developed and summarised in surveys such as \[1, 7\].
Two main approaches have been developed for multiprocessor: the so-called partitioning strategy, where an algorithm allocates a distinct subset of the task set to each processor (this can be done at task or job level), and the global strategy, where there is a single queue from which the tasks with the highest priority, according to the policy, execute on the available processors. In this paper, we only consider global strategy. Classical scheduling policies can easily be extended for multiprocessor. When we speak of an extension of a classical policy on a multiprocessor platform, we add the letter "g" before the name (for global). We have chosen to focus on the following policies:

1. **FP** (fixed priority) is the simplest policy and has the favor of industrialists. The tasks are allocated offline a unique priority.

2. **EDF** (Earliest Deadline First) was introduced by Liu and Layland [15]. In that case, the priority is given according to the next absolute deadline. The highest priority is for the task with the smallest deadline. It is an optimal policy for constrained-deadline periodic task set on monoprocessor but not on multiprocessor.

3. **LLF** (Least Laxity First) was introduced by Mok [16]. The priorities are assigned every time unit and the task with the highest priority is the one with the smallest laxity \( l_r(t) = D_r - C_r(t) = \left(t - O_r\right) \mod T_r \) where \( C_r(t) \) is the remaining execution time to do at time \( t \) for the task \( r \). This policy belongs to the family of fully dynamic priority where two jobs can be ordered differently at two distinct instants. For monoprocessor platform, this property does not act on optimality since both EDF and LLF are optimal for constrained-deadline periodic tasks. But, for multiprocessor platform, gLLF and gEDF are incomparable [13].

4. **LLREF** (Largest Local Remaining Execution First) was introduced by Cho and al. [4] specifically for multiprocessor. This policy mixes ideas of Baruah and al. on PFair [2] and LLF. Under PFair, tasks make progress at steady rates. In LLREF, the execution is divided in time slots whose length depends on the closest next wake or next deadline of a job. At instant \( t \), let us denote the next time \( T' \), which is equal to the closest next wake or next deadline. In order to assign the priority, two parameters are computed:

(a) \( l_r \) represents the local remaining execution time of \( r \) in the interval \([t, T']\). It is a fair execution to be done:

\[ l_r(t) = C_r(t) \times \frac{T' - t}{D_r - t} \]  \hspace{1cm} (1)

(b) \( L_r \) is the local laxity of \( r \) in the interval \([t, T']\):

\[ L_r(t) = T' - t - l_r(t) \]  \hspace{1cm} (2)

The policy works as follows:

(a) if the laxity is zero, that is \( L_r(t) = 0 \), the remaining time till \( T' \) is necessary to complete the execution of \( r \). Thus, the execution is urgent and the task is assigned the highest priority;

(b) for the set of tasks which laxity is not null, the highest priority is given to those with the highest fair execution \( l_r \).

LLREF is optimal for periodic task set with \( O = 0 \) and implicit deadline \( D = T \) on multiprocessor.

These four policies are interesting, thus we want to formally prove the schedulability of a task set on a given [multiprocessor] platform for any of them.

### 1.3 Contribution and related research

**Schedulability analysis.**

A schedulability analysis takes as an input a platform of resources, a task set and a scheduling policy. The objective is to determine whether any execution always respects the temporal constraints, i.e. there is no deadline miss.

For instance, for a monoprocessor, a distinct fixed priority scheduling and a task set, it is sufficient to simulate an execution where the tasks take their worst case execution time and this during a sufficient window \([0, \max(O_i) + 2H] \) where \( H \) is the hyperperiod of the periods of the tasks. For monoprocessor platform, there exists a series of techniques for proving the schedulability which are implemented in many tools, such as Cheddar [17] for instance.

In contrast, for multiprocessors machines, the schedulability analysis problem is a relatively new research area, and few methods are available or are restricted to specific task configurations.

The aim of this article is to present an exact method for multiprocessor schedulability analysis based on a brute force approach. We deal with the four scheduling strategies presented in the previous section. And we show that, in most cases, our brute force method is very efficient: up to 1000 tasks for fixed priority and gLLF, 150 tasks for gEDF and 85 tasks for LLREF.

**Related research.**

Several authors have exhibited numerous utilization bound tests that are sufficient criteria: if the task set satisfies some formula, then the task set is schedulable. Utilization bound test is very efficient in terms of complexity. But it is often too pessimistic and leads to system over-dimensioning. We would rather be interested in an exact algorithm. Existing exact schedulability analyses are often brute force search and few tools are available. Therefore, efficient schedulability analysis and a generic tool remain to be two open problems.

For monoprocessor, the modelling of the schedulability analysis as property checking is not new: the authors of [6] modelled a real-time system as a network of timed automata; the authors of [11] proposed a timed Petri net model; the authors of [8] used timed process algebra. The first two works use dates in \( \mathbb{N} \) while the last look for general dates which can be in \( \mathbb{R} \).

For multiprocessor, Guan and al. [11] showed that if all the real-time attributes (periods, offsets, deadlines and worst case execution time) are in \( \mathbb{N} \), then it is sufficient to consider integer dates and traces in \( \mathbb{N} \). The problem becomes then discrete. In other terms, it is useless to apply hybrid or dense-time verification methods. In their first contribution [11], the authors have modelled the schedulability analysis for the fixed priority policy as a model checking problem of a
network of UPPAAL timed automata. They used UPPAAL [3] for evaluating the performances. In their second work [4], they provide a modelling using finite SMV automata for the fixed priority and EDF global scheduling policies. The task model was restricted with null offset and deadline equal to period for full migration scheduling.

In the same way, David and al. proposed in [6] a framework allowing the analysis of tasks configuration for multiprocessor machines. Their framework supports rich tasks models including timing uncertainties in arrival and execution times, dependencies between tasks (such as precedences and resource occupation). It takes into account three scheduling policies: FIFO, fixed priority and gEDF. The schedulability analysis is based on the approximate analysis of stopwatch automata in UPPAAL.

However, both schemes in [12] and [6] quickly encountered the problem of combinatorial explosion beyond configurations of tasks composed of more than 20 tasks. To our opinion, this difficulty comes from the modelling approach chosen for describing the configuration of tasks and the scheduling policies. They modelled each task as an explicit automaton composed of several states (about 5 to 9 states). The global system is then described as a network of timed automata, leading to a global state space containing possibly up to $5^n$ states where $n$ is the number of tasks. This modelling approach seems to be not suitable for realistic configurations.

An other way could be to define a specific search algorithm. But this solution encounters two difficulties:

- The first problem comes from the feasibility interval, that is the time period where we have to check that no task misses its deadline. In [5], Cucu and Goossens show that the hyperperiod $(0, H)$ is a feasibility interval for fixed priority scheduling policy and for synchronous (with null offset) implicit and constrained deadlines (deadlines equal to period) task sets. For arbitrary asynchronous task systems, they show the feasibility interval exists (the system is globally periodic), but as far as we are aware, no exact characterization is known for this feasibility interval.

- The second problem deals with efficiency. Even in the case of the theoretical feasibility interval is known, the algorithm must visit every scenarios which can be numerous.

The consequence of these two points is that any specific brute force algorithm must memorize all the explored states in order to detect when the system becomes periodic, and then to stop the search. Model checkers like UPPAAL or SMV are designed to efficiently encode state space exploration and to detect when the system returns in previously visited states. Our opinion is that it could be more efficient to combine a specific algorithm with a model checker which supports the storage and exploration. This is the approach we propose in this article.

**Contribution.**

Contrary to Guan et al. and David et al., our modelling approach is based on

- a set of tables describing the task set current configuration;
- two evolution transitions:
  - one representing the time step. At each firing, 
  C-like functions compute the new configuration (values of the tables and priorities according to the scheduling policy);
  - the second representing the non determinism since a simulation is generally not sufficient. Note that this question was not treated in the existing modellings.

Combined with the UPPAAL model checker, we show that this modelling leads to a very small state space, and then is able to manage realistic task sets. Note that for synchronous task sets with implicit deadline, there exists a necessary and sufficient test for LLREF: $\Sigma \frac{C_i}{T_i} \leq m$ where $m$ is the number of processors. But for other kinds of task sets, a complete verification is necessary.

In addition, we provide a general tooset [7] to analyse a system. The user declares the set of tasks and the number of processors. The modelling in UPPAAL for the specified scheduling policy is automatically generated.

The paper is organised as follows: in the section 2, we describe the formalisation of the schedulability analysis. In section 3, we summarize experimental results before concluding.

## 2. EFFICIENT EXPLORATION

We present the coding of the schedulability analysis based on the contribution of (1) the encoding of all the scenarios and (2) the application of the UPPAAL model checker to handle the efficient and complete exploration.

### 2.1 Encoding of an execution

We first need to encode the executions. For this, we introduce some variables to describe the current configuration of each task based on the real-time attributes. For a task $\tau$ with attributes $(T_r, C_r, O_r, D_r)$, an execution is coded with:

- $T_r^\tau : \mathbb{N} \rightarrow [0, T_r]$ where $T_r^\tau(t)$ represents the remaining time till the beginning of the next activation,
- $C_r^\tau : \mathbb{N} \rightarrow [0, C_r]$ where $C_r^\tau(t)$ represents the remaining execution time of the current job,
- $O_r^\tau : \mathbb{N} \rightarrow [0, O_r]$ where $O_r^\tau(t)$ represents the remaining time till the starting time of the first job,
- $D_r^\tau : \mathbb{N} \rightarrow [0, D_r]$ where $D_r^\tau(t)$ represents the remaining time till the deadline of the current job,
- $n_r^\tau : \mathbb{N} \rightarrow \mathbb{B}$ where $n_r^\tau(t)$ represents the access to a processor. The value of this variable depends on the policy and will be discussed in the next section.

At time $t$, the configuration of task $\tau$ is the tuple $(T_r^\tau(t), C_r^\tau(t), O_r^\tau(t), D_r^\tau(t), n_r^\tau(t))$. To highlight the coding, we use an example.

**Example 1.** Let us for instance consider the task set:

1. The source code can be found at [http://www.cert.fr/francais/deri/mcordovi/modelisation.html](http://www.cert.fr/francais/deri/mcordovi/modelisation.html)
<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>τ₀</td>
<td>(5,2,0,5,t)</td>
<td>(4,1,0,4,t)</td>
<td>(3,0,0,3,t)</td>
<td>(2,0,0,2,t)</td>
<td>(1,0,0,1,t)</td>
<td>(0,0,0,0,f)</td>
<td>(5,2,0,5,t)</td>
</tr>
<tr>
<td>τ₁</td>
<td>(5,2,1,5,t)</td>
<td>(5,2,0,5,t)</td>
<td>(4,2,0,4,t)</td>
<td>(3,1,0,3,t)</td>
<td>(2,0,0,2,t)</td>
<td>(1,0,0,1,t)</td>
<td>(0,0,0,0,f)</td>
</tr>
<tr>
<td>τ₂</td>
<td>(5,5,1,5,t)</td>
<td>(5,5,0,5,t)</td>
<td>(4,4,0,4,t)</td>
<td>(3,3,0,3,t)</td>
<td>(2,2,0,2,t)</td>
<td>(1,1,0,1,t)</td>
<td>(0,0,0,0,f)</td>
</tr>
</tbody>
</table>

For the policy gEDF there are several possible executions which are partly drawn below.

\[ \begin{array}{cccccc}
\tau₀ & \tau₁ & \tau₂ & \tau₀ & \tau₁ & \tau₂ \\
5 & 5 & 5 & 0 & 0 & 0 \\
5 & 2 & 1 & 5 & 2 & 0 \\
5 & 5 & 5 & 1 & 1 & 1 \\
\end{array} \]

On the left size, the scheduling is correct while on the right side, the task \(\tau₂\) misses its deadline. This illustrates the non-determinism. Unlike the monoprocessor case, a simulation is insufficient for multiprocessor platform.

The fig. encodes the left execution of example. Note that when a new job is activated, there are two implicit configurations: the end of the current job execution which must be equal to (0,0,0,0,f) when everything went fine and the values for the new job. This is symbolised by (0,0,0,0,\ldots) \(\rightarrow (T_r, c_r, 0, d_r, \ldots)\).

### 2.2 Exploration with Uppaal

The brute force exploration is represented as an automaton and a set of C-like functions which are applied on time step execution.

#### 2.2.1 Automaton

The automaton associated to the exploration is depicted in the Fig. 2. The automaton works as follows:

- the loop labelled update_tab simply computes the values of the tuples \((T^r_r(t), C^r_r(t), O^r_r(t), D^r_r(t), n^r_r(t))\) after one unit of time (i.e. from \((T^r_r(t-1), C^r_r(t-1), O^r_r(t-1), D^r_r(t-1), n^r_r(t-1))\));
- the second loop generates the non-determinism and schedulers for tasks with the same priority. It applies only for gEDF and LLREF. For FP we assume the priorities to be distinct and for gLLF, see theorem G.
- the transition labelled unhappy=1? can be fired only if the variable unhappy is equal to 1 which only occurs when the task set is not schedulable.

#### 2.2.2 Task set configurations

The configurations are represented in Uppaal by a set of tables. More precisely, we use the variables:

1. The constant \(nbTasks\) is the number of tasks and \(nbProc\) is the number of processors. The task set is thus \(\{\tau_1, \ldots, \tau_{nbTasks}\}\). The real-time attributes are stored in the constant arrays int \(list_T[nbTasks]\), int \(list_O[nbTasks]\), int \(list_D[nbTasks]\) and int \(list_C[nbTasks]\).
2. The variables representing the remaining times are stored in the arrays int \(list_TC[nbTasks]\), int \(list_CC[nbTasks]\), int \(list_OC[nbTasks]\) and int \(list_DC[nbTasks]\).
3. The variable \(int list_prio[nbTasks]\) gives the ordered list of tasks. It is equivalent to the definition of \(n_i\):

\[ n_i = true \Leftrightarrow (list_prio[k] = i \land k \leq nbProc) \]

The task \(i\) can execute on a processor if it appears at position \(k\) in the list and \(k\) is less than the number of processors;
4. The variable \(int nbtaskawake\) counts the current number of awaken tasks, i.e. such that \(O^c_r = 0 \land C^c_r > 0\).
5. The variable \(int unhappy=0;\)

#### 2.2.3 Time step

We describe more precisely the C-like functions. First, update_tab represents the execution of one time unit:

```c
void update_tab() {
    int i, nbtaskawake_inter=0;
    for (i=0;i<nbTasks;i++) {
        /* step 1: update of the execution times */
        if (needed) {
            list_cc[ list_prio [i] ]--; 
            if (list_cc [ list_prio [i] ]<=0) {
                nbtaskawake--; 
            }
        }
        /* step 2: update of the other parameters */
        if (list_oc [i] ==0) {
            list_tc [i]--; 
            if (list_dc [i]>0) list_dc [i]--; 
            if (list_tc [i]==0) {
                list_cc [i]--; 
                list_tc [i] = list_dc [i];
                list_cc [i] = list_dc [i];
                nbtaskawake++; 
            }
        }
    }
}
```

The execution can be seen in 4 main steps:

1. step 1: the current execution time is modified for the highest priority tasks. If some task ends, with \(C^c_r = 0\), the variable nbtaskawake is decremented by 1.
2. step 2: the other parameters are updated. If the offset is not null and the task has not reached the starting time, the variable list_loc[i] is decremented. If the starting time is passed, the variables list_tc[i] and list_dc[i] are decremented.

3. step 3: a condition necessary is evaluated, depending on the policy. If the condition holds, there is a new computation of the list of priority. For the policy FP, list_prio is constant. For gEdward, the condition is true when there is a wake up or an end of a task. For gLLF, the condition is true when a task wakes up or ends too, but also whenever there is a change in the priority ordering. For the sake of simplicity, we assume that necessary is always true if nbtaskwakeup is greater than nbProc. For LLREF, the condition is true when a task wakes up or ends, and also when the variable nextwakeup is null. This will be detailed in section 2.3.2.

4. step 4: at each time step, the function verifies that no deadline is missed.

2.2.4 Non determinism
The loop labelled with swap_priority() manages the question of non determinism. The C-like function swap_priority() swaps two functions with a same priority at time t. This transition is not necessary for gLLF thanks to theorem 1.

THEOREM 1. [Simulation is sufficient for gLLF] If an execution satisfies the real-time constraints, then any executions respects the constraints. This entails that a simulation is sufficient as a schedulability test for gLLF.

Proof. Let us consider a task set \( \tau_i, i = 1..n \) which is successfully scheduled on \( p \) processors by a gLLF simulation. Let us assume that at some point \( SP_1 \), there are \( q \leq p \) processors available for \( r \) tasks with a same laxity with \( r > q \). There is a conflict and the scheduler has to make a choice of \( q \) tasks among \( r \).

Since the simulation is successful, it implies that at some future scheduling point \( SP_2 \) all the \( r \) tasks, which were in conflict in \( SP_1 \), ended correctly. This means that between \( SP_1 \) and \( SP_2 \) there is enough CPU room for all tasks. Since gLLF computes at each step the priorities, and gives the CPUs to the highest priority tasks, there are \( r \) paths composed of segments of length one time unit from \( SP_1 \) to \( SP_2 \). For another simulation run, the \( r \) tasks will use differently the segments, and will finally reach \( SP_2 \) correctly too. Thus choosing any \( r \) tasks at \( SP_1 \) will not change the result that at \( SP_2 \) the \( r \) tasks ended.

2.3 Taking into account the scheduling policy
In this part, we detail how to compute the array list_prio. This fully depends on the scheduling policy. We use a C-like function which generates a deterministic ordering of list_prio. This function combined with the swapping transitions produces all the combinations of ordering.

2.3.1 Computation of list_prio
The C-like function update_prio, when called, creates the new list by looking on the awaken tasks. The first awaken task to be encountered is stored at the first place. The next ones are stored after the already stored tasks with a higher or equal priority. For FP, list_prio is constant, for gEdward, the priority is sorted by the deadline, for gLLF, the priority is sorted by the laxity \( l_i = D'_i - C_i \). The case of LLREF is a bit more complicated.

2.3.2 For LLREF
We first need to compute the next time where there will be some modification in the configuration. This occurs when a new job is released or when a deadline expires. At the instant \( t \), \( nexttime = \min_{\tau_i} \{nexttime_{\tau_i} \} \) with

\[
nexttime_{\tau_i} = \begin{cases} O'_i & \text{if } O'_i > 0 \\ D'_i \text{ else } D'_i > 0 \\ T'_i & \text{otherwise} \end{cases}
\]

The condition \( n_r \) depends on the values of the two parameters required in the LLREF policy:

- the local remaining execution time in eq 1 can be rewritten with our variables:
  \[
l_r = \begin{cases} 0 & \text{if } O'_i > 0 \\ C'_i \times \frac{nexttime}{D'_i} & \text{otherwise} \end{cases}
\]
- the local laxity in eq 2 can be rewritten as:
  \[
l_r = nexttime - l_r
\]

These two parameters take their values in \( \mathbb{Q} \). Unfortunately, UPPAAL is not able to handle efficiently float or rational variables. We have decomposed each parameter into two parts: the integer quotients and the rest of the euclidean division. Let us denote by \( \delta_{cond} \) the function which is equal to 1 if \( cond \) is true and 0 otherwise. We obtain the 3 parameters:

\[
\begin{align*}
l'_r &= (\delta_{O'_i > 0} \times C'_i \times nexttime)/D'_i \\
l'_r &= (\delta_{O'_i > 0} \times C'_i \times nexttime) \mod D'_i \\
L'_r &= nexttime - l'_r
\end{align*}
\]

We do not need the parameter \( L'_r(t) \) to decide the priority. A task is urgent when \( L_r(t) = 0 \) and in that case \( L'_r(t) = 0 \) too.

EXAMPLE 2. Let us consider again example 2. Applying the LLREF strategy on this example proves that the task set is schedulable and leads to the execution below:

<table>
<thead>
<tr>
<th>date</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>nexttime</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>nextwakeup</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>list_prio</td>
<td>{0, \ldots }</td>
<td>{2,1,0}</td>
<td>{2,1,0}</td>
<td>{2,0,1}</td>
</tr>
<tr>
<td>{0, \ldots }</td>
<td>(0,3,1)</td>
<td>(1,3,0)</td>
<td>(1,3,0)</td>
<td>(1,0,1)</td>
</tr>
<tr>
<td>{2,1,0}</td>
<td>(5,2,0,5,0)</td>
<td>(4,1,0,4,3)</td>
<td>(5,1,0,3,1)</td>
<td>(2,1,0,3,1)</td>
</tr>
<tr>
<td>{2,1,0}</td>
<td>( \ast )</td>
<td>(1,3,0)</td>
<td>(6,3,3)</td>
<td>( \ast )</td>
</tr>
<tr>
<td>{2,0,1}</td>
<td>(5,2,0,5,0)</td>
<td>(4,1,0,4,3)</td>
<td>(3,0,0,3,1)</td>
<td></td>
</tr>
<tr>
<td>{2,0,1}</td>
<td>( \ast )</td>
<td>(4,0,0)</td>
<td>(3,0,0)</td>
<td>(2,0,0)</td>
</tr>
<tr>
<td>{2,0,1}</td>
<td>(5,5,1,5,0)</td>
<td>(5,5,0,5,0)</td>
<td>(4,4,0,4,3)</td>
<td>(3,3,0,3,1)</td>
</tr>
</tbody>
</table>

The variable nextwakeup that intervenes in the necessary condition is computed:

\[
nextwakeup = \min_{\tau_i} \{l'_r + \delta_{O'_{\tau_i} > 0} \}
\]

3. PERFORMANCE EVALUATION
3.1 Automatic conversion

We have developed a prototype named converter which converts a simple text description of a task set. Example 1 can be described by:

```
# Example1 task set
# Task "Name" T C D O
Task "Tau0" 5 2 5 0
Task "Tau1" 5 2 5 1
Task "Tau2" 5 5 5 1
```

The converter takes several command line arguments: the file name containing the task set description, the number of cores, the scheduling policy (FP, gEDF, gLLF, LLREF). The source code is around 3000 lines of portable C code. The converter is part of the wider software kit which contains building blocks for embedded realtime middleware consisting of offline and online tools. Thus part of this work has been done by the authors in the context of the SATRIMMAP project. In order to validate the programming, we have compared several test cases on our Uppaal models and NuSMV models (extension of those of [12]).

3.2 Task set generator

We have also included an automatic task set generator with varying parameters: the number of task sets to generate, the number of tasks in each set, the geometric progression of period (r), the period limit (l) and the WCET ratio (w). The generator then produces task sets such that:

\[
T = l \times (1 + (\text{rand}() \mod r)), \text{if } r > 0
\]

\[
T = 1 + (\text{rand}() \mod l), \text{if } r = 0
\]

\[
D = T
\]

\[
O = \text{rand}() \mod D
\]

\[
WCET = 1 + ((\text{rand}() \mod (D - O))/w)
\]

where \text{rand}() gives 32 bit integer random numbers (our current Linux implementation uses /dev/urandom).

3.3 Experiments

The experiments were run on a 64 bits Debian Linux host (Intel Xeon X5472 @3.00GHz) with 16GB of DDR2 memory and Uppaal v4.0.11 binary distribution. We applied the task set generator in order to evaluate more than 10000 task sets. After numerous experiments, the two determinant factors were the number of tasks and the lcm of the periods.

The Fig. 3 describes the impact of the lcm for a fix number of tasks (50). We easily reach lcm = 35000 (except for LLREF). The time is linear for deterministic models (FP and gLLF) while it grows linearly for gEDF. The time of computation consumed and exponentially for LLREF.

The Fig. 4 describes the impact of the number of tasks. The time of computation is growing linearly w.r.t. the number of tasks for FP and gLLF such that we may compute the schedulability of 1000+ tasks which is far better than [11, 12] who stop their experiment with 20 tasks and 3000 seconds. Private discussions with one author of [6] confirms that they cannot currently target the 1000+ tasks range with their method either, however we cannot really compare with them because we do not handle dependencies nor resource sharing.

Currently our limits seem driven by two intrinsic Uppaal limits:

1. Uppaal is a 32 bits application which cannot use more than 4GB of memory which limits the number of states that can be stored. Something around 3000000 states for the gEDF Uppaal model.

2. the Uppaal integer range is [-32768, 32767] (16 bits values) which limits the maximum value of task period.

The time of computation is growing exponentially for gEDF and LLREF probably because of exponential state growth due to priority choice for those algorithms. Indeed, we have also generate deterministic gEDF which performances are equivalent to those of fixed priority. Compared to [11, 12] or [6], our results are much better in terms of performance on deterministic representation and moreover neither of them treat the non determinism. Note however that non-determinism may be avoided if it is possible to enforce task ordering with another key like their unique process ID.

Our handling of non-determinism reveals odd results. Sometimes, very similar task sets exhibit very different behavior. In particular, we have found 2 task sets with 97 tasks scheduled on 2 processors with similar parameters and different periods. The taskok is schedulable with gEDF and Uppaal explores of 38089 states. The taskko makes Uppaal fail due to “out-of-memory” error after exploring 3000800 states. The same taskko can be verified without any trouble for FP, gLLF or deterministic gEDF.
We then tried to evaluate the number of priority swaps induced by the non-deterministic gEDF model in both cases. For this, we introduce a meta variable \( cswap \) which does not add extra state. This variable, initialised to 0, is incremented each time the priority swap transition is taken. We then add to our model checking formulas an extra condition of the form \( cswap < 7 \). If the model checker succeeds, we are ensured that no more than 7 priority swaps occurred. It appears that the taskok does not generate more than 7 priority swaps, while the taskko generates at least 100 priority swaps.

4. CONCLUSIONS

The aim of this research was to show that a brute force approach for analysing the schedulability of a task set could be efficient enough for realistic configurations (1000+ tasks), even when considering dynamic scheduling strategies such as gEDF, LLF or LLREF. We have then developed a toolset allowing to automatically generate verification models from task set configurations, and consequently allowing schedulability analysis with UPPAAL. Our main conclusion is that, according to our experimental results, the schedulability analysis becomes tractable on industrial examples.

The next work to develop lies on an extension of the task model adding dependences between tasks (precedence and resource dependences). In that case, we believe that such new constraints will restrict the state space (by preventing certain scenarios), and then will not deteriorate the performance of the analysis method. The second work to explore consists of adding some metrics to the model or the analyser, such as the number of task migrations, of context changes, the minimal and maximal values of jitters... The objective is to go further and to evaluate the cost of a scheduling strategy on a given task configuration.

5. REFERENCES


