A Brute-Force Schedulability Analysis for Formal Model under Logical Execution Time Assumption

PIERRE-EMMANUEL HLADIK, LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France

This article presents a schedulability analysis for real-time systems designed under the Logical Execution Time (LET) assumption. This assumption increases the predictability of real-time systems by separating time events from scheduling events. A toolchain based on the formal language Fiacre combined with the LET assumption is designed to organize a set of tools to model, verify, and generate code. In this context, an exact brute-force schedulability analysis based on a simulation is proposed. The tools and algorithms to manage the computation are described and a speedup is proposed. An experiment on a synthetic system shows the efficiency of this approach.

CCS Concepts: • **Computer systems organization → Embedded systems**; **Real-time system architecture**; • **Software and its engineering → Scheduling**; • **Theory of computation →** *Verification by model checking*;

Additional Key Words and Phrases: Embedded Systems, Formal Verification, Real-Time, Scheduling

ACM Reference Format:
Pierre-Emmanuel Hladik. 2018. A Brute-Force Schedulability Analysis for Formal Model under Logical Execution Time Assumption. 1, 1 (February 2018), 13 pages. https://doi.org/10.1145/3167132.3167199

1 INTRODUCTION

The design of embedded real-time systems requires specific toolchains to guarantee time constraints. These tools need to be managed in a coherent way via the design process and need to deal with the system modeling, verification, and code generation.

This paper presents such an integrated toolchain and focuses especially on the schedulability analysis. The toolchain follows the Logical Execution Time (LET) assumption. This assumption, first introduced by Henzinger *et al.* [8], increases the predictability of a real-time system by separating time events from scheduling events. An indirect advantage is the possibility of also model-checking the behavior of such a system without considering the scheduling, which reduces the risk of a state explosion.

The schedulability analysis presented in this paper makes full use of the LET assumption. The problem is reduced to a schedulability analysis of a concrete system, *i.e.*, a system of periodic tasks with a known offset with multiple activation schemes. The problem is then to compute all the activation scenarios and to efficiently analyze them. To achieve this, the approach introduced in Ref. [7] is followed and a scheduling simulation is used to produce an exact scheduling analysis.

The remainder of this document is structured as follows. Section 2 describes the context, the toolchain, and the LET assumption used for this study. Section 3 presents work relevant to the scheduling analysis. The following section presents the model and assumptions used. Section 5 introduces a process to analyze the schedulability and describes the

Author's address: Pierre-Emmanuel Hladik, LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France, pehladik@laas.fr.

Manuscript submitted to ACM

principal algorithms. Section 6 shows an evaluation of the schedulability analysis on a synthetic example, and finally Section 7 addresses the conclusions and future work.

2 CONTEXT

The schedulability analysis presented in this paper is part of a larger project developing a toolchain called Hippo [9]. This toolchain aims to provide tools to design, verify, and generate code for embedded real-time applications to enforce their temporal safety. The Hippo toolchain is based on a design methodology called MCSE (Méthodologie pour la Conception de Systèmes Électroniques), which was proposed by Calvez [4] during the 1990s. This methodology provides a framework to design embedded systems. It is a straightforward method, dealing with a full development cycle and proposing a domain-specific language with a narrow but nevertheless meaningful syntax. Figure 1 outlines the main steps of the Hippo toolchain.

The verification process of Hippo is based on the toolbox Tina [3] and the formal language Fiacre [2]. Tina is a toolbox to design and analyze Time Transition Systems (TTS), which are time Petri nets with data handling. It includes various tools to construct reachability graphs and to model-check linear temporal logic (LTL) or computational tree logic (CTL) formulas. Fiacre is a formally defined language for modeling the behavioral and timing aspects of embedded and distributed systems for formal verification and simulation purposes. A dedicated compiler, called `frac`, is availed to transform a Fiacre model into a TTS model. Both Tina and Fiacre are documented and can be downloaded from http://projects.laas.fr/tina/ and http://projects.laas.fr/fiacre/, respectively.

The main problem addressed by Hippo concerns the generation of an executable that guarantees that the timing constraints are respected. One difficulty is to avoid a semantic gap between the model produced by the designer, the model used by the model-checker, and the executable [11]. For the Hippo toolchain, we chose to generate a code as close as possible to the TTS formal model. Therefore, during the generation step, a runtime system is used to produce C code that guarantees a control flow that is identical in every detail to the behavior of the TTS model. This approach is similar to the BIP toolset [1].

A second difficulty concerns the control of the time behavior and its verification. Traditionally, two approaches are distinguished (a time-triggered approach versus an event-triggered approach [10]) by the time representation and the instants at which events are considered. However, an intermediate approach, based on the Logical Execution Time (LET) assumption, was introduced with Giotto [8].

Using LET, the system designer specifies the logical execution time of each task, that is, the duration between the instant at which the task is activated and the instant at which the task provides its outputs. When the LET expires, the outputs are made visible for other tasks. This buffering of outputs achieves determinacy in both timing (no jitter) and functionality (no race conditions). LET programming trades code efficiency in favor of code predictability compared

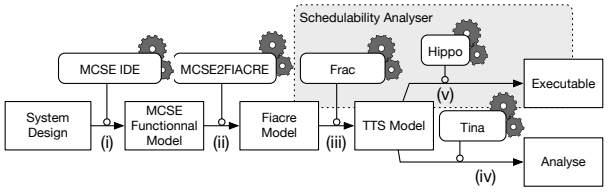
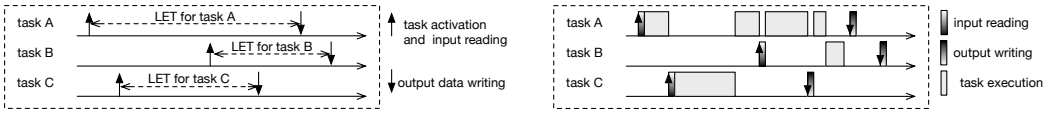


Fig. 1. Main steps of the Hippo toolchain.



(a) LET view (b) Scheduling EDF view

Fig. 2. Timing diagram for three tasks under the LET assumption.

to traditional task scheduling, which makes all outputs visible as soon as they become available. Figure 2(a) shows examples of LET for three tasks.

To respect the LET assumption, the code generator needs to make sure that, on a specified platform, all the outputs are computed in time. This can be handled by a code composed of multiple tasks to ensure concurrent execution and a real-time operating system thereby leveraging the strengths of real-time scheduling during execution. Figure 2(b) shows an example where an EDF scheduler is used to guarantee that the outputs are computed in time. Under the LET assumption, the scheduling can be ignored to model-check the behavior of the application. However, during the code generation step, the compiler needs to check the schedulability of the task. This paper addresses this problem.

The transformation from an MCSE Model to a Fiacre Model is not the focus of this paper and is therefore omitted in the following sections. A prototype for the TTS Execution Engine is implemented on Xenomai (https://xenomai.org).

3 RELATED WORK

Numerous methods to analyze the schedulability have been developed since the initial work of Liu and Layland [13]. A large portion of these methods uses analytical expressions to produce a schedulability test. The advantages of these approaches are their ability to deal with various behavior task models and their often low computational complexity. However, it may be difficult to extend these tests to non-usual behavior and they are often non-optimal for multiprocessor architectures.

Other studies use formal methods to verify the schedulability of a system. For example, Lime *et al.* [12] modeled a preemptive scheduler with a scheduling time Petri net, and Peres *et al.* [14] proposed a formal language to model a scheduled system. These approaches have the advantage of directly modeling the scheduling algorithm with a behavior model; however, a serious issue with these methods is the state space explosion, which drastically limits the number of tasks that can be considered. Moreover, the preemption induces undecidability. To overcome these difficulties, some researchers, such as Cordovilla *et al.* [6], propose modeling the task behavior with a formal model and the scheduler with an ad-hoc code.

Goossens *et al.* [7] proposed another approach to analyze the schedulability of a multiprocessor system. They show that, under certain hypotheses, it is possible to use a simulator to conduct an exact schedulability test. Our paper is a continuation of this work.

There are several tools dedicated to the simulation of real-time systems such as Cheddar, YASA, TORSCHE, MAST, Storm, and SimSo. Most of these tools are designed to validate, test, and analyze systems. SimSo [5] is the most advanced and modular tool focusing on the study of the scheduler itself.

In this paper, we chose to use a formal method to model-check the behavior of a system without considering the schedulability and to analyze the schedulability with a dedicated algorithm combined with a simulator. The aim of this

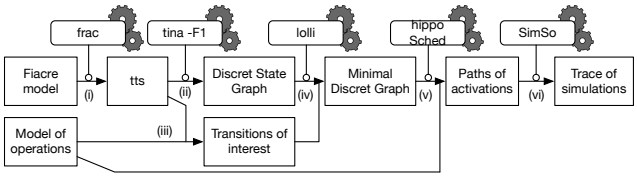


Fig. 3. Toolchain for the schedulability analysis.

approach is to avoid a state space explosion with formal methods while maintaining an exhaustive verification of the system's behavior.

4 MODEL AND HYPOTHESIS

The toolchain used to conduct the schedulability analysis is sketched in Figure 3. The main steps are:

(i) the Fiacre model is compiled to a TTS Model via the compiler `frac`;
(ii) (ii) the state space is generated by the tool Tina (as explained in the next section, the state graph is obtained by firing integer unit delay transitions and discrete transitions);
(iii) the elements formally described in the TTS are linked with scheduling time parameters of tasks such as the execution time and deadline;
(iv) the tool `lolli` reduces the state space graph to a minimal determinist automaton with only transitions that represent a unit delay or a task activation;
(v) all sequences of task activations are computed for a feasibility interval; and
(vi) each sequence is run with a scheduling simulator and each deadline is checked.

The next subsections describe each step in detail, especially step (v). The tools that support steps (i), (ii), and (iv) are available with the toolbox Tina; therefore, only the input and output models will be presented below. An alternative toolchain is also proposed in Section 5.4 via embedding the scheduling simulation into the search for the sequences of activations, *i.e.*, steps (v) and (vi) are merged.

The simulator used is SimSo [5]. It is a discrete event simulator used to evaluate real-time scheduling algorithms (mono or multiprocessor). It is only used in this study to generate the scheduling of a sequence of task activations and to check the deadlines.

From a general perspective, this approach follows the idea introduced by Goossens *et al.* [7] to produce an exact schedulability test based on simulations. The hypotheses of the scheduling algorithm used to conduct this test are exposed in the following. The purpose of this paper is to propose a practical solution to proceed any behavioral model of the execution described in the specialized Fiacre for MCSE.

Fiacre model. The Fiacre [3] language is used to model the behavior and timing aspects of systems. This language is composed of parallel processes communicating via ports and shared variables. A process describes the behavior of sequential components and is defined by a set of control states, each associated with a program segment built from classical deterministic constructs (assignments, if-then-else conditionals, while loops, and sequential compositions), communication data-events on ports, and jumps to the next state. To consider MCSE behavior and the LET assumption, a specialization of Fiacre is performed.

```
1  process pExample is
2  [...]
3    argOp := arg; /* read data */
4    wait[0,0] /* trigger operation */
5    to BeginOp
6  from BeginOp
7    res := Op (argOp);/* operation */
8    wait[12,12]; /* LET */
9    to EndOp
10 from EndOp
11 [...]
12 pExample
```

Listing 1. Pattern to model the operation Op with a logical execution time.

From a temporal perspective, we assume that all instructions and transitions in a Fiacre model take zero execution time. The timing is modeled via the `wait` statement, which represents a delay. This statement is associated with an interval $[a, b]$ and signifies that the control state needs to wait a duration between a and b before forwarding. For our purpose, only punctual intervals with integer numbers are permitted, *i.e.*, `wait[a, a]` with `a` $\in \mathbb{N}$. This restriction ensures that the behavior of the system is deterministic and induces a discrete time.

The LET assumption is associated with the functional treatments, which are embedded into functions called operations. An operation is basically a C function with input and output data. These operations are executed in Hippo via tasks. Each operation has its own task and is scheduled by the operating system.

Listing 1 shows the pattern used in Fiacre to model an operation under LET. The input data are read at line 3, and activation is triggered at line 4. The usage of the statement `wait [0,0]` forces the transition to occur immediately, *i.e.*, the inputs are immediately read. The call of the operation `Op` is represented in line 7. This function is the treatment realized by the operation. The LET assumption is modeled by the value of the `wait` statement in line 8. In Fiacre, a value is only updated during a transition, so that the value of `res` is made visible, *i.e.*, the outputs are provided, only when the delay (line 8) is expired.

The Fiacre model offers the possibility of writing any type of activation pattern for the operations: the periodicity can be modeled with a simple delay or a precedence relation with synchronization via a port or condition. The Listing 2 gives an example of a periodic behavior. Thanks to the `wait` statement line 7, the process `clock1` is periodic. The process `p1` is synchronized with this periodic clock through the label (i.e. port in Fiacre terminology) `tick1`. It means that the transition `start` from the process `clock1` is synchronized with the transition `waitPeriodic` of `p1`. The composition is done with the component `cExample` (line 17) where the `par` statement is used to synchronized ports of different processes.

The task behavior does not need to be characterized; it is implicitly described by the model.

TTS Model. The Fiacre model is compiled to a TTS Model with the `frac` compiler. A TTS model is a time Petri net with data handling and guards described in a C file.

We do not need to consider data in the schedulability analysis; therefore, we do not present how to manage them here. This is done by the compiler during the code generation at the runtime.

Temporal characteristics of an operation. Two temporal characteristics are associated with an operation: its execution time and its deadline. These are described in an xml file. It is possible to extend these characteristics to add information for the scheduler, for example, a priority level. The scheduling algorithm is also described by the name of the scheduling policy in SimSo.

```
process clock1 [tick1 : none] is
  states start, order
  from start
    tick1;
    to order
  from order
    wait [period,period];
    to start

process p1 [tick1 : none] is
  states waitPeriodic, ...
  from waitPeriodic
    tick1;
[...]
    to waitPeriodic

component cExample is
  port tick1 : none in [0,0],

  par * in
  clock1           [tick1]
    || p1          [tick1]
end
```

Listing 2. Pattern of a periodic activation.

Scheduling policy. The scheduling policy used by the operating system to execute tasks is not stated in the behavior model. The scheduler policy only needs to be known during the simulation and is therefore directly implemented in SimSo.

There are no restrictions on the scheduler (*e.g.*, uniprocessor, multiprocessor, global, or partitioned); however, as Goossens *et al.* [7] show, only deterministic and memoryless schedulers need to be considered. A memoryless scheduler is a scheduler "for which the scheduler decision depends only on the state of the system at the current instant" (see Definition 1 in Ref.[7] for a more formal definition). Moreover, in the same article [7], the authors specify that simulation-based schedulability tests can only be used when the context is such that the simulation is *C*-sustainable. "A schedulabitlity test is *C*-sustainable is a system deemed schedulable when tasks are using their WCET is schedulable even if some tasks do not use up to their WCET".

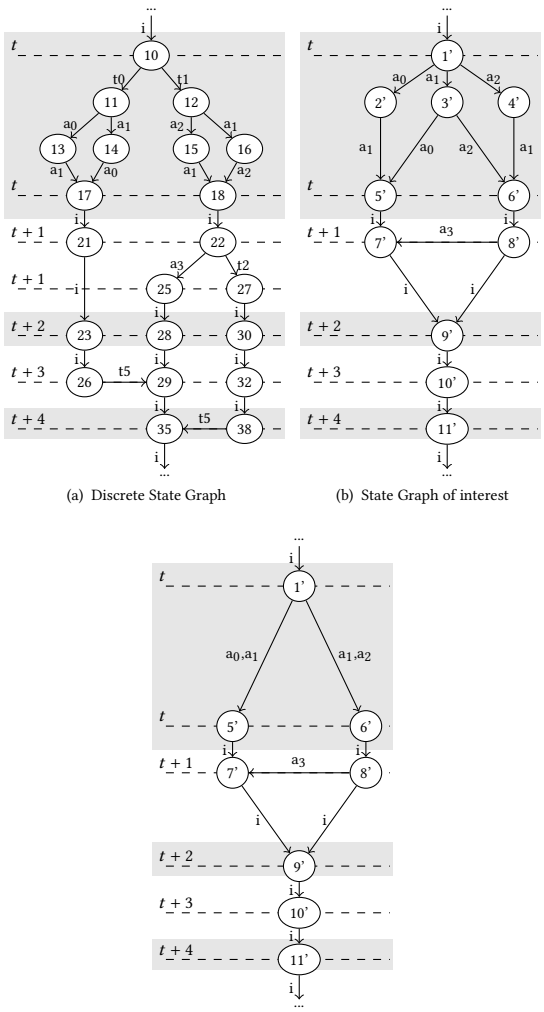
In practice, the Hippo runtime is based on industrial operating systems and, therefore, only the Fixed Priority scheduler and Earliest Deadline First have been used (with the global version for the multiprocessor platform). These two schedulers are deterministic, memoryless, and *C*-sustainable when the tasks are independent. In the case of a multiprocessor architecture, only identical multiprocessor platforms are considered.

Moreover, to conduct the schedulability analysis, it is necessary to know the simulation interval. This interval is related to the notion of the feasibility interval, *i.e.*, a finite interval $[a, b]$ such that if all the deadlines of jobs released in the interval are met, then the system is schedulable. Readers are invited to refer to the comprehensive overview on simulation intervals in Ref. [7].

5 SCHEDULABILITY ANALYSIS

5.1 Computation of the Discrete State Graph

The tool Tina (step (ii) in Fig. 3) is used with option `-F1` on the TTS model to build a subgraph of the state graph obtained by firing integer unit delay transitions and discrete transitions. This graph preserves the reachability and linear time temporal properties. Figure 4(a) shows an example of part of a graph computed using this tool. The edges labeled with `"i"` represent a unit time, and other edges are internal events in the model. Note that all timed values (delays) are integers; therefore, a graph with discrete transitions has the same behavior as the initial model.



(a) Discrete State Graph (b) State Graph of interest

(c) Reduced State Graph

Fig. 4. Examples of graph refinement.

Fig. 5. Example of a state graph obtained with `lolli`.

For the schedulability analysis, only events linked to the activation of an operation need to be considered and other non-timed transitions can be considered to be silent transitions (the set of events linked to an activation can be automatically found using the Fiacre model). For example, a transition not-related to an activation can be a transition used to model a conditional statement in a process.

By considering the state space graph to be an automaton, it can be reduced to the set of transitions linked to the activation of an operation. To achieve this, the tool `lolli` was developed by Bernard Berthomieu at LAAS-CNRS. It replaces transitions that are not in the set of interest with ϵ-transitions and determines and reduces this new graph. In other words, the graph produced by `lolli` represents all timed sequences of activations that can occur in the system and ignores other events. Figure 4(b) shows the result for the graph presented in Figure 4(a), where only a_1, a_2, and a_3

are transitions related to an activation. Other transitions are not linked to an activation and therefore are not treated by the schedulability analysis.

5.2 Computation of sequences of activations

The execution of a system can be described by the set of timings of activations of operations. This set is called a sequence of activations.

Definition 5.1. By denoting $\langle op_i, t\rangle$ the activation of an operation op_i at an instant t, a sequence of activations is defined as an ordered set of activations $\{\langle op_i, t_1\rangle, \langle op_j, t_2\rangle, ..., \langle op_k, t_3\rangle\}$ with $t_i \leq t_{i+1}$.

5.2.1 Computing the sequence from a path. We denote the directed graph obtained with `lolli` to be $G = (V, E)$, where V is the set of vertices and E is the set of directed edges that connects an ordered pair of vertices.

The set of edges of unit time transitions is denoted C, *i.e.*, the set of transitions labeled "`i`", and the set of edges that represents a transition related to an activation is denoted $\mathcal{A}$. The sets $\mathcal{A}$ and C are a partition of E: $E = \mathcal{A} \cup C$ and $\mathcal{A} \cap C = \emptyset$. An edge e in $\mathcal{A}$ is linked to a set of operations activated during the transition e; this set is denoted Op_e.

For a directed path $p = e_1 e_2 ... e_n, e_i \in E$ (to simplify, we omit the vertices in the path) from a graph $G = (V, E)$, the sequence of activations $s(p)$ can be computed with an iterative expression:

$$
\begin{aligned}
& t_0 = 0 \\
& s_0 = \emptyset \\
& \text{if } e_i \in C \quad \begin{cases} t_i = t_{i-1} + 1 \\ s_i = s_{i-1} \end{cases} \\
& \text{otherwise} \quad \begin{cases} t_i = t_{i-1} \\ s_i = s_{i-1} \bigcup_{o_j \in Op_{e_i}} \{\langle o_j, t_i\rangle\}. \end{cases}
\end{aligned}
\tag{1}
$$

5.2.2 Equality between two sequences. If two operations op_1 and op_2 are activated at the same time t, then for the sequences $s_1 = \{s_{start}, < op_1, t >, < op_2, t >, s_{end}\}$ and $s_2 = \{s_{start}, < op_2, t >, < op_1, t >, s_{end}\}$, the order of the releases of the operations is chosen by the scheduling policy; therefore, these sequences produce the same scheduling (if the scheduler is deterministic). From a schedulability perspective, we consider s_1 and s_2 to be equivalent.

Definition 5.2. Two sequences of activations s_1 and s_2 are said to be equal, denoted $s_1 = s_2$, iff $\forall a = \langle o_j, t_i\rangle \in s_1$, a is also in s_2 (and vice versa).

From a discrete state graph computed using `lolli`, the computation of all sequences can produce multiple equal sequences, *e.g.*, in the example depicted in Figure 4(b), the sequence $s(p_1) = \{\langle a_0, 0\rangle, \langle a_1, 0\rangle\}$ from the path $p_1 = (3, 4)(4, 7)$ is equal to the sequence issued from the path $p_2 = (3, 5)(5, 7)$, $s(p_2) = \{\langle a_1, 0\rangle, \langle a_0, 0\rangle\}$.

To speed up the schedulability analysis of a system, it is of interest to not consider multiple sequences and to avoid equal ones.

5.2.3 Equivalence and minimality. For a graph $G = (V, E)$ and a vertex $v \in V$, we denote $P(G, v)$ to be the set of all simple paths from v (a simple path refers to a path that contains no repeated vertices).

Definition 5.3. G_1 is equivalent at G_2, denoted as $G_1 \sim G_2$, from $v_1 \in V_1$ and $v_2 \in V_2$ iff $\{s(p) \mid p \in P(G_1, v_1)\} = \{s(p) \mid p \in P(G_2, v_2)\}$.

Definition 5.4. A graph G is said to be minimal from a vertex v, if all paths from v produce different sequences of activations:

$$\forall (p_1, p_2) \in P(G, v)^2, p_1 \neq p_2 \implies s(p_1) \neq s(p_2)$$

.

5.2.4 Graph reduction. It is possible to compute a minimal graph equivalent to a graph $G = (V, E)$ from an initial vertex. The details of the algorithms are not given here. Only the main steps are described.

First step: The reduction algorithm begins computing all subgraphs $G_i = (V_i, E_i)$ of G that have connected components such that:

(i) all edges in E_i are in $\mathcal{A}$,
(ii) all other edges in E connected to V_i are in C, and
(iii) all vertices in V connected to a edge in E_i are in V_i.

Therefore, a subgraph G_i describes a set of activations occurring between two unit times, *i.e.*, in $[t, t+1)$. In the example depicted in Figure 4(b), we have two subgraphs G_1 = ({1', 2', 3', 4', 5', 6'}, {(1',2'), (2',5'), (1',3'), (3',5'), (3',6'), (1',4'), (4',6')}) and $G_2 = (\{7', 8'\}, \{(8', 7')\})$.

Moreover, for each subgraph $G_i = (V_i, E_i)$, the set I_i of vertices v in V_i with an edge in C that points to v is computed and the set of vertices v in V_i with an edge in $\mathcal{A}$ that points from v is denoted O_i. For the example depicted in Figure 4(b), for G_1, we have I_1 = {1'} and O_1 = {5',6'}.

Second step: The algorithm computes all paths from vertices in I_i to O_i for a subgraph G_i. For each path p, the set of activated operations $\{o | o \in Op_e, \forall e \in p\}$ is computed and saved only if another path does not have (i) the same initial vertex, (ii) the same final vertex, and (iii) the same set of activated operations. Therefore, for each subgraph G_i, we have a minimal list of paths, $l_i = [p_{i,1}, p_{i,2}, ...]$, where each path p is described by (i_p, f_p, s_p) with i_p as an initial vertex, f_p as a final vertex, and s_p as a set of operations. In the example depicted in Figure 4(b), for the subgraph G_1, the minimal list is $l_1 = [(1', 5', \{a_0, a_1\}), (1', 6', \{a_1, a_2\})]$.

Third step: The algorithm computes a new graph $G_{\min}$ by replacing all subgraphs G_i in G with edges linked to a set of operations $\{e = (i_p, f_p), Op_e = s_p \mid p \in l_i\}$. In this example, we obtain the new graph given in Figure 4(c).

Fourth step: It is possible that two edges of $G_{\min}$ with different initial nodes have the same set of operations, and therefore a more compact representation of the graph may exist. Therefore, in the last step, the new graph $G_{\min}$ is considered to be an automaton where each set of Op_e is associated with a symbol. This step uses the classic Hopcroft's algorithm to reduce the automata.

5.2.5 Equivalence and minimality of the reduction algorithm.

Theorem 5.5. *The graph $G_{\min}$ obtained with the reduction algorithm is equivalent to G.*

Proof. According to the construction of $G_{\min}$, for all parts $p' = e_i e_{i+1} ... e_{i+k}$ of a path $p = n_1 e_1 n_2 e_2 n_3 ... e_n n_{n+1}$ from G such that $e_m \in \mathcal{A}, m \in [i, i+k]$ and $(e_{i-1}, e_{i+k+1}) \in C^2$, there exists an edge $e = (n_i, n_{i+k+1})$ in $G_{\min}$ such that $s(e) = s(e_i e_{i+1} ... e_{i+k})$. Conversely, the algorithm ensures that all activation sequences from a path of $G_{\min}$ exist if there exists a path with the same sequence in G. □

Theorem 5.6. *The graph $G_{\min}$ obtained with the reduction algorithm is minimal.*

Proof. The automaton minimization at the end of the reduction algorithm guarantees that a word accepted by the automaton has a unique path, and therefore the graph is minimal in the sense of the definition 5.4. □

5.3 Computation of the schedule traces

The reduction algorithm obtains a minimal graph that represents all the possible sequences of activations for a given system. To compute all the paths, *i.e.*, all sequences of activations that the system can experience, a depth-first search algorithm can be used combined with a stop condition based on the duration of the path, *i.e.*, the algorithm is stopped when t_i from equation (1) reaches a fixed value. Accordingly, how can this duration be set to guarantee that all execution scenarios have been checked?

Note that all graphs have a shape of *lollipop* (or *racket*), as shown in Figure 5. This means that the system has an initial phase followed by periodic behavior. Therefore, if each operation is considered to be an independent task with a period equal to the periodic behavior of the "lollipop" and an offset equal to its first instance of activation, the system can be considered to be a concrete periodic system and therefore the classical results depending on the scheduling policy can be used to determine a simulation interval [7].

After enumerating all the activation sequences, this sequence is translated into the input format for the scheduling simulator SimSo. Note that any scheduling policy can be chosen with the simulator; however, the scheduler needs to be C-sustainable to ensure the faisability of the analysis [7].

5.4 Embedded scheduling policy

It is possible to adaptively check the schedulability by computing the scheduling directly while traversing the minimal activation graph. The details of this algorithm are not provided here.

The main idea is to compute the state of the scheduler and to check the schedulability at each step when the paths are computed. The description of a state depends on the scheduling algorithm, for example, for EDF, the remaining work for each job and the absolute deadline are sufficient information to compute the next state.

A simple depth-first search algorithm is therefore used to compute the new scheduler state and check the schedulability of all paths from each vertex. If all operations have finished, the algorithm reaches the next vertex with an out edge not in C. This new vertex is then marked as a source, and a new computation from this vertex is released, but only if the vertex has never been used as a source before. Note that only memoryless scheduling is considered; therefore, all past events that lead to a state can be forgotten and, if this state is met again the future, it will be exactly the same.

This algorithm ends when all paths are explored. Its termination is guaranteed by the finite number of vertices. This algorithm has two major advantages on the toolchain with SimSo: (i) the simulation duration is computed by the algorithm, *i.e.*, the algorithm is stopped when all activation sequences have been explored, and (ii) the number of paths to explore is reduced thanks to the cuts made when all the remaining work has been treated. However, the main drawback is the necessity to write a specific procedure to update the scheduler state for each scheduling policy.

6 EXPERIMENTS

The toolchain is implemented in ML for `frac`, Tina, and `lolli`. SimSo is implemented in Python, as are the other tools. The NetworkX package (https://networkx.github.io) is used to manipulate the graphs.

To show the efficiency of the approach, an experiment was conducted on a synthetic benchmark. This example is described in Figure 6 and is composed of seven execution flows. Each flow is periodically activated and triggers the activation of operations. In Figure 6, the uplets under the clocks represent the offset, *i.e.*, the first activation, and the

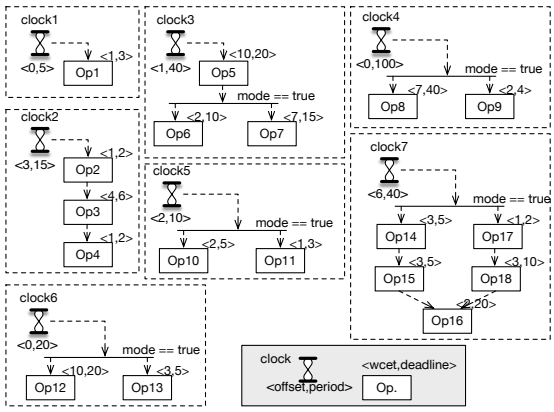


Fig. 6. Synthetic example for the experiments.

period of the flow. An operation is represented by a rectangle and an uplet with its WCET and deadline. An arrow between two operations indicates that the second operation is activated when the deadline of the previous one is reached. A vertical line represents a conditional synchronization and indicates that an operation is activated only if the condition is verified. For example, operation 9 is activated only if the variable `mode` is true, otherwise operation 8 is activated.

Note that the value `mode` is shared by various flows and can be thought of as a mode-change. Here, the change can happen at any time but only once.

By considering the conditional cases, the maximum utilization factor, *i.e.*, the sum of the WCET operations divided by the period, is 185%; therefore, a minimum of two processors is necessary to schedule this system. For this study, the scheduler was a Global EDF on 3 processors.

The model of this system is written in Fiacre with the pattern given in Section 4 for the operations. The clocks, conditions, and the mode-change are also modeled with the Fiacre language. The Listing 3 gives an example of Fiacre code for the `clock4` and the flow 4. The WCETs are specified in a dedicated xml file (see Section 4).

The experiments were performed on a 2.6 GHz Intel Core i5 with 8 GB of memory. The state graph obtained with Tina has 31,125 states and 72,139 transitions. The discrete graph produced by `lolli` has 2244 nodes and 2834 edges, and the reduced graph has only 1780 nodes and 1870 edges. The computation time is 1.03 s with Tina and 1.89 s with `lolli`. The transition phase of the "lollipop" has a duration of 22 units of time and a period of 600 (equal to the lcm of the periods of all the flows).

The number of paths explored with the brute force method coupled with SimSo is 282. The computation time to compute all the activation sequences is 0.89 s, and the simulation of all the sequences with SimSo is 75.8 s. All deadlines are respected.

```
process clock4 [tick4 : none] is
  states offsetState, startState, orderState

  from offsetState
    wait[0,0]; /* offset: 0  */
    to startState
  from startState
    tick4; /* synchronization event */
    to orderState
  from orderState
    wait [100,100]; /* period: 100  */
    to startState

process p4 [tick4 : none](&mode : nat) is
  states waitState, readMode, updateInputDataOp8, computeStateOp8,
       updateInputDataOp9, computeStateOp9
  var i : nat := 0, arg : nat := 0, tmp : nat

  from waitState
    tick4;
    to readMode
  from readMode
    tmp := mode;
    wait[0,0];
    if tmp = 1 then
      to updateInputDataOp8
    else
      to updateInputDataOp9
    end
  from updateInputDataOp8
    arg := i;
    wait [0,0];
    to computeStateOp8
  from computeStateOp8 /* Op8 : deadline 40 wcet 7 */
    i := Op8(arg);
    wait [40,40];
    to waitState
  from updateInputDataOp9
    arg := i;
    wait [0,0];
    to computeStateOp9
  from computeStateOp9 /* Op9 : deadline 4 wcet 2 */
    i := Op9(arg);
    wait [4,4];
    to waitState
```

Listing 3. Fiacre model excerpt of the synthetic example.

The method with the scheduling policy embedded in the tool explores 269 sequences with a mean size of 42 operations in 1.2 s.

This experiment shows the feasibility of the method and that it is possible to proceed to an exhaustive schedulability analysis with the simulation tool. Note that the use of SimSo increases the duration of the analysis; however, it is easy to experiment with various schedulers on the same system. Here, the example is not schedulable with two processors. The simulation tools can easily give a trace where a task do not respect its deadline.

7 CONCLUSION

In this article, a schedulability analysis was presented for a system under the LET hypothesis. This hypothesis has the advantage of clearly separating the temporal analysis of the application behavior from the scheduling. The revealed solution is based on the work of Goossens *et al.* [7], and a scheduling simulation was used to produce an exact schedulability test. Two versions were proposed. The first one uses the notion of the feasibility interval of a concrete task system, and the second adaptively enumerates all scheduling scenarios.

A complete implementation was carried out, and experiments were conducted on a synthetic system. The results demonstrate the usability of this method.

REFERENCES

[1] Tesnim Abdellatif, Jacques Combaz, and Joseph Sifakis. 2010. Model-based implementation of real-time applications. In *Proc. of ACM International Conference on Embedded Software (EMSOFT)*.

[2] Bernard Berthomieu, Jean-Paul Bodeveix, Patrick Farail, Mamoun Filali, Hubert Garavel, Pierre Gaufillet, Frederic Lang, and François Vernadat. 2008. Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. In *Proc. of the Embedded Real Time Software (ERTS)*.

[3] Bernard Berthomieu, Pierre-Olivier Ribet, and François Vernadat. 2004. The tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *International Journal of Production Research* 42, 14 (2004).

[4] Jean-Paul Calvez. 1993. *Embedded Real-Time Systems – A Specification and Design Methodology.* Wiley.

[5] Maxime Chéramy, Pierre-Emmanuel Hladik, and Anne-Marie Déplanche. 2014. SimSo: A Simulation Tool to Evaluate Real-Time Multiprocessor Scheduling Algorithms. In *Proc. of the 5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*.

[6] Mikel Cordovilla, Frédéric Boniol, Eric Noulard, and Claire Pagetti. 2011. Multiprocessor Schedulability Analyser. In *Proc. of the 26th Annual ACM Symposium on Applied Computing (SAC)*.

[7] Joël Goossens, Emmanuel Grolleau, and Liliana Cucu-Grosjean. 2016. Periodicity of real-time schedules for dependent periodic tasks on identical multiprocessor platforms. *Real-Time Systems* (2016).

[8] Thomas Henzinger, Benjamin Horowitz, and Christoph Kirsch. 2003. Giotto: a time-triggered language for embedded programming. *Proc. IEEE* 91, 1 (2003).

[9] Pierre-Emmanuel Hladik, Silvano Dal Zilio, Olivier Pasquier, Sébastien Pillement, and Bernard Berthomieu. 2016. Outillage pour la modélisation, la vérification et la génération d'applications temporisées et embarquées. In *15èmes journées Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL)*.

[10] Hermann Kopetz. 1991. Event-Triggered Versus Time-Triggered Real-Time Systems. In *Proc. of the International Workshop on Operating Systems*.

[11] Cédrick Lelionnais, Jérôme Delatour, Matthias Brun, Olivier H. Roux, and Charlotte Seidner. 2014. Formal Synthesis of Real-Time System Models in a MDE Approach. *International Journal on Advances in Systems and Measurements* 7 (2014).

[12] Didier Lime and Olivier H. Roux. 2009. Formal verification of real-time systems with preemptive scheduling. *Real-Time Systems* 41, 2 (2009).

[13] Chang L. Liu and James Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (1973).

[14] Florent Peres, Pierre-Emmanuel Hladik, and François Vernadat. 2009. Specification and verification of real-time systems using the POLA tool. In *Proc. of the 3rd International Workshop International Conference on Verification and Evaluation of Computer and Communication Systems (VECoS)*.