The Generalized Graph Real-Time Task Model

David Doose, Luca Santinelli

The Generalized Graph Real-Time Task Model

1st David Doose
ONERA
Toulouse, France
david.doose@onera.fr

2nd Luca Santinelli
BMW
Germany
Luca.Santinelli@bmw.de

***Abstract*—Safety-critical real-time embedded systems are becoming difficult to model and analyze as they accommodate more and more functionalities, and they rely on more and more complex HW and SW implementations. One of the main difficulty with such systems is to find the right coupling between the task model and the analysis method. Indeed, if the task model is too simple it may add pessimism to the result of the analysis, and it may not be possible to prove the correctness of the systems. If the model is too complex, the analysis may be not efficient enough to be usable; or worst the model may have an incomplete method only or no method at all, so the system cannot be analyzed. Finding, the right couple task model analysis method is a balance between expressiveness and analysis efficiency.**

In this article, we introduce a new graph real-time task model (noted GGTM) which has been used for several years for the deployment and the analysis of different embedded systems. We discuss its expressiveness compared to the literature and demonstrate its advantages. We also, present a decidable and efficient, exact schedulability analysis.

***Index Terms*—Real-Time Systems, Graph Task Model, Schedulability Analysis**

I. Introduction

Today's HW and SW implementations are making safety-critical real-time embedded systems more and more complex. This is especially true with drones and robots, for which there are also the missions and the interactions with the environment to become more diverse and their human interactions implies safety guarantees. In such a complex ecosystem, the timing behavior of task is crucial, since modeling, and analysis apply for assuring the system timing behavior, i.e. determinism; timing analyses and schedulability analysis enforce the timing guarantees.

An expressive model must capture and abstract the complexity of the real system. In the past decade's different real-time task models have been developed. The widely known Liu and Layland task model [1] has been upgraded with new models and conditions that have been introduced to answer real problems of modeling, execution, and analysis. Then the notion of sporadic tasks was introduced [2], [3], followed by the notion of multiframe tasks [4], [5], [6] in which a task is split in a sequence of executions. These two models are subsumed by the Generalized Multiframe Tasks model [7]. All these models have based "list" task modeling in which the task is represented with a sequence of executions. The next structural extension of the tasks model is to take into account the "tree" tasks model, more precisely the DAG task model. The recurring real-time task model [8], [9], [10], noncyclic generalized multiframe tasks [11] and the non-cyclic recurring real-time task model [12], [13] are embedding that. Finally, complex real-time systems composed graph tasks model [14], [15] was introduced [16], [17], [18]. Each previously cited task model came with a specific schedulability analysis, which can determine if the real-time system is schedulable or not.

Another approach to handle a more complex real-time systems is to use other techniques with more general and more powerful models like model-checking. Some theories like timed automata [19] or Timed Petri Nets [20], because they are dealing with time seem well-fitted for this kind of problem. But it has already been shown that are undecidable while deadline with real-time preemptive systems [21], [22]. And even if the decidability can be proven on a specific model the method can still be not efficient enough [23], [24] or incomplete [25].

To analyze more complex real-time systems it is important to have a precise model and a well-fitted analysis. If the task model is too abstract or if the analysis is too simple, then it can not take into account the fineness of the actual behavior of the system. From the schedulability point of view, this means that method is pessimistic (i.e. it indicates that the system is not schedulable whereas a more precise method demonstrates the schedulability). On the other hand, there are model-checking techniques that can approach complex task models but cannot analyze it due to the undecidability problem or the lack of efficiency. In between there is a gray zone that this paper wants to tackle with.

Contribution: We propose a new task model named Generalized Graph real-time Task Model (GGTM), in which the task execution is described by a graph. We discuss the expressiveness of this task model compared to the previously cited ones. Moreover, we propose a non-pessimistic (exact) and decidable analysis method. The complexity and the efficiency of the method will also be discussed and benchmarking is provided to support considerations on analysis and method computation time.

Organization of the paper: In section II we present the GGTM and its real-time execution behavior semantic. Section III details the expressiveness of the model and compares it to the state of the art. Section IV presents the exact analysis method. Section V the schedulability analysis is applied to a simple example first, then a benchmarking analysis is presented to highlight the cost of the analysis method proposed. Section VI is for concluding and future work.

II. GGTM Task model

A. Task model notations

A GGTM system is a set of independent tasks ($\Gamma = \{ \tau_1, \ \ldots \ \tau_N \}$). Each task τ_i has a priority P_i, an affinity cpu_i, and a graph $\mathcal{G}_i$ which describes its internal behavior: $\tau_i = \{P_i, cpu_i, \mathcal{G}_i\}$. Notice that task migration from one processor to another is not allowed in this model. The graph $\mathcal{G}_i$ is composed of a set of vertices $\mathcal{V} = \{v_{i,1}, \ldots, v_{i,n}\}$, an initial vertex v_1, a set of arcs $\mathcal{A} = \{a_{i,1}, \ldots, a_{i,m}\}$ representing the transitions between the vertices. Each oriented arc is a pair of two vertices, its source and its destination $a_{i,j} : \langle a^-_{i,j}, a^+_{i,j} \rangle$. The set of v_i previous (resp. next) vertices in the graph is noted $prev(v_i)$ (resp. $next$):

$$\begin{aligned} prev(v_i) &= \{v_j \mid \langle v_j, v_i \rangle \in \mathcal{A}\} \\ next(v_i) &= \{v_j \mid \langle v_i, v_j \rangle \in \mathcal{A}\}. \end{aligned}$$

Each vertex of $\mathcal{G}_i$ can be either an *execution* vertex or a *wait* vertex. To ease the presentation, if a vertex $v_{i,j}$ is an execution (resp. wait) vertex then it will be noted $e_{i,j}$ (resp. $w_{i,j}$). An execution vertex represents the execution of a specific code of the task. Thus, an execution vertex ($e_{i,j} : \langle c_{i,j}, d_{i,j} \rangle$) is characterized by its worst-case execution ($c_{i,j}$) time and optionally its relative deadline ($d_{i,j}$, or _ if none). A wait vertex is characterized by its clock duration $w_{i,j} : \delta_{i,j}$. A wait vertex means that the task will wait until its internal clock reached the value specified in the corresponding vertex. These notions will be detailed later. The $type$ function is also added to ease the following formula; the type of vertex v_i can be either $Exec$ for an execution vertex or $Wait$ for a wait vertex. Moreover, the GGTM contains a parameter that characterizes the internal clock bound noted K_i. This parameter has an important role during the analysis, thus it will be explained in detail later in this article.

Notice that worst case-executions, deadlines, and clocks are positive (non-strict) values. We can also notice that this model allows for arbitrary deadlines. To simplify the notations, the task indexed will be omitted while considering a single task.

B. Semantic of the task model

The execution of a GGTM τ_i is based on its internal graph model and the interactions of the higher priority tasks. During its execution, a task has exactly one active vertex at a time. The current active vertex of a task changes according to the GGTM arcs. Each task has an internal clock which is set to zero when the system starts. This clock increases according to time and can be reduced while finishing a wait vertex. There is no restriction in the GGTM structure: execution vertex (resp. wait) can be followed by either another execution or a wait.

Execution vertex: While running an execution vertex ($e_{i,j}$) the task is computing its corresponding code, thus the task is consuming 0 to $c_{i,j}$ time processor unit. But this task can be delayed or preempted by others, thus this action can take longer than $c_{i,j}$. During the execution vertex run, the internal clock (I_i) is increased with time. Notice that I_i is still increasing if the task is preempted by another task. At the end of the execution vertex any next vertex $v_{i,k}$ can be selected according to the GGTM arcs.

Wait vertex: A wait vertex ($w_{i,j}$) does not consume CPU time. While entering a wait vertex two cases may occur: 1) if the task internal clock is lower than the wait vertex clock then the task waits until the internal clock value is equal to the vertex clock value; 2) if the task internal clock is greater than or equal to the wait vertex then its internal clock is decreased by the wait clock value and the next vertex is selected.

a) Task system: The behavior of the complete system is the combination of the behavior of each task according to the preemptive fixed priority scheduler in which each task is mapped to a single processor according to its affinity (cpu_i).

Deadline: In our model, the deadline is relative to the internal clock of the task. A task τ_i misses its deadline iff while running an execution vertex ($e_{i,j} : \langle c_{i,j}, d_{i,j} \rangle$) its internal clock value I_i is greater than the deadline specified in the vertex: $I_i > d_{i,j}$. Notice that deadlines of the execution vertices have no impact on the real-time execution of the tasks. Thus, if a task misses one of its deadlines it goes on.

Killing parameter: The killing parameter K_i is the maximum value of the task internal clock allowed. As stated before, the task internal clock increases with time, it is decreased when running a wait vertex. If the internal clock of the task reaches this bound (K_i) then the task is stopped. For the analysis point of view, the killing parameter is used as a task clock bound.

C. Example

Let's consider the example in Figure 1 composed of one GGTM task composed of three execution vertices (e_1, e_2 and e_1), three wait vertices (w_4, w_5 and w_6) and eight transitions.

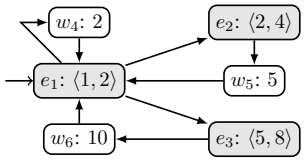

Fig. 1. GGTM example.

A trace is a sequence of vertices of the studied task noted $[v_{i,j}, \ldots, v_{i,k}]$. In order to explain the behavior of such a task model we will consider the following trace: $[e_1, w_4, e_1, e_2, w_5, e_1, w_4]$. In this trace, the task is executed without being delayed nor preempted by any other task. The timeline of the trace is presented in figure 2.

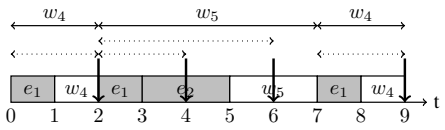

Fig. 2. Execution trace $[e_1, w_4, e_1, e_2, w_5, e_1, w_4]$.

III. GGTM Expressiveness and Motivations

We describe the expressiveness of the GGTM, and compare it to relevant execution models from the literature.

A. Periodic task model

Let's consider the classical periodic task model [1] with release date (r_i), priority (P_i), period (T_i), deadline (D_i), and WCET (C_i).In our model, the periodic task is easily translated with a GGTM composed of 2 wait vertices (the first represents the release, and the second the period) and as execution vertex for the task computation, and the deadline. Figure 3 represents the GGTM for a periodic task. Consequently, our model is strictly more expressive than the periodic task model.

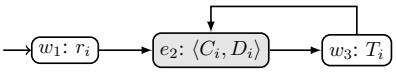

Fig. 3. Periodic task representation.

B. Multiframe and generalized multiframe task model

The multiframe task model and the Generalized Multiframe (GMF) task model [7] are two extensions of the periodic task model. The GMF task model subsumes the multiframe task model [5]. Both have a schedulability analysis for the fixed-priority preemptive scheduler. The translation of the GMF task model into the GGTM is straight forward. Indeed, a GMF task is a sequence of alternate an execution vertex followed by a wait vertex. However, a major difference exists. GMF deals with sporadic tasks, thus the period is the minimum separation between two execution. In our execution model, the wait vertex represents an exact "separation" between two vertices.

C. Graph based task model

Another kind of extension is to use a graph (or Direct Acyclic Graph - DAG) to describe the task behavior. The Recurring Real-Time task model [8] (RRT) extends the GMF task model by considering a DAG instead of a sequence of execution and "period". The non-cyclic RRT mode [13] also extends both recurring RT and non-cyclic GMF. Finally, the digraph real-time task model [16] subsumes the previous ones by considering a directed graph composed of vertices representing the executions (and deadlines) and edges for the "period" (delay between executions). Let's consider the digraph (Directed graph) task represented in figure 4.

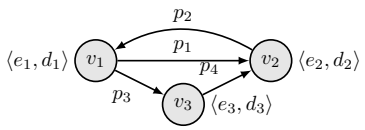

Fig. 4. Digraph task.

It can be translated into the GGTM by representing each digraph vertex with an execution vertex and each edge of the digraph with a wait vertex in the proposed model. Figure 5 shows the result of the translation.

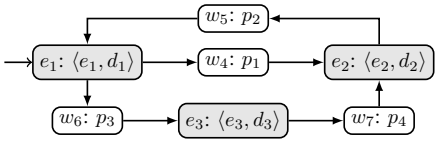

Fig. 5. Digraph task translation.

The GGTM is more general than the digraph real-time task model because we can chain execution vertices and also wait for vertices contrary to the digraph real-time model in which an execution is always followed by a "period". However, there are four main differences: first, the GGTM has a unique initial vertex, contrary to the digraph RTT in which the initial node is not specified. Then, the digraph real-time task model is sporadic; the edges are minimum inter-release separation time and not exact inter-release time. Moreover, the digraph real-time task model is based on the preemptive earliest deadline first scheduler. Finally, in the task model presented in this paper, we can have deadlines greater than the periods.

D. Concrete motivation

a) Re-synchronization: The first issue we had on different robots came from the sensors reading. In practice, the sensors (laser, GNSS, cameras, ...) produce data with a specific frequency. The rate is accurate but never perfect. Thus, sometimes the synchronization can be lost, and some data may be lost. To avoid this issue we need to specify a task made with two-part: one for the synchronization and one for data computation. Figure 6 shows how to answer this problem with the proposed task model. The cycle composed of e_1 and w_3 is used to check if a new data is produced by the sensor. If one data is produced then the vertex e_2 is executed to treat the input.

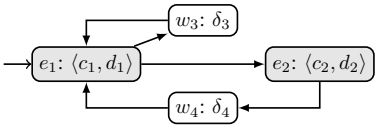

Fig. 6. Synchronization issue solved.

In practice, this pattern is characterized by a tiny wait duration in w_3 depending on the sensor specification and a longer wait duration after the data computation in w_4. This pattern also gives a precise bound of the delay between the data produced by the sensor and the beginning of its computation: δ_3.

b) Complex tasks: In practice, if a task has complex behavior from the time point of view, the GGTMmodel can be helpful. This execution model has proven to be useful for the implementation of various complex tasks such as path planning and tracking, SLAM, and various decision tasks.

IV. GGTM Analysis

A. Methodology

The main objective of the analysis of the GGTM is to compute the behavior graph of each task. Once this graph obtained we can check if the deadlines are missed and highlight counter-examples traces. This graph may also provide valuable information about the real-time behavior of the systems.

Theorem 1. *In GGTM, there is no possible interaction between tasks mapped to different processors.*

Proof. The tasks are mapped on a single processor and are not allowed to change their allocation during the execution. Moreover, there is no synchronization nor precedence between tasks. Consequently, tasks from different processors are independent, from the real-time point of view. □

Theorem 2. *In GGTM, task behavior can only be altered by the tasks running on the same CPU with a higher priority.*

Proof. The only interactions between tasks can be either delay or preemption. A task can only be delayed (resp. preempted) by another task with a higher priority. Consequently, the interactions between tasks can be completely ordered. □

Lemma 1. *Consequently, the analysis of system composed of several GGTM can be done incrementally: one processor at a time and one task at a time-ordered by their priorities.*

The main algorithm consists of: for each processor, for each task τ_i allocated on this processor ordered by priority, compute its behavior graph (noted $\mathcal{B}_i$). The behavior graph of a task is computed by taking into account its supply graph of the processor. The supply graph represents the availability of the processor for a specific task. Because we are dealing with graphs for the tasks, the supply "function" is also a graph. Once the behavior graph of a task computed we can check different properties such as looking for deadline miss. Finally, the current processor utilization induced by the behavior of the current task is added to the supply graph for the next task. Then the supply graph is compacted to reduce as possible its size. This methodology is described by the algorithm 1.

```
foreach cpu ∈ cpus do
    tasks ← {τ_i ∈ Γ | cpu_i = cpu};
    tasks ← sort_by_priority(tasks) ;
    supply ← empty_supply_graph();
    foreach τ_i ∈ tasks do
        B_i ← compute(supply, G_i) ;
        check_properties(B_i) ;
        supply ← next(B_i) ;
        supply ← compact(supply) ;
    end
end
```

Algorithm 1: Analysis algorithm principle

B. Supply graph definition

A supply graph is a directed graph noted $\mathcal{S}$ composed of vertices noted $\mathcal{V}^{\mathcal{S}}$ and arcs noted $\mathcal{A}^{\mathcal{S}}$. Each vertex $v_i^{\mathcal{S}}$ is a tuple made of the vertex type and the duration (noted Δ_i); the vertex type can be either $Idle$ or loaded $Loaded$. While analyzing the task τ_i the $Loaded$ vertices correspond to the executions of the tasks with a higher priority than τ_i. If the supply graph current vertex type is $Idle$ then the task τ_i can be executed. According to the analysis algorithm 1 a specific supply graph is only used for the computation of specific task behavior. Thus, to ease the paper reading while analyzing the task τ_i the corresponding supply graph will be noted $\mathcal{S}_i$.

C. Behavior graph definition

The main objective of the behavior graph of a task is to show its behavior according to its supply graph. Consequently, it contains both information related to the supply graph and information on the task. The behavior graph $\mathcal{B}$ is a directed graph composed of vertices noted $\mathcal{V}^{\mathcal{B}}$ and arcs noted $\mathcal{A}^{\mathcal{B}}$. Each vertex contains a tuple $\langle s, t, begin, end\rangle$ in which: s is the current supply vertex, t is the current task vertex, $begin$ is used to represent the internal behavior parameters while entering the current vertex, and end at the end of the vertex. Both $begin$ and end contains three parameters $\langle S, I, E\rangle$:

1) S is a positive number representing the remaining duration of the **S**upply vertex s. This duration can only decrease.
2) I is a positive number representing the task's **I**nner clock value. This value can only increase up to K_i.
3) E is a positive number representing the remaining **E**xecution time needed by the task to finish t. This value can only decrease.

A behavior vertex represents the evolution of the system behavior while time changes but the vertices of both the supply graph and task graph remain unchanged. The arcs of the behavior graph represent the change of either the supply graph vertex or the task graph vertex. The pointed notation will be used if the vertex is not needed (ex. $begin.I$).

D. Killing parameter

As previously described, each task τ_i has a killing parameter K_i which is an upper bound for the internal clock. Thus, during the computation of the behavior task, if the internal clock reaches K_i is marked as killed. Because our main objective is to study the schedulability of the task the killer parameter must be long enough to detect deadline miss, but it also has to be as small as possible to reduce computation time. Thus, it must be greater than all the deadlines ($K_i > max(d_{i,j})$) and greater than the wait duration to avoid stopping the task while waiting ($K_i > max(\delta_{i,j})$).

We can determine K_i value with the following formula:

$$K_i = max(d_{i,j}) + max(\delta_{i,k}) + 1$$

E. Supply graph initialization

The initial supply graph is used to represent a processor fully available for the tasks. It is composed of a single vertex:

$$\langle Idle, +\infty\rangle$$

F. Behavior graph computation

The behavior graph computation principle is to compute, for each vertex from the initial one, the next possible vertices. This is repeated until no new vertex can be added to the graph. We will explain later why this process terminates.

1) Initial vertex: The first vertex is composed of the supply and the task vertex are the corresponding initial vertices: $s_1 = v_1^{\mathcal{S}}$ and $t_1 = v_1$. The remaining time into the initial supply vertex is equal to its duration. The initial internal clock value is set to zero. The initial remaining computation time of the initial execution vertex is set to its WCET if the first task vertex is equal to an execution vertex, or it equals to zero if it's a wait vertex.

$$\langle v_1^{\mathcal{S}}, v_1, begin, end \rangle$$

The initial vertex begin parameters depend on the first task vertex type. If the task vertex type is $Exec$ then E the parameter representing the remaining computation time is equal to c_1 otherwise it is equal to 0. In both cases, the supply duration is set according to the first supply vertex duration and the initial task internal clock value is set to 0.

$$\begin{array}{lcl} (type(v_1) = Exec) & \implies & (begin = \langle \Delta_1, 0, c_1 \rangle) \\ (type(v_1) = Wait) & \implies & (begin = \langle \Delta_1, 0, 0 \rangle) \end{array}$$

2) Vertex length: The $length$ of a behavior vertex $v_i^{\mathcal{B}}$ is the time spent is this vertex before a transition occurs. Thus, the length of a behavior vertex depends on the supply and the task vertex types.

Supply is loaded: The supply graph represents the tasks with higher priority. Thus, if the supply is loaded then the system have to wait for the end of the current supply vertex:

$$(type(s) = Loaded) \implies (length = begin.S)$$

Task is running: If the supply is idle and the current task vertex is an execution then the system is running the studied task. This ends as soon as the execution is finished or the current supply vertex ends:

$$\begin{array}{c} (type(s) = Idle \wedge type(t) = Exec) \implies \\ (length = min\left(begin.S, begin.E\right)) \end{array}$$

Task is waiting: If the supply is Idle and the task is waiting then the current vertex ends as soon as the supply vertex ends or the clock reached the wait vertex clock duration.

$$\begin{array}{c} (type(s) = Idle \wedge type(t) = Wait) \implies \\ (length = min\left(begin.S, max\left(begin.I - \delta, 0\right)\right)) \end{array}$$

The max part of the formula is necessary to maintain a non-negative duration if the wait vertex is late ($I > \delta$). In this configuration, the behavior vertex ends without consuming time.

3) Time evolution: A time transition is represented in each behavior graph vertex using the $begin$ and end parameters which defined the internal changes of the systems when the time evolves. The $begin$ parameter is computed while entering the vertex, thus the end parameter depends on the inner vertex parameters: the supply and the task current vertex and the 'begin' parameter.

Task is waiting: If the task is waiting then the supply duration decreases according to time, the clock increases and the remaining computation is 0:

$$\begin{array}{c} (type(t) = Wait) \implies \\ \left(end = \left\langle \begin{array}{l} begin.S - length, \\ begin.I + length, \\ 0 \end{array} \right\rangle \right) \end{array}$$

Task is running: If the supply is idle and the task is running an execution vertex then the supply duration decreases according to time, the clock increases and the remaining computation of the task vertex decreases:

$$\begin{array}{c} (type(s) = Idle \wedge type(t) = Wait) \implies \\ \left(end = \left\langle \begin{array}{l} begin.S - length, \\ begin.I + length, \\ begin.E - length \end{array} \right\rangle \right) \end{array}$$

Task is preempted: If the supply is loaded and the task is an execution vertex (the tasks is preempted by higher priority tasks) then the supply duration decreases according to time, the clock increases and the remaining computation stay unchanged:

$$\begin{array}{c} (type(s) = Idle \wedge type(t) = Wait) \implies \\ \left(end = \left\langle \begin{array}{l} begin.S - length, \\ begin.I + length, \\ begin.E \end{array} \right\rangle \right) \end{array}$$

4) "Choice" transition:

Supply transitions: At the end of the vertex time evolution, the next supply or task graph transition has to be selected. The supply graph represents the higher priority tasks, thus if its current vertex is finished ($end.S = 0$) then its next transition is selected even if the task graph vertex is also finished. It is important to notice that the process is repeated for each next transitions. To ease the following formula, the next vertex according to the task graph will be noted v'. This notation is also applied to every behavior graph element including to supply vertices, $begin$ and end parameters.

$$\langle s, t, begin, end \rangle \rightsquigarrow \langle s', t, begin', end' \rangle$$

$$(end.S = 0) \implies (begin' = \langle \Delta_{s'}, end.I, end.E \rangle)$$

Task transition: The next task vertex is selected only if the current task vertex is finished and the supply vertex is not finished.

$$\langle s, t, begin, end \rangle \rightsquigarrow \langle s, t', begin', end' \rangle$$

If the next task vertex type is wait then the clock is reduced by the clock duration specified in the task graph vertex:

$$\begin{array}{c} (end.S = 0 \wedge type(t') = Wait) \implies \\ (begin' = \langle S, end.I - \delta, 0 \rangle) \end{array}$$

If the next task vertex is an execution then the remaining computation time is set to the WCET of the corresponding task vertex:

$$\begin{array}{c} (end.S = 0 \wedge type(t') = Exec) \implies \\ (begin' = \langle S, end.I, c' \rangle) \end{array}$$

G. Next supply graph

Once the behavior graph computed for a task, the next supply graph, for the next task, has to be computed. The next supply graph is a graph with the same structure as the current behavior graph (same vertices number and same transitions). Each behavior vertex is transformed into a supply vertex depending on the fact the vertex uses the processor or not:

$$\langle s, t, begin, end\rangle \rightsquigarrow \begin{cases} Loaded(length) & if\ type(s) = (Loaded \vee Exec) \\ Idle(length) & otherwise \end{cases}$$

H. Compact supply graph

The computation of the behavior graph highly depends on the supply graph size that's why it is important to keep it as small as possible. A next supply graph created from the behavior graph can be compacted by two operations that maintain the supply graph's semantic.

Remove empty: consists in removing all the empty vertices. A supply vertex s_i is empty if its duration is equal to zero ($\Delta_i = 0$). To achieve this simplification every out coming arc of the empty vertex is added to every previous vertex. Finally, the empty supply vertex is removed.

Merge: consists in merging two consecutive "single" vertices with the same type. Indeed, if two vertices s_1 and s_2 have the same type ($type(s_1) = type(s_2)$) and the first one has only one out coming transition to the second ($next(s_1) = \{s_2\}$) which has only one incoming transition ($prev(s_2) = \{s_1\}$) (from the first one), then these two vertices can be merged into a single one. This functionality is described by algorithm 2.

foreach $x \in \mathcal{V}^{\mathcal{S}}$, $y \in \mathcal{V}^{\mathcal{S}}$ **do**
  if $mergeable(x, y)$ **then**
    $v \leftarrow$ new_vertex;
    $type(v) \leftarrow type(x)$;
    $\Delta_v \leftarrow \Delta_x + \Delta_y$;
    $\mathcal{V}^{\mathcal{S}} \leftarrow \mathcal{V}^{\mathcal{S}} + v$;
    foreach $p \in prev(x)$ **do**
      $\mathcal{A}^{\mathcal{S}} \leftarrow \mathcal{A}^{\mathcal{S}} + \langle p, v\rangle$
    end
    foreach $n \in next(y)$ **do**
      $\mathcal{A}^{\mathcal{S}} \leftarrow \mathcal{A}^{\mathcal{S}} + \langle v, n\rangle$
    end
    $\mathcal{V}^{\mathcal{S}} \leftarrow \mathcal{V}^{\mathcal{S}} - x - y$
  end
end

Algorithm 2: Merge two supply vertices.

I. Check task properties

Once the behavior graph computed we can check properties.

Deadline miss: A behavior graph vertex is deadline miss if the task vertex is execution and if the internal clock is greater than (strictly) its corresponding deadline:

$$deadline\_miss := (typet(t) = Exec \wedge end.I > d)$$

Killed: A behavior graph vertex is killed if its internal clock is greater than the killing parameter:

$$killed := end.I > K.$$

Late: A behavior graph vertex is late if its internal clock greater than its corresponding wait duration:

$$late := end.I > \delta.$$

Response time: The worst-case response time (WCRT) of a task execution vertex e is the maximum internal clock value for a behavior vertex with the same task vertex:

$$wcrt(e) := max\left\{end.I \mid t = e\right\}.$$

Schedulable: A task is schedulable if the behavior graph contains neither killed vertex nor deadline miss vertex.

Other properties: Any other properties related to the behavior graph can also be tracked. For example, we already used the behavior graph to study the impact of some specific task preemption on the worst-case response time of another task. The behavior graph can also be used for explanation, for example, if the task is not schedulable it is interesting to determine which paths can lead to a deadline miss.

J. Discussion

a) Pessimism: The main principle of the analysis method presented in this article is to compute every possible behavior of GGTM tasks. By construction, the behavior graph exhibits the exact behavior of the corresponding tasks. Consequently, the analysis method is exact (non-pessimistic).

b) Completeness: Moreover, the analysis can be applied to any GGTM systems without any restriction. Consequently, the method is complete.

c) Decidability: To prove the decidability of the method we have demonstrated that the decision procedure ends. The main algorithm iterates over the tasks. The number of tasks is bounded. For each task three algorithms are applies: computing behavior graph, checking real-time properties, computing the next supply graph and compacting the supply graph.

Checking the real-time properties is a simple request on the behavior graph, at worst each node is checked once. Thus, the termination is guaranteed.

Computing the next supply graph principle is to create a supply graph with the same structure as the input behavior graph. Thus, the next supply computation simply iterates on the behavior graph. Thus, the termination is also guaranteed.

Compacting the supply graph applies recursively two operations (remove empty and merge). Each operation reduces the size (number of vertices); thus the procedure ends.

To demonstrate that the computation of the behavior graph ends, we have to demonstrate that the number of vertices is bounded. Each node is a tuple $\langle s, t, begin, end\rangle$ in which begin and end are tuples $\langle S, I, E\rangle$. The supply graph and the task graph are bounded by definition. The parameter S is a positive number bounded by the maximum duration of the supply nodes. The I parameter is a positive number bounded by the killing parameter. The parameter E is a positive number bounded maximum duration of the executions vertices of the task graph. All the parameters are bounded then the number of nodes is also bounded. Consequently, the size of the behavior graph is bounded, and the procedure ends.

V. Experimental results

A. Example

To illustrate the analysis method proposed in this article let's consider an example of three tasks. The three tasks are executed on the same processor and ordered by priority.

1) Tasks set: The first task is a simple periodic task with a first release date equals to zero, $C = 1$ and $D = P = 10$. The second task has the same pattern as the one presented in figure 6 with the following parameters values: e_1: $c_1 = 1$ and $d_1 = 2$, e_2: $c_2 = 5$ and $d_2 = 7$, w_3: $\delta_3 = 5$, w_4: $\delta_4 = 10$. The third task is a periodic task with release date equals to zero, $c_1 = 4$, $d_1 = 10$ and $\delta_2 = 10$

2) Analysis: First task: The analysis is made task by task: for each task, the corresponding supply graph is given, and the result is the behavior graph and/or traces that highlight a specific property.

a) Supply graph: The supply graph of the first task is the empty supply graph in which the processor is never used.

b) Behavior graph: As a result of the analysis of the first task, its behavior graph is computed. Because the supply graph is empty the behavior graph is very similar to the GGTM. Figure 7 shows the behavior graph.

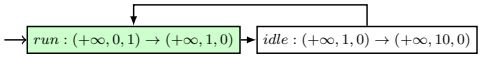

Fig. 7. Behavior graph of the first task

3) Analysis: Second task:

a) Supply graph: The supply graph of the second task is computed based on the behavior graph of the first task. Thus, it only contains two vertices: at first, the processor is loaded for 1 time unit and then the processor is idle for 9 time units. These two vertices are repeated indefinitely.

b) Behavior graph: The behavior graph of the second task is made of 15 vertices as presented in figure 8:

- the vertices in blue and noted P_{xxx} represents the preemption of the second task by higher priorities tasks;
- the vertices in green and noted R_{xxx} represents the moments in which the second task is running;
- the vertices in white and noted I_{xxx} represents the vertices in which the processor is idle:
- the vertices in gray and noted L_{xxx} represents the moments in which the higher priority tasks are executed while the second task is waiting.

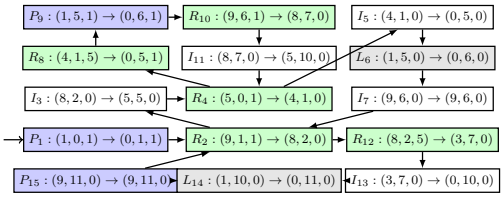

Fig. 8. Behavior graph of the second task.

c) Properties: The behavior graph of the second task shows that there is no deadline miss, thus the second task is schedulable. It also shows the preemption and other information about task real-time behavior.

4) Analysis: Third task:

a) Supply graph: The supply graph of the third task is based on the behavior graph of the second task but compacted. Figure 9 shows the supply graph.

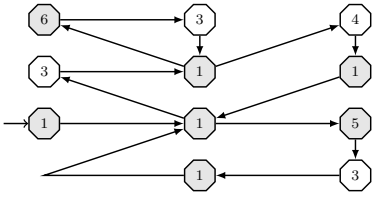

Fig. 9. Supply graph of the third task

For example, the vertices R_8, P_9 and R_{10} are merged because they represent three consecutive processor utilization with a total duration of 6. The vertices I_7 and P_{15} are removed because they are not consuming time.

b) Properties: Due to the number of vertices, the behavior graph won't be displayed here. But our analysis tool also provides the minimal (in length) traces for each property. In this case, the task is not schedulable and the minimal trace that leads to a deadline miss composed of five vertices:

1) $\langle L_0, e_1, (1, 0, 4) \to (0, 1, 4)\rangle$: at first the processor is used by the higher priority tasks for 1 time unit;
2) $\langle L_1, e_1, (1, 1, 4) \to (0, 2, 4)\rangle$: then the processor is loaded for 1 other time unit;
3) $\langle L_3, e_1, (5, 2, 4) \to (0, 7, 4)\rangle$: the processor is loaded for 5 time units; at this point the task is unschedulable.
4) $\langle I_4, e_1, (3, 7, 4) \to (0, 10, 1)\rangle$: the task is running the vertex e_1 for 3 time units
5) $\langle L_5, e_1, (1, 10, 1) \to (0, 11, 1)\rangle$: finally the task is preempted for 1 time unit and the task misses its deadline for the vertex e_1.

B. Computation Time Benchmark

In the following, we discuss the computation time of the analysis we proposed.

a) Tasks generation: We have previously demonstrated the expressiveness of our model and its schedulability analysis in the previous sections thus, we decided to generate periodic tasks because generating a GGTM for many tasks is a very complex problem. Indeed, most of the generated complex tasks system may be trivially not schedulable and then the computation won't increase with the number of the tasks. Moreover, the use of periodic tasks provides a clearer insight into the complexity of the system. Thus, We decided to generate periodic tasks and increase the number of tasks to increase the computation of our method. The task period is randomly chosen, its computation time is randomly picked between 0 to T_i/n with n the number of tasks of the system.

Thus, the task load is homogeneously dispatched between the different tasks and the load of the system stays relatively small. Each task has a deadline equals to its period. For each task number n a hundred simulation is run.

b) Results: The benchmark is executed on an *i7-8850H@ 2.60GHz*. Figure 10 shows that the computation increase according to the number of the tasks but even for many tasks the computation time is just a few seconds. Let's remind that the computation time is for one processor and increases linearly with the number of processors.

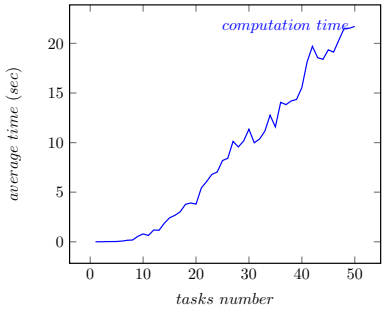

Fig. 10. Computation time benchmark.

As mentioned previously, we used this method on different systems and the computation time was never an issue; first, because of the number of tasks by processor is quite small. Moreover, task time parameters are correlated.

c) Discussion: The previous benchmark highlight the efficiency of the analysis method. However, it is also interesting to notice the difference between the complexity and the computation time. Indeed, the complexity relies on the worst-case scenario in which : during the computation of the behavior graph each time unit leads to a new vertex, each behavior graph and supply graph are completely connected, the size of the supply graph is the size of the behavior graph, and the compact algorithm does not affect. In practice, the compact algorithm and the links between timings restrain the size of the behavior and supply graph.

VI. Conclusions

In this paper, we introduce the Generalized Graph Real-Time Task Model. Then, we compare its expressiveness with the state of the art and demonstrated that the proposed model is more general than most of the previous ones, and does not subsume by the other ones. We also present an exact analysis method, and shows that it can scale to large real-time systems.

Different extensions are under investigation. First, we are exploring the possibility to check different temporal logic formulas (LTL/CTL) on the behavior graph obtained by the analysis. The main objective is to checks complex properties about the real-time behavior of the tasks.

Another perspective to be considered is to translate part of the algorithm semantic on the task graph. The intent here is to add branching conditions described by first-order logic formulas. Indeed, in our model, all the next arcs are allowed but in practice, the transitions are ordered and guarded. This enhancement may remove unreachable traces.

References

[1] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, 1973.

[2] A. K. Mok, "Fundamental design problems of distributed systems for the hard-real-time environment," Cambridge, MA, USA, Tech. Rep., 1983.

[3] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *Proceedings of the Real-Time Systems Symposium*. IEEE Computer Society, 1990.

[4] A. K. Mok and D. Chen, "A multiframe model for real-time tasks," in *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, 1996.

[5] A. K. Mok and D. Chen, "A multiframe model for real-time tasks," *IEEE Transactions on Software Engineering*, 1997.

[6] S. K. Baruah, D. Chen, and A. K. Mok, "Static-priority scheduling of multiframe tasks," in *11th Euromicro Conference on Real-Time Systems (ECRTS)*, 1999.

[7] S. Baruah, D. Chen, S. Gorinsky, and A. Mok, "Generalized multiframe tasks," *Real-Time Systems*, vol. 17, no. 1, pp. 5–22, Jul 1999.

[8] S. K. Baruah, "A general model for recurring real-time tasks," in *Proceedings of the 19th IEEE Real-Time Systems Symposium*, 1998.

[9] ——, "Dynamic and static-priority scheduling of recurring real-time tasks," *Real-Time Systems*, 2003.

[10] S. K. Baruah and S. Chakraborty, "Schedulability analysis of non-preemptive recurring real-time tasks," in *20th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2006.

[11] N. Tchidjo Moyo, E. Nicollet, F. Lafaye, and C. Moy, "On schedulability analysis of non-cyclic generalized multiframe tasks," in *2010 22nd Euromicro Conference on Real-Time Systems*, 2010.

[12] S. Baruah, "The non-cyclic recurring real-time task model," in *2010 31st IEEE Real-Time Systems Symposium*, 2010.

[13] S. K. Baruah, "Preemptive uniprocessor scheduling of non-cyclic GMF task systems," in *16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA*, 2010.

[14] N. Guan, C. Gu, M. Stigge, Q. Deng, and W. Yi, "Approximate response time analysis of real-time task graphs," in *Proceedings of the IEEE 35th IEEE Real-Time Systems Symposium*, 2014.

[15] M. Stigge and W. Yi, "Graph-based models for real-time workload: a survey," *Real-Time Systems*, vol. 51, no. 5, pp. 602–636, 2015.

[16] M. Stigge, P. Ekberg, N. Guan, and W. Yi, "The digraph real-time task model," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2011, pp. 71–80.

[17] M. Stigge, P. Ekberg, N. Guan, and W. Yi, "On the tractability of digraph-based task models," in *23rd Euromicro Conference on Real-Time Systems, ECRTS*, K. Årzén, Ed. IEEE Computer Society, 2011.

[18] N. Guan, P. Ekberg, M. Stigge, and W. Yi, "Resource sharing protocols for real-time task graph systems," in *23rd Euromicro Conference on Real-Time Systems, ECRTS*, K. Årzén, Ed., 2011.

[19] Rajeev Alur and David L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, p. 183–235, 1994.

[20] C. Ramchandani, "Analysis of asynchronous concurrent systems by timed petri nets," USA, Tech. Rep., 1974.

[21] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" in *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*, F. T. Leighton and A. Borodin, Eds., 1995.

[22] F. Cassez and K. Larsen, "The impressive power of stopwatches," in *CONCUR 2000 — Concurrency Theory*, C. Palamidessi, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 138–152.

[23] Y. Abdeddaïm and O. Maler, "Preemptive job-shop scheduling using stopwatch automata," in *Tools and Algorithms for the Construction and Analysis of Systems*, J.-P. Katoen and P. Stevens, Eds. Springer Berlin Heidelberg, 2002.

[24] E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "Schedulability analysis of fixed-priority systems using timed automata," *Theoretical Computer Science*, vol. 354, 2006.

[25] E. Fersman, P. Krcal, P. Pettersson, and W. Yi, "Task automata: Schedulability, decidability and undecidability," *Information and Computation*, vol. 205, no. 8, pp. 1149 – 1172, 2007.