Periodic state-machine aware real-time analysis

Nicolas Gobillot David Doose Charles Lesire Luca Santinelli
ONERA – the French Aerospace Lab
2 avenue Edouard Belin, FR-31055 TOULOUSE CEDEX 4
email: firstname.lastname@onera.fr

***Abstract*—Nowadays dangerous, repetitive or precision requiring jobs are done by robots like flying drones, industrial assembly arms or medical assistants. In all these cases, human beings can interact with the machines, therefore it is essential to guarantee that every part of the robot's software and hardware will produce a safe behavior: both the overall behavior and the local behaviors of such embedded systems have to be carefully analyzed. The complexity of embedded systems software architectures increase with more and more tasks involved; state-machines are applied to implement more functional capabilities of the tasks; and the task models used in the analyses gain in complexity. The analysis techniques have to be adapted in order to face such new complexities. This paper focuses on the real-time analysis of state-machine-based software architectures. We propose a method to analyze the temporal behavior of a component-based architecture in which the components are described by state-machines. The method computes an accurate worst-case response time by taking into account the state-machines of the components. Finally, we validate our approach with a real real-time robotic case study.**

I. Introduction

Computer-based systems occupy an increasing place in our everyday or professional life. The increasing capabilities of the robots imply that more and more tasks are performed in the presence of humans and even in collaboration with them. Therefore we need to ensure that these robots will not damage themselves or their environment and more importantly they do not hurt any human being in their whole life. Safety concern has been considered regarding several aspects of robotic embedded systems such as collision avoidance [1], safe interaction with human beings [2], fault detection and tolerance [3] or controller synthesis [4].

In all the existing physical and software safety procedures, the notion of time is a matter of importance since the temporal behavior have to be guaranteed. These guarantees are brought by timing analyses on the real-time software. During the past decades, the widely known *Liu and Layland* [5] fixed priority models, as well as dynamic priority task models such as *Earliest Deadline First* (EDF) [6] have been precise enough for software analysis.

Due to the increasing complexity of the real-time systems, the analysis methods had to be adapted taking into account resource partitioning with methods such as Priority Inheritance Protocol (PIP) [7], Priority Ceiling Protocol (PCP) [8] or Stack Resource Policy (SRP) [9]. Other improvements needed to account for task interdependency [10], but also for more refined elements such as the operating system [11].

In modern systems, the computing needs growth implies structural modifications in the task concept; moreover, the new task models require new analysis methods such as the *Digraph real-time task model* [12], which has been designed to extend the expressiveness of the task models by taking into account the state-machines of the components. With a practical perspective, [13] uses the *digraph real-time task model* to model and analyze Simulink and SCADE programs.

In embedded systems middlewares separate software development from operating system, abstracting operating system specific parameters and functions from user-defined programs. The most common robotic middlewares are ROS [14] and Orocos [15]. In the context of this work, we have focused on the study of the Orocos middleware which is widely used in the robotic community thanks to its inherent real-time capabilities.

We propose a modeling method of the depicted systems by precise temporal behavior analysis. The method faces states machines describing task execution behaviors, and it computes accurate Worst-Case Response Times (wcrts).

The paper is organized as follows: Section II presents the state-machine-based architecture concept with the definition of the tasks and their state-machines. Section III presents our proposed method, decomposed in two steps: the creation of execution traces and the computation of the wcrt using these traces. Section IV shows the efficiency of the analysis and an intuition of the gain over the classical analysis with relation to wcrt. Section V presents a robotic case-study on which we have used this method.

II. State-machine-based software model

This section of the paper presents our software model. Many complex software systems are made of a component-based architectures, and the set of components is mapped on tasks modeled as state-machines. To increase accuracy, we have also modeled internal parts of each state of the state-machines.

A. Component and task description

A component is a software device carrying an elementary function. The component model is made of two main parts: the communication interface, and the behavior which is modeled with a state-machine. The components communicate with each other by non-blocking data transfer or function call. The non blocking aspect of the communication implies a temporal independence between the tasks. The state-machine based behavior is preferred over a simple task one (without state-machine), due to its increased overall robustness to failures and the possibility of a more fine grained representation of the task behavior.

978-1-4673-7929-8/15/$31.00

B. Task definition

The components are mapped by the middleware onto tasks of the host operating system. The behavior of the task is then defined by the component's state machine which is run by the task. In the rest of the paper we make use of both component and task to denote the same, since a task is always an instantiation of a component. The activity of a task indicates its real-time properties such as its period, its deadline and its priority. The real-time behavior is guaranteed by using a real-time operating system (i.e. a Xenomai patched Linux) with a fixed priority scheduler. The period configures the task execution frequency; the priority is used by the fixed priority scheduler to schedule the tasks; the deadline corresponds to the time at which the task must have finished its execution. Finally, the task has an execution time depending on the hardware platform and the function it has to exploit. A task τ_i is referred to a tuple $(R_i, T_i, P_i, D_i, state - machine_i)$ such that

$$\begin{cases} R_i & : release\ time \\ T_i & : period \\ P_i & : priority \\ D_i & : deadline. \end{cases} \quad (1)$$

$state - machine_i$ belongs to the task model since it defines its behavior. The worst-case execution times of the tasks $wcet_i$ are computed from the state-machines of the tasks (see Section III-A), hence they are not part of the task model.

In the rest of the paper the following hypotheses are assumed.

Hypothesis 1: The deadlines are lesser or equal than the period: $\forall i, D_i \leq T_i$

Hypothesis 2: The release time of the tasks is unknown between their wake-up and their period: $\forall i, R_i \in [0..T_i]$

The wake up of the task is done by the operating system. Since the execution models of the tasks do not provide the ability to set or to control their activation times, it is impossible to know the first release times R_i of the tasks. We then assume that the release time of each task can take any value between 0 and its period.

C. Task model

In this paper we will focus on the temporal behavior analysis of tasks, which depends on both the real-time properties of the tasks activities, and on the internal behavior of the tasks (functional and timing behavior defined by the state-machine).

A task state-machine is a set S of n states and a set E of m edges (or transitions), such that:

$$S = \{s_1, \ldots, s_n\} \quad \text{and} \quad |S| = n \quad (2)$$
$$E = \{e_1, \ldots, e_m\} \quad \text{and} \quad |E| = m \quad (3)$$

These state-machines carry all the functions and algorithms of their host task. The functionality of a task is time consuming and our model uses the state-machine structure to precisely compute the worst-case execution times of the tasks.

1) State-machine: Each state s_i contains up to four parts: $entry_i$, run_i, $handle_i$ and $exit_i$.The $entry_i$ part contains the code to be run whenever the state-machine arrives to state s_i; the $exit_i$ step is run as soon as the state-machine leaves s_i; the run_i method contains the core of the state, that is executed at each iteration when the state-machine is in state s_i; lastly, the $handle_i$ part executes each period the state-machine stays in the same state, just after executing run_i. Two possible execution cycle per iteration for a task: 1) the state-machine stays in the same state at the next iteration: it executes run_i and $handle_i$; 2) the state-machine goes from state s_i to state s_j: it executes run_i, $exit_i$ and $entry_j$. Regarding the task's periodic execution, the state-machine fires a transition per task period. Moreover the state-machine can fire at most one transition per iteration. Not all the 4 parts are need for a state to be defined, but at least run_i has to, depending on the modeled application.

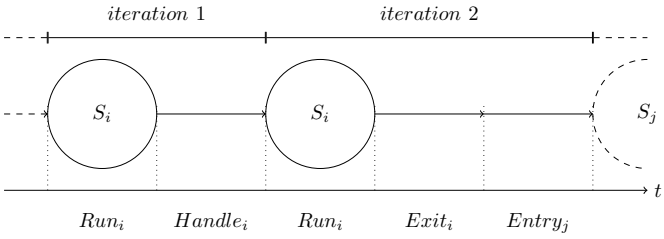

Fig. 1. State-machine transition protocol

Fig. 1 shows two consecutive iterations of the execution of a task. In this task all the $entry_i$, run_i, $handle_i$ and $exit_i$ parts are defined. In the first iteration, the task stays in state s_i, and the run_i and $handle_i$ methods are executed. In the second iteration, the task goes from state s_i to state s_j, and the run_i, $exit_i$ and $entry_j$ methods are executed.

2) Execution times: In order to obtain the execution times of the different elements of the state-machines ($entry$, run, $handle$ and $exit$ for each state), we have used two techniques: static analysis [16] on the compiled source code, and probabilistic analysis [17] on execution logs. In both cases, we extracted the worst case execution times of the state-machines in order to compute the wcrt and analyze the deadline respect of the tasks.

III. Analysis

To ensure that a task system is schedulable, the response times of the system's tasks have to be computed. In our

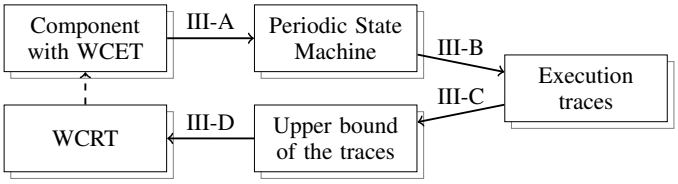

Fig. 2. Process of the task model analysis

modeling and analysis framework we make use the component model with a state-machine, while the wcet of the state-machine's parts are computed to create a *Periodic State-Machine* (PSM), Section III-A. A PSM is basically a state

machine that carries its actions on its transitions. A method to extract a request bound function rbf per instance of the PSM is proposed in Section III-B. A state-machine can be executed in many different sequences and we produce a request bound function per possible sequence of the PSM. Our method uses all the request bound functions of a component to compute an upper bound that not necessarily represent a possible execution of the PSM but it is safely defined, Section III-C. This upper request bound function is used to compute the wcrt of the components, Section III-D. For that, we have modified a fixed point formula to use the upper request bound functions in place of the task's execution time. Lastly, the wcrt is injected in the component model to compare it with its deadline and to check the component schedulability. This process is shown in Fig. 2.

A. Periodic state-machine

The tasks are deployed following the specification of the software, which implies that the tasks have a periodic activity. In order to analyze the state-machine's behavior, we have to model it as a PSM. This way, we can abstract away the implementation specificities such as the *entry*, *run*, *handle* and *exit* parts of the state-machine. We also abstract into the PSM's transitions the delays induced by the middleware's interactions. The periodic aspect of the state-machine states that at most one transition has to be fired per task period.

The PSM model specifies that all the time spent in the periodic state-machine is taken by the transitions for two reasons: a) The four methods (entry, run, handle, exit) of the state-machine states contain code that takes time to execute. These four methods are carried by the transitions between states and no executable code is present in the states. b) The operating system on which are mapped the tasks has not been taken into account in the model. We assume that it does not have any effect on the temporal behavior.

We define a PSM as a set of states S, which are the same states than the original state-machine (see Equation (2)), a set of transitions Σ, along with a timing function δ, defined by:

$$\Sigma = E \cup \{s \to s \,|\, s \in S\} \tag{4}$$

$$\forall \sigma \in \Sigma, s_i \xrightarrow{\sigma} s_j,\ \delta(\sigma) = \begin{cases} s_i \neq s_j: & wcet(run_i) + wcet(exit_i) + wcet(entry_j) \\ s_i = s_j: & wcet(run_i) + wcet(handle_i) \end{cases} \tag{5}$$

The set of transitions Σ contains all the original transitions of the state-machine (E) plus all the loops over states of S. The timing function δ associates to each PSM transition the Worst Case Execution Time (wcet) of the methods involved in this transition (see Fig. 1 for a description of the transition protocol). According to the definition of PSM, the following property holds:

Property 1: The periodic state-machine fires a transition at every execution period.

This transition may be a loop over the same state, or a transition from a state to another. The second assumption is on the strongly connection of the state machine:

Property 2: The periodic state-machine is strongly connected: every state can be reached from any state through a sequence of transitions.

$$\forall s_i, s_j \in S, \exists \sigma_1 \dots \sigma_n \in \Sigma \mid s_i \xrightarrow{\sigma_1} \dots \xrightarrow{\sigma_n} s_j \tag{6}$$

This assumption is useful, because it avoids considering that some dead states exist in the task behavior, and it is a needed assumption for the reasoning on the wcrt computation (see Section III-D). Moreover, this assumption is in a sense a good practice that we should enforce when developing the tasks: it allows to be able to put the task again in its initial state at execution in the case where a faulty behavior occurs. In practice, it is possible to make a state-machine strongly connected by adding transitions from each state to the initial state of the state-machine.

B. Execution traces

In this section, we explain how an execution trace is constructed from the states and the transitions of a state-machine. In order to be used in the analysis, the traces have to be ordered so we use the *request bound function* of the traces to get a partial order between traces.

1) Traces: To access the input and output states of a transition, we define the following notations:

$$\left.\begin{matrix} from\,(\sigma) \in S \\ to\,(\sigma) \in S \end{matrix}\right| from\,(\sigma) \xrightarrow{\sigma} to\,(\sigma) \tag{7}$$

Reasoning on the executions of a state-machine so we define a trace $\mathcal{T}$ as an ordered sequence of transitions as indicated in the following expression: $\mathcal{T} = \langle \sigma_1, \dots, \sigma_N \rangle$

In order to use traces on our computations, we have to know the amount of transitions contained in a trace. The length of a trace is defined as the amount of transitions contained in the trace: $|\mathcal{T}| = |\langle \sigma_1, \dots, \sigma_N \rangle| = N$

Since this sequence is ordered, we can define the operator $\mathcal{T}[k]$ to access to the k^{th} transition of the trace: $\mathcal{T}[k] = \langle \sigma_1, \dots, \sigma_k, \dots, \sigma_N \rangle [k] = \sigma_k$

A trace is then constructed as a concatenation of transitions. To add the transition σ_M to the trace $\mathcal{T}$, we define the operator $\langle \mathcal{T}, \sigma_M \rangle$: $\langle \mathcal{T}, \sigma_M \rangle = \langle \sigma_1, \dots \sigma_N, \sigma_M \rangle$

In order to simplify the following equations, other concatenation operators are defined such as $\langle \sigma_i, \mathcal{T} \rangle$ to add the transition σ_i at the beginning of the trace $\mathcal{T}$ or $\langle \sigma_i, \mathcal{T}, \sigma_M \rangle$ to add transitions at the beginning and at the end of the trace:

$$\begin{aligned} \langle \sigma_i, \mathcal{T} \rangle &= \langle \sigma_i, \sigma_1, \dots \sigma_N \rangle \\ \langle \sigma_i, \mathcal{T}, \sigma_M \rangle &= \langle \sigma_i, \sigma_1, \dots \sigma_N, \sigma_M \rangle \end{aligned} \tag{8}$$

2) Partial pre-order relation on traces: Since we want to use the traces to compute the worst case execution time of the tasks traces, we have to define the conditions for which a trace is worst than another one. For that we introduce a partial order relation between traces with the operator $\leq$.

Looking for a worst trace regarding the execution times, we use the *request bound function* (rbf) of the traces [18]. The request bound function is the maximum processor request by task over a time interval. The tasks are executed periodically then the request bound function is a piecewise-constant and

increasing function. The step height relies on the execution times of transitions and thus the wcet of the task methods. The request bound function of a trace $\mathcal{T}$ at time t is noted $rbf(\mathcal{T}, t)$:

$$rbf(\mathcal{T}, t) = \sum_{i=1}^{|\mathcal{T}|} \left\lfloor \frac{t}{T} \right\rfloor \delta(\mathcal{T}[i]). \tag{9}$$

We then define the relation $\leq$ on traces by comparing the rbf of the traces:

$$\mathcal{T}(t) \leq \mathcal{T}'(t) \text{ iff} \forall t, rbf(\mathcal{T}, t) \leq rbf(\mathcal{T}', t) \tag{10}$$

The $\leq$ relation is clearly reflexive ($\forall \mathcal{T}, \forall t, \mathcal{T}(t) \leq \mathcal{T}(t)$) and transitive ($\forall (\mathcal{T}, \mathcal{T}', \mathcal{T}''), \forall t, \mathcal{T}(t) \leq \mathcal{T}'(t) \wedge \mathcal{T}'(t) \leq \mathcal{T}''(t) \Rightarrow \mathcal{T}(t) \leq \mathcal{T}''(t)$). It then defines a partial pre-order relation on the set of traces.

3) Feasible traces: A trace is feasible if every pair of consecutive transitions of the trace is feasible. Feasibility implies that the arrival state of the first transition is the starting state of the second one. The following equation defines a feasible trace:

$$\begin{aligned} \phi(\langle \sigma_1, \sigma_2 \rangle) &\equiv to(\sigma_1) = from(\sigma_2) \\ \phi(\langle \mathcal{T}, \sigma \rangle) &\equiv \phi(\mathcal{T}) \wedge to(\mathcal{T}[|\mathcal{T}|]) = from(\sigma) \end{aligned} \tag{11}$$

C. Upper bound of the traces

In order to compute an upper bound of a trace used to obtain the worst-case response time of a task, we have to compute every feasible traces of this task. We produce a trace maximizing the feasible traces regarding the request bound function.

1) Construction of a trace: An execution trace is built incrementally: it starts from any state and every feasible transition is fired. This process is repeated for all the possible transitions and executed recursively until the trace reaches a specific length defined in Equation (17) (see Section III-D1).

In order to define the following equations, we have to define the $next$ operator representing the set of traces that follows a given trace. The $next$ operator is defined as:

$$next(\mathcal{T}) = \{\langle \mathcal{T}, \sigma \rangle \mid \sigma \in \Sigma \wedge to(\mathcal{T}) = from(\sigma)\} \tag{12}$$

$next(\mathcal{T})$ is the set of all feasible traces beginning by $\mathcal{T}$ and having one more transition. We also define the operator to to get the last state reached by a trace: $to(\mathcal{T}) = to(\mathcal{T}[|\mathcal{T}|])$

2) Construction of a trace set $\mathcal{U}$: The trace set computation is an iterative process. At the first iteration, the trace set contains all the traces with one transition of the analyzed task. Then the reachable transitions are added. Consequently all the traces of the trace set have the same transition count at each iteration. The process is repeated until the traces of the trace set reach the time scope defined in Section III-D1.

This iterative process is defined by the following recursive equation:

$$\begin{aligned} \mathcal{U}^1 &= \{\langle \sigma \rangle \mid \sigma \in \Sigma\} \\ \mathcal{U}^{n+1} &= \bigcup_{\mathcal{T} \in \mathcal{U}^n} next(\mathcal{T}) \\ &= \{\mathcal{T} \mid \mathcal{T} = \langle \mathcal{T}', \sigma \rangle \wedge \sigma \in \Sigma \wedge \mathcal{T}' \in \mathcal{U}^n \\ &\qquad \wedge \langle \mathcal{T}', \sigma \rangle \in next(\mathcal{T}')\} \end{aligned} \tag{13}$$

Since all the traces are built using the $next$ operator that adds only the feasible traces, all the traces from the trace set are feasible by construction.

3) Optimization of the trace set construction: The computation of the wcrt of the tasks relies on the computation of an upper bound of all the feasible traces of a task. Therefore, all the feasible traces of the task are not interesting, as we only need to reason on the maximal feasible traces (according to relation (10)). From Equation (13), two traces arriving at the same state will be extended the same way (with the $next$ operator). Therefore if two traces terminate with the same state and one trace is smaller than the other (according to the $\leq$ relation) then all the feasible traces following this particular trace will be smaller than any of the feasible traces following the greater trace. It is then possible to upgrade the trace set computation equation by deleting the traces leading to smaller traces. Equation (13) is replaced by the following optimized equation:

$$\begin{aligned} \mathcal{U}^1 &= \{\langle \sigma \rangle \mid \sigma \in \Sigma\} \\ \mathcal{U}^{n+1} &= \{\mathcal{T} \mid \mathcal{T} = \langle \mathcal{T}', \sigma \rangle \wedge \exists \sigma \in \Sigma \wedge \exists \mathcal{T}' \in \mathcal{U}^n \\ &\qquad \wedge \langle \mathcal{T}', \sigma \rangle \in next(\mathcal{T}') \wedge \forall \mathcal{T}'' \in \mathcal{U}^n \\ &\qquad \wedge \forall \sigma' \in \Sigma \wedge \langle \mathcal{T}'', \sigma' \rangle \in next(\mathcal{T}') \\ &\qquad \wedge to(\mathcal{T}') = to(\mathcal{T}'') \wedge \mathcal{T}' \geq \mathcal{T}''\} \end{aligned} \tag{14}$$

4) The upper bound trace is not the worst trace: The trace set constructed in the previous section represent all the possible executions of the task's state-machine. From these trace sets, we can extract the upper bound trace $\mathcal{T}^+$. $\mathcal{T}^+$ is an upper bound (according to (10)) of all the feasible traces of the task.

$$\mathcal{T}^+ : \forall n, \forall \mathcal{T} \in \mathcal{U}^n \mid \mathcal{T}^+ \geq \mathcal{T} \tag{15}$$

It is important to notice that the worst trace is defined as a trace but is not necessarily a feasible trace.

Example 1: In order to illustrate the computation of $\mathcal{T}^+$, we take an example with a state-machine, i.e. PSM, with two states (S_i and S_2) and four transitions (Fig. 3). The two states S_1 and S_2 are connected by the two transitions σ_3 and σ_4. The transitions σ_1 and σ_2 loops on S_1 and on S_2 respectively. The transition times (within brackets) are arbitrary values for this example. The first iteration $\mathcal{U}^1$ contains only the transitions of

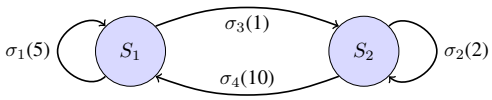

Fig. 3. A simplified periodic state-machine

the state-machine for which the output state has the highest rbf value:

$$\mathcal{U}^1 = \{\langle \sigma_2 \rangle, \langle \sigma_4 \rangle\}. \tag{16}$$

The next iterations are built by adding to the traces the transitions leading to each of the existing states with the highest accumulated time. This process is shown in Fig. 4 where the transitions from S_1 and S_2 in $\mathcal{U}^1$ are tested. The resulting states are both S_1 and S_2, but as different instances. The transition σ_4 is taken, instead of σ_1, because it induces a higher cost. The same happens for σ_2 and σ_3. For $\mathcal{U}^2$, the

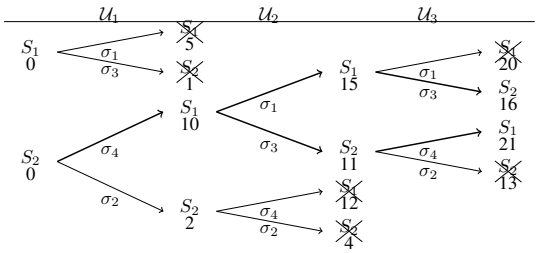

Fig. 4. $\mathcal{U}^n$ values on three iterations

process is similar: σ_1 is added to the trace since the resulting trace cost $\delta(\sigma_4) + \delta(\sigma_1)$ is higher than $\delta(\sigma_2) + \delta(\sigma_4)$.

This technique allows us to build a much smaller trace set than with the naive method (13). Even though it does not contain all the possible traces, it contains the necessary traces to build the upper bound trace. Fig. 5 presents the two traces from the trace set $\mathcal{U}^3$. The two traces interleaves themselves and the upper bound trace is the upper bound of the graph.

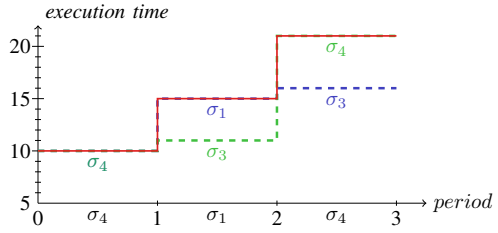

Fig. 5. $\mathcal{U}^n$ values on three iterations using the optimized equation

D. wcrt computation from the upper bound trace

We have previously proposed a method to compute an approximation of the request function of a task by the computation of an upper bound trace of length n. In order to complete our analysis and compute the worst-case response time of every task we have to determine the maximum useful length of the traces. Then we have to adapt the computation of the wcrt to consider the upper bound trace.

1) Maximum trace length: Hypothesis 2 indicates that the release times of the tasks are unknown and Property 1 says that a task's state-machine fires a transition every period. Property 2 indicates that the state-machines are strongly connected. Furthermore, any task can stay in the same state meaning that for every task, it is possible to be in any state in the future. For every task it is possible to reach any state (because the state machine is strongly connected) and to remain in this state. Moreover, not knowing the exact release times it is possible that all the tasks can wake up at the same time. For every task the upper bound trace is greater than all of the task's execution traces. It is then possible to find a worst case critical instant when all the tasks start their execution at the same time with the first transition of their upper bound trace.

Therefore we can say that, for every task, the first deadline is the most time restrictive, response time analysis [19], and we can restrict the study period to the maximum of the deadlines. We define

$$D = \max_i D_i, \tag{17}$$

and we compute the upper bound trace of each task i by computing its trace sets until $\mathcal{U}^{\left\lceil \frac{D}{T_i} \right\rceil}$.

2) Worst-case response time: In order to compute the wcrt, we adapted the usual recursive procedure proposed by [5]. Instead of using the task's execution time, we use the upper bound trace's values: at each iteration of the recursive procedure, we use the the next iteration of the upper bound trace.

Defining $\mathcal{T}^+(t)$ as the request function of the upper bound trace at time t, it is possible to compute the wcrt of the i^{th} task by initializing its value at its first instance's execution time. The wcrt is then recursively incremented with the higher priority instance's execution time.

$$\begin{aligned} \mathcal{R}_i^0 &= \mathcal{T}_i^+(0) \\ \mathcal{R}_i^{n+1} &= \sum_{j \leq hp(i)} \mathcal{T}_j^+(\mathcal{R}_i^n) + \mathcal{T}_i^+(0) \end{aligned} \tag{18}$$

with $hp(i)$ the higher priority task's instance. Two conditions stop this incremental loop: a) whenever two consecutive iterations have the same $\mathcal{R}_i$ value, b) whenever $\mathcal{R}_i^{n+1}$ reaches the task's deadline. In the latter the software (i.e. tasks) is not schedulable. On the other hand, if all the wcrt and lesser than the deadlines (i.e., $\forall i, R_i \leq D_i$), the tasks are *schedulable*.

IV. Performance of the analysis

So far we have proposed a new wcrt computation method for real-time systems by using tasks modeled with state-machines. It is then time to evaluate the computation overhead by taking into account the state-machines of the tasks into the analysis. We intend to prove that such overhead remains low and does not impact the analysis computation time. We also give an intuition of the precision gain provided by this analysis.

A. Complexity of the proposed method

In this section we compare the complexity overhead of the proposed method with respect to the usual task's execution time based analysis. The computation cost is computed in terms of operations per each step.

1) State-machine to PSM: For each state-machine, the PSM construction cost is linear: the PSM is made of the same amount of states and transitions as its corresponding state-machine.

2) Trace computation from the PSM: For each PSM_i of the task i, the amount of computed traces depends on: a) the study period of the system D; b) the amount of states of the PSM $|S|$ and c) the connectivity of the PSM, representing the amount of possible transition per state $\mathcal{C}$ For each PSM_i the amount of traces is bounded by the amount of possible path of length $\left\lceil \frac{D}{T_i} \right\rceil$ in the graph represented by the PSM. The cost for the trace computation from the PSM is then: $|S| \times \mathcal{C}^{\left\lceil \frac{D}{T_i} \right\rceil}$.

3) Upper bound of the traces: The computation cost of the upper bound depends on the amount of traces and the length of the traces. For each task i, the complexity is:

$$|S| \times \mathcal{C}^{\left\lceil \frac{D}{T_i} \right\rceil} \times \left\lceil \frac{D}{T_i} \right\rceil \tag{19}$$

4) Worst case response time: This part of the computation does not induce a computational overhead because it is identical to the classical method.

From a practical point of view, we found out that the computation time of the PSM based analysis is practically instantaneous:

- The overall dimension of the elements is low. The number of states of the PSM is identical to the states of the state-machine given by its developer, which is usually low and often below ten states. The connectivity is bounded by the number of states and is usually lower. The number of iterations depends on the heterogeneity of the periods and of the deadlines.
- The inclusion property 2 allows to reduce the amount of traces to take into account in the analysis.

B. *Gain on the* request bound function

In terms request bound function, the precision gain from our technique quantifies the schedulability improvements.

1) Practical observations: The method presented in this paper studies the schedulability of a task set by computing the wcrt of each task, scheduled by a fixed priority scheduler. The wcrt computation of each task τ_i depends on the higher priority tasks and more particularly of the *request bound function* of all the higher priority tasks. The precision of our method comes from a precise computation of the *request bound functions* of the higher priority tasks using their internal state-machine.

In a task model without state-machine, the "worst" transition is executed at every period. In out task model, this "worst" transition is executed at the first period and then the other transitions are taken into account. The precision further increases with the complexity of the state-machine and with its execution times variability. It means that the more the upper bound trace is computed for a long time, the better the precision is. We have shown that, under our hypotheses, only the first deadline matters for the schedulability analysis. As a consequence, the precision of our technique also increases with the deadline of the component we are computing the wcrt for.

These two observations are interesting from a practical point of view, because they correspond to real-time systems best practices. For systems with deadline on request tasks, we often set the priorities of the tasks depending on their period (with a Rate Monotonic scheduler). In the case the deadline on the tasks are lesser than their period, we prefer priorities defined from their deadlines (using a Deadline Monotonic scheduler). By setting the priorities using these good practices, the *request bound function* of the higher priority tasks are computed on a longer timespan and thus increases the precision gain.

2) Indicator: The precision gain of our method is quite easy to compute with relation to the purely task oriented technique that do not take into account the state-machines. For each task τ_i with a deadline D_i, the precision gain is:

$$\int_0^{D_i} rbf_i^* - \mathcal{T}_i^+ \tag{20}$$

with rbf_i^* the *request bound function* of the considered task.

This process is shown in Fig. 6: the worst case appears at the first step when the transition σ_4 is fired, equaling the *request bound function* value. However, in the second step, the upper bound trace fires the σ_1 transition, which is not the most time consuming transition of the state-machine but is still an upper bound of the task's execution time. Thanks to the precision provided by the state-machine, we have gained five time units at the second step. At the third step, the difference between the two traces is further increased providing an even lower upper bound of the task's overall execution time.

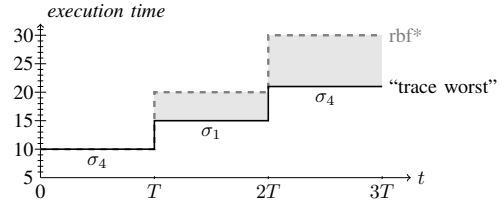

Fig. 6. Comparison between $\mathcal{U}^n$ values and the "classical" *request bound function*

V. Experiments

We have applied our framework to trace analysis and wcrt computation on a real robotic use-case.

A. *Robotic platform architecture*

The Pioneer 3-DX is a wheeled robot equipped with an internal computer as a controller interface. Its stock capabilities allow it to move around through its wheels speed controlled and avoid some obstacles using its sonar range finders. Its engines are sufficiently powerful to move around outdoors on reasonably rough terrains and its small size allows it to find its way into corridors. In order to increase its capabilities, we have plugged a Hokuyo UTM-30LX[1] laser scanning range finder and an Asus Xtion Pro Live[2] depth and colour camera.

A traditional laptop computer with an Intel i5 architecture is used for our experiments. This embedded computer run Linux kernel patched with Xenomai, which allows the user to create hard real-time tasks on top of a non-real-time kernel. Xenomai receives all the system interrupts and checks if it has to handle them. If not, it passes the interrupts to the Linux kernel. A fixed priority scheduler is built in Xenomai to achieve true real-time performances.

B. *Component architecture*

We have developed a Navigation, Guidance and Control component-based architecture meant to be run on mobile robots [20]. This architecture is made of three main components (Navigation, Guidance and Control) designed to compute a path to follow for the robot, to avoid obstacles, and to drive the robot's engines (see Fig. 7). Other components have to be connected in order to provide sensor information, robot's status, or high level planners. Moreover, we have connected a

[1] http://www.hokuyo-aut.jp/02sensor/07scanner/utm_30lx.html
[2] http://www.asus.com/Multimedia/Xtion_PRO_LIVE/

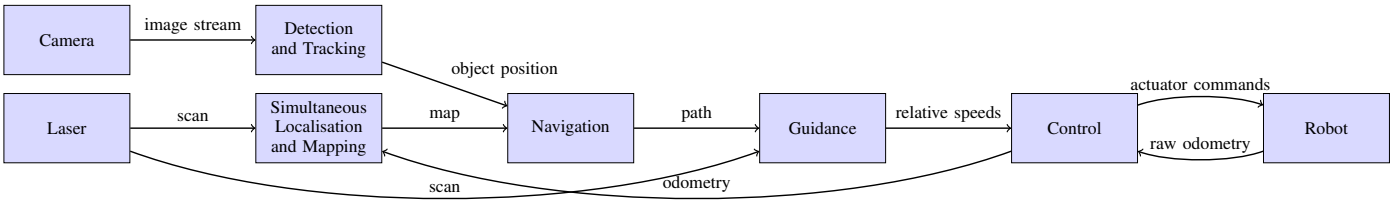

Fig. 7. Simplified component-based architecture running on the robot

Simultaneous Localization And Tracking (SLAM) component for the robot to build a map of its environment via a laser range-finder and the robot's odometry. We also have embedded an image detection and tracking component to a path planning task in order to have the robot go where the detected object is. The Detection and tracking component detects and tracks an object on a picture stream. In this particular example, without loosing generality only the Detection and tracking component is modeled with a state-machine, while the other tasks associated to components are modelled as simple tasks.

C. Illustration of the upper bound trace calculation for the detection and tracking component

The state-machine of the detection and tracking component (see Fig. 8) has four states: the *Initialize* and the *Cleanup* are respectively creating and cleaning the memory needed by the component; the *Detect* state, through which the component has to find an object on an image stream; the *Track* state is for tracking the position of the detected object on the picture stream. The *Track* state also provides the position of the tracked object relatively to the camera's position. As usual on image processing programs, the object detection costs more computation time than the tracking, because it has to scan the whole image for finding the object. When the object is detected, the tracking needs only to check around the detected position to follow its movements. In this state-machine, we have represented the worst execution times of the state's *run*, *entry* and *exit* parts.

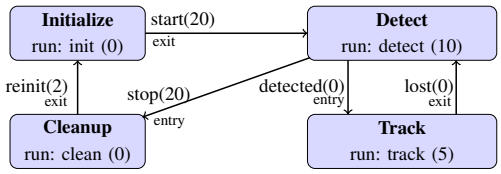

Fig. 8. State-machine of the Detection and tracking component. Transition duration correspond to the sum of the *exit* of the previous state and the *entry* of the next state. worst-case execution times are expressed in ms

The corresponding PSM of the Detection and tracking is shown on Fig. 9. All the functions carried by the states are moved to the state-machine's transitions and the execution times of the transitions are adapted accordingly. For example, the transition from *Detect* to *Cleanup* carries the execution times of the *detect* (run: detect = $10ms$) and *stop* functions ($20ms$), i.e. $30ms$. The computation of the traces for this component and the resulting upper bound trace are illustrated on Fig. 10 where the rbf of some feasible traces are drawn so as the upper bounding rbf.

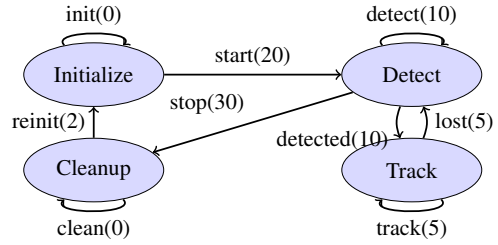

Fig. 9. PSM for the detection and tracking component

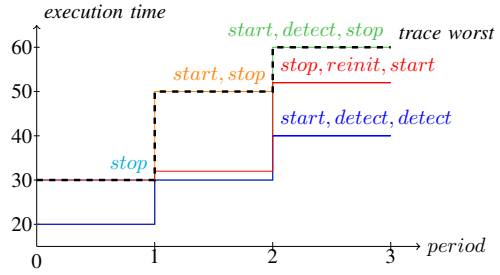

Fig. 10. Some execution traces

Table I presents some possible traces along with the upper bound trace and the classical wcet computation (i.e., taking the max of all the transitions at each step). We can see that our approach produces an important gain with respect to the classical wcet computation, then leading to a less pessimistic, thus more precise response times for the tasks. For the first iteration our approach provides no gain over the classical method. The next iterations show the gain can go up to 42% and stabilises to 32% for this state-machine.

TABLE I. rbf VALUES OF SOME TRACES OF THE DETECTION AND TRACKING COMPONENT, ALONG WITH THE UPPER BOUND TRACE AND THE WCET VALUES.

iteration	1	2	3	4	5
*track**	5	10	15	20	25
*detect**	10	20	30	40	50
start, ...	20	30	40	45	50
*(start, stop, reinit)**	20	50	52	72	102
*(stop, reinit, start)**	30	32	52	82	84
*(reinit, start, stop)**	2	22	52	54	74
⋮					
upper bound trace	30	50	52	82	102
cumulative wcet	30	60	90	120	150
gain (in %)	0	17	42	32	32

D. Architecture schedulability

In order to evaluate the schedulability of our architecture, we compute the wcrt of all the components. We first computed the schedulability using the classical approach, where each component has a unique wcet. Then we applied our method, that consists in computing the upper bound trace of each component, and then applying the modified version of the wcrt computation , Equation (18).

Table II shows the real-time characteristics of the several components of the architecture, and the results of the schedulability analysis. For each component there is, from left to right: its priority (the higher the value, the higher the priority), its wcet (using the classical task-based reasoning), its wcrt (limited to the component's deadline ; named wcrt*) using the previous wcet, its wcrt based on the upper bound trace computation (named wcrt+), and its period. All the task's deadlines are equal to their period.

TABLE II. REAL-TIME CHARACTERISTICS OF THE ARCHITECTURE'S COMPONENTS

component	prio.	wcet	wcrt*	wcrt+	period
Robot	8	16	16	16	100
Control	7	3	19	19	100
Guidance	6	12	31	31	100
Laser	5	22	53	53	150
SLAM	4	30	83	83	150
Camera	3	10	93	93	250
Det.&Track.	2	30	237	237	250
Navigation	1	30	**307** (338)	**297**	300

The classical analysis cannot prove that this architecture is schedulable because the Navigation component hits its deadline: the worst-case response time iterative computation has been stopped when its value (307ms) has hit the deadline (300ms). If we do not stop the iterative process, for this precise task set, it eventually arrives at a fixed point of 338ms. With the method we proposed in this paper, we computed a less pessimistic but still not optimistic wcrt for the components: the Navigation component has a wcrt+ of 297ms which is lesser than the deadline. As all the wcrt+ are lesser than the tasks periods so the new technique proves the architecture is schedulable, reducing the pessimism from classical response time analysis techniques.

VI. CONCLUSIONS

In this paper, we have presented an innovative method for analyzing state-machine-based software architectures. It computes more accurate and realistic execution times than the purely task oriented methods by making use of the task's state-machines. The presented method first models the state-machines as Periodic State-Machines. From the PSMs, execution traces are built and a maximum bound, namely the upper bound trace, is computed for each task. Then a new fix-point formula has been defined to computate the worst-case response times from the upper bound trace of each task. The precision gain brought by this technique is even better with complex systems with complex and uneven state-machines. The experiments made on an embedded computer for a mobile robot running a Navigation, Guidance and Control architecture on a real-time Linux kernel showed that our approach allows to conclude on the schedulability of the architecture, while the classical approach concluded otherwise.

For future developments, we plan to investigate several possible enhancements. We want to take into account the middleware: for now, we have considered that its protocols, such as the communication interface, takes sufficiently few time to not be considered in the computation. A second possibility is to adapt our technique for multicore or multiprocessor devices since they are common nowadays. Since our method computes the execution times of the tasks per iteration, we want to adapt other scheduling analyses such as EDF to use the PSMs. We also plan to investigate soft real-time applications since we can use probabilistic execution times to compute the execution traces and thus a probabilistic schedulability analysis. Finally, we also want to provide tools to automate the design, the development and the analysis of component-based architectures using the Mauve DSL [21].

REFERENCES

[1] T. Lens *et al.*, "Investigation of safety in human-robot-interaction for a series elastic, tendon-driven robot arm," in *IROS*, 2012.

[2] P. Rybski *et al.*, "Sensor fusion for human safety in industrial workcells," in *IROS*, 2012.

[3] A. Nakamura *et al.*, "Error recovery using task stratification and error classification for manipulation robots in various fields," in *IROS*, 2013.

[4] S. Pathak *et al.*, "Ensuring safety of policies learned by reinforcement: Reaching objects in the presence of obstacles with the iCub," in *IROS*, 2013.

[5] C. L. Liu *et al.*, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *JACM*, 1973.

[6] M. Spuri, "Analysis of Deadline Scheduled Real-Time Systems," Research Report, 1996, projet REFLECS.

[7] L. Sha *et al.*, "Priority inheritance protocols: An approach to real-time synchronization," *Computers*, 1990.

[8] M.-I. Chen *et al.*, "Dynamic priority ceilings: A concurrency control protocol for real-time systems," *RTS*, 1990.

[9] T. P. Baker, "Stack-based scheduling of realtime processes," *RTS*, 1991.

[10] H. Chetto *et al.*, "Dynamic scheduling of real-time tasks under precedence constraints," *RTS*, 1990.

[11] M. Klein *et al.*, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Springer, 1993.

[12] M. Stigge *et al.*, "The Digraph Real-Time Task Model," *RTAS*, 2011.

[13] H. Zeng *et al.*, "Schedulability Analysis of Periodic Tasks Implementing Synchronous Finite State Machines," *ECRTS*, 2012.

[14] M. Quigley *et al.*, "ROS: an open-source Robot Operating System," in *ICRA, OSS Workshop*, 2009.

[15] P. Soetens and H. Bruyninckx, "Realtime hybrid task-based control for robots and machine tools," in *ICRA*, 2005.

[16] C. Rochange *et al.*, "OTAWA: An Open Toolbox for Adaptive WCET Analysis," in *IFIP, SEUS Workshop*, 2010.

[17] J. Hansen *et al.*, "Statistical-Based WCET Estimation and Validation," in *WCET*, 2009.

[18] S. K. Baruah *et al.*, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor," in *RTS*, 1990.

[19] M. Joseph *et al.*, "Finding response times in a real-time system," *The Computer Journal*, 1986.

[20] N. Gobillot *et al.*, "A Component-Based Navigation-Guidance-Control Architecture for Mobile Robots," in *CAR*, 2013.

[21] ——, "A Modeling Framework for Software Architecture Specification and Validation," in *SIMPAR*, 2014.