Evaluation of a Real-Time Monitoring Framework

Thomas Robert, Matthieu Roy, Jean-Charles Fabre

Evaluation of a Real-Time Monitoring Framework

Thomas Robert, Matthieu Roy, Jean-Charles Fabre

LAAS-CNRS, 7 av. Colonel Roche 31000 Toulouse Cedex 4, France

Abstract:
Many algorithms exist to generate real-time run-time monitors. This paper focuses on the assessment of an implementation of a real-time monitor designed to handle timed automata specification as input. The monitoring algorithm has been analysed to propose a simple and yet realistic assessment method. The feature measured is the CPU-time overhead introduced by the monitor, per event. Our methodology proceeds by first providing an analytical formula to point out the main sources of overhead. According to this formula, we derive the main parameters of a specification that influence the induced overhead. Then, "real-life" specifications are analyzed to get a realistic range for these quantitative parameters. They are used to generate equivalent applications to measure the overhead. Finally, the hypothesis introduced by the analytical formula on the actual overhead is discussed, with respect to the empirical results.

Keywords: Run-time Monitor, Evaluation, Parameter estimation, Real-time Systems.

1. Introduction

In critical real-time systems, there is a strong need for automated processes that improve the reliability of software in operation. Many static methods, such as model checking, can be used to verify the system correctness and feasibility on models. Nevertheless, both for complexity and coverage reasons, these static validations can be complemented with *run-time verification*. Run-time verification consists in inserting components in the system in order to check at run-time that the system does not fail, i.e. complies with its specification [1]. Run-time monitors and failure detectors are examples of such components.

A run-time monitor focuses on assessing the correctness of the dynamics of the system. By definition, a system is said to have failed when its observed behaviour does not match its expected behaviour. Given a specification describing correct behaviours, the detector is in charge of signalling any failure accurately and timely. Note that in the real-time context, detection *latency* is as much important as detection *accuracy*.[1]

This paper is a follow-up of our previous work that focussed on the implementation of a real-time run-time wrapping framework for the real-time operating system `Xenomai 2.3`. The paper is organised as follows: first, the monitoring framework is recalled in order to identify the strategic points to assess it empirically. Then, the experimentation strategy is detailed to provide all the elements needed to reproduce and adapt it to another framework. Finally the results are discussed with respect to the alternatives available to implement the same kind of services.

2. The monitoring model.

This paper focuses on an empirical assessment of the run-time monitoring framework RTRV[2]. The aim of this paper is to get some evidences on the kind of overhead introduced by our monitoring framework. The sources of overhead have to be identified before carrying out the experimentation. Beside the overhead, the detection latency with respect to failure occurrence time is a major feature, for such monitors. Those two features define its efficiency. For such a system, the impact on interruption handling latency is another important feature to be measured. Nevertheless, we focus on developing a simple experimental process to estimate the overhead that can be expected, depending on the complexity of the specification to monitor. The proposed monitor is in charge to check at run-time that the behaviour of the system complies with a specification provided off-line. Although such systems are more frequently used lately, their performances for an actual real-time application would be questionable if the delivered service is not predictable with respect to two parameters: its *maximum detection latency*, and the *overhead* introduced by the tracking of the system behaviour.

2.1 The system specification: a timed model.

We assume that a timed automaton provides the description of expected behaviours at run-time. Such models are well spread in the community of the formal languages used to describe Real-Time systems. More precisely, it is a model for which many formal validation methods exist. Thus, we can assume that the descriptions provided in this format do really represent what the developers expect from their system. The monitor is generated from this model and ensures basics but essential properties along the detection process.

[1] In addition, detection latency may have also a strong impact on the system dependability.

[2] Cf. `www.laas.fr/~trobert/RTRV/`

Timed automata are a particular case of transition systems in which transitions are labelled with enabling conditions and clock resets. Roughly speaking, each transition is active if and only if the associated condition is true. A condition on a transition is defined by a set of equations on variables representing clocks. Once a transition is enabled (its condition is satisfied), it can be fired. Moreover, transitions are labelled with reset actions, which define the clocks to be reset when the transition is fired.

Because of the tight relations between clocks and transition, Timed Automata are rather complicated to analyze directly at run-time. As an example, the system may perform actions whereas no final state can be reached any longer. This scenario happen whenever: 1) there is a path along edges leading to some final state, but for any of those path there is at least one disabled edge. Whereas the state is connected to final state, it would be impossible to reach any of them.

Definition 1: Timed Automata

Timed automata are defined by:

- A set of discrete states L, called locations. They represent functional states.
- A set of clock variables represented by a clock vector: Vect. These variables represent the different clocks of the system and all increase at the same rate.
- A set of events Σ. They correspond to observable activities of the system.
- A set of edges E between locations. Each edge is labelled by three elements:
    - The event emitted by the system when crossing it;
    - The condition on clock variables to enable this transition;
    - The set of clocks that are to be reset when this edge is crossed.
- A location that defines the initial location.
- A set of locations that defines the final locations.

In order to check at run-time that the system complies with its specification, one needs to detect as soon as possible any divergence between the current behaviour and the expected ones defined by the timed automaton. The model underlying the run-time verification will be a transition system which is synchronized with the system activity. Nevertheless, the model that will be synchronized on the system execution is not the timed automaton itself.

The challenge with such kind of specifications is that timed automata "allow" more behaviours than what is actually expected. The conjunction of the set of final states and the automaton itself entails that some behaviours will be rejected later even if they are locally allowed. In order to clarify this point, let us consider the example of a simple resource manager specification.

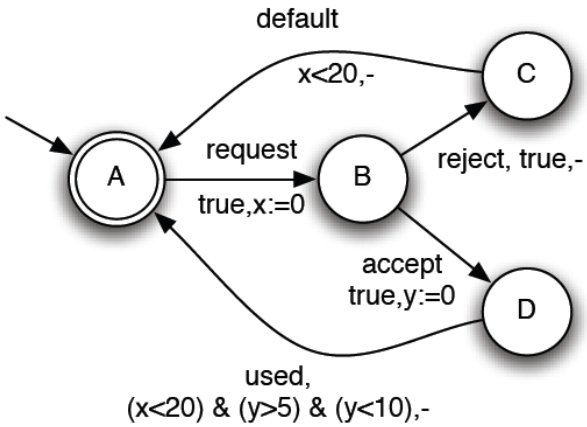


Figure 1: Specification of a resource manager

In this example, the final state is A. The "accept" and "reject" events are both enabled from state B, firing the "accept" condition may in some situations put the automata in a state where the final state can no longer be reached. Assume that the transition on "accept" is crossed 16 time units since the "request" event occurred, then the automata enters location D with x=16 and y=0, and, letting time elapse, the system won't be able to satisfy both conditions (x<20) and (y>5) at the same time.

In order to cope with this problem, we transform the specification in a normal form in which this kind of scenario does not occur, i.e. failures can be detected as soon as they occur. The problem lays in the fact that even if in a given state a transition is enabled, it does not mean that this transition is on a path to a final state. This restriction is a consequence of the requirements for reaching a final in any state of the automaton.

From the example, as soon as an "accept" event occurs when the clock x belongs to [15,+∞[, the system cannot match its specification. As we highlighted in this section, the main issue with a timed automata model is that its transition system cannot be used as is for run-time verification. Even if the state and the value of the clocks are known, it is extremely difficult to determine at run-time whether an event can be accepted, given the state of the system.

2.2 Time abstraction to save computations

To be able to verify at run-time a specification given by a timed automaton, we transform it in a more suitable model known as a time abstraction. Time abstractions of timed automata have been extensively used in model checking to test reachability properties [3]. We use it to produce, from a timed automaton, a model that can easily be used at run-time. In this section we briefly recall timed

abstractions definitions and their use in the context of run-time verification.

A time abstraction of a timed automaton $\mathcal{A}$ is a labelled transition system where the timed automaton conditions on transitions are reported on states. In this model, any transition leaving from one node is always active, and the reachability of a final location can be determined off-line. The time abstraction ensures that the set of generated traces (i.e. the specification of the system) is the same as for the initial timed automata.

In a time abstraction, there are two kinds of transitions: time transitions and event transitions.

As an example, for the specification of the resource manager (Figure 1), the corresponding time abstraction splits the state B in three sub-nodes: B with x in [0,15[, B with x in [15,20[, and B with x greater than 20. Semantically, it corresponds to the example given above that shows that being in location B with x≥15 does not have the same meaning than being in B with x<15. The corresponding nodes are connected by time transitions leading from the former to the later. In general, for each location, the time abstraction will partition the space of clock values in a finite set of zones.

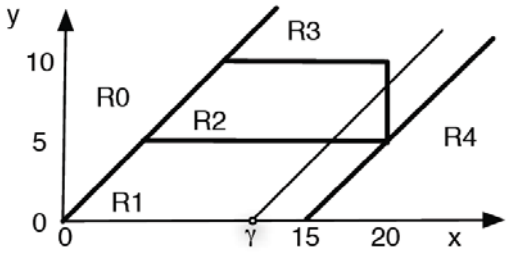


Figure 2: Partition of the clock values for D

This partition is structured in parallel "lines" (R1→R2→R3 is an example of such a line). This property is a direct consequence of the identical rate of time elapse on each clock. As we shall see, it plays a major role in the monitoring engine to catch deadline failures.

The graph obtained as a result of the time abstraction is then analysed to delete node that are forbidden –in the example (D,R3) is forbidden–, or not reachable –like (D,R0). Finally any transition not present in this graph is by definition forbidden.

Finally, this model ensures that a local decision can be taken for each event as long as the "current" node is known. It will be used at run-time to perform the run-time verification of the compliance of the system to its specification.

3. The monitoring engine.

As presented above, time abstractions can be computed off-line, and then be used at run-time by synchronizing it with the system activity. Nevertheless, the monitoring engine is not a simple, passive trace reader. Many nodes of the time abstraction are not stable: the system may stay in these states only for a bounded duration. For instance, if we consider the discrete location[3] D, the maximum time for which the system can remain inactive is defined as TtF (time to failure) as shown on Figure 3. Each time an event occurs and is accepted by the monitor, the monitor computes the maximum amount of time during which the system may remain silent. For instance, on this example, assume that "accept" occurred exactly 12 time units after the "request" event. Thus the system has to emit an event within 8 time units to avoid a timing failure, as represented in Figure 3.

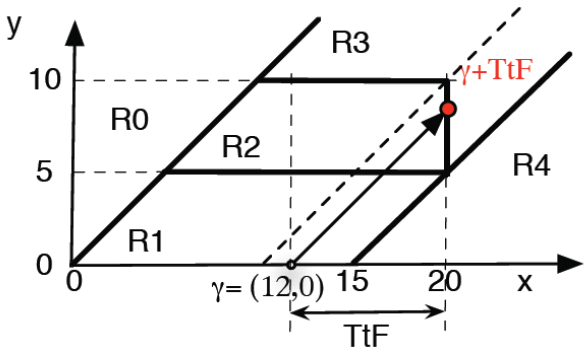


Figure 3: Predicting deadline miss

2.1 The complete monitoring process.

The computations at run-time can be grouped in six main steps, corresponding to model manipulation, and system calls. The same processing is performed for each event:

1. Shield the monitor execution from pre-emption (using a dedicated system call).
2. Compute the impact of time elapse on the current state of the system.
3. Check whether the event being intercepted is allowed or not in the current system state.
4. If the event is allowed, then fire the corresponding transition and update the current state value. Update the clock vector using the resets attached to the fired event transition.
5. Compute the new clock vector corresponding to the system state right after the event

[3] A location is a discrete state of a timed automata. A,B,C,D are the locations in the example automaton.

occurrence. In each node, the maximal allowed values for each clock is known

6. If a deadline exist in the current node, set a timer to wake at the deadline miss time (using a system call).
7. Un-shield the execution of the monitor from pre-emption (using a dedicated system call).

As presented above, unless there is a deadline miss, the monitoring system is only activated when an event occurs. Moreover, most of the introduced overhead comes from its activity along a valid behaviour.

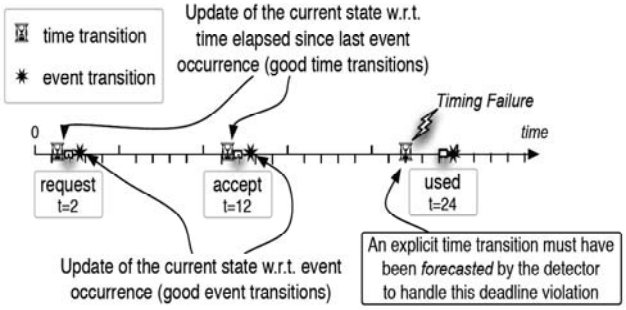


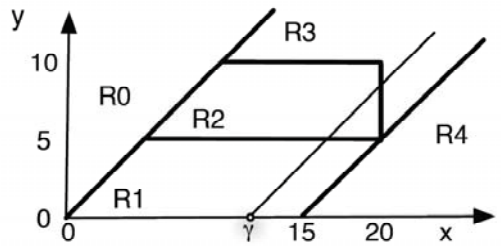
Figure 4: Activation times of the monitor.

3.2 Theoretical profile for cpu-time overhead.

Before the actual measurements of the different latencies and/or overheads introduced by our system, we analytically identified the relationships between the complexity of a specification and the monitor overhead. Given the knowledge of the algorithm used to perform the run-time verification, we provide in this section the relationship between the overhead of the monitoring algorithm and the complexity of a specification.

With respect to algorithms used in steps 2,4 and 5, we introduce two parameters that can be computed from the time abstraction of any timed automaton, namely its *branching degree* Br and the length of the *longest path with time transitions* only TP_{max}.

In step 2, the graph of the time abstraction is browsed to find the up-to-date value of the current node. The path followed for this task is made of time transitions only. Recall that from any node, the states, that can be reached crossing time transitions only, are organized along a linear topology ("lines").



Consider the following example: the monitor intercepts "used", and the last estimated state was (D,γ), where $\gamma=(x_0,0)$ represents the last estimation of the clock vector. If *d* units of time elapsed since the last event occurrence, the monitor will jump from clock region R1 to R2, and possibly to R3, following the first diagonal (Δ). Thus the size of the longest path of time transitions bounds the number of nodes considered along step 2.

In step 3, the list of locally available events needs to be browsed. The cost of this step increases linearly with the branching degree of the node, i.e. the number of event transitions leaving from this node.

The overhead introduced in the fourth and fifth step will only depend on the number of clocks used in the automaton. This property is true for the fifth step as the moves along the nodes connected by time transitions can be computed off-line: the deadline is defined in the last node of the "line" of time transitions leaving from the current node. For instance, the transition between the time regions R2 and R3 allows obtaining the knowledge about the most urgent deadline for R1.

Steps 1,6 and 7 involve for each event at most three "system calls". The maximal duration of these three system calls will be denoted ΔSysCall

If we sum up the conclusion of this subsection, we can establish a profile that gives an idea of the overhead that can be expected w.r.t. the complexity of the specification. Notice that this profile is not there to predict or give quantitative values of the introduced overhead given the parameters linked to a specification. The aim of the profile is to guide experimentation in order to find evidence on the usability of the monitor. Thus the time overhead ***Oh*** introduced by the monitor at run-time is bounded by an analytical formula that involves TP_{max}, Br, n (the number of clocks in the timed automaton) and ΔSysCall and four unknown constants (denoted M0, M1, M2 and M3) related to the hardware and the real time operating system:

$$Oh \leq TP_{max}.M0+Br.M1+n.M2+n+M3+\Delta SysCall \quad [1]$$

The remainder of this paper will try to identify which parameters are the most important. The main problem with this profile is the lack of direct link with the automaton provided as input. This is particularly true for the parameter TP_{max} .

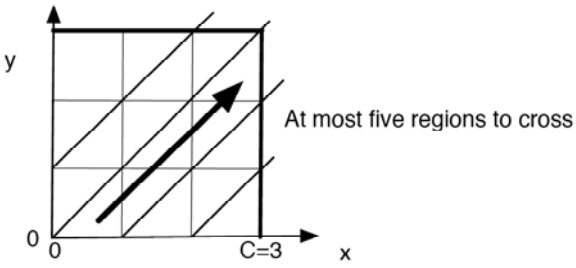
2.3 Estimation of (B,TP_{max}) from visible parameters

For a given timed automaton ***A***, the following parameters are used to estimate at first glance, i.e. without computing the time abstraction, the cost of monitoring an application depending on the timed automaton:

- B, the maximum number of edges leaving from a location.
- C, linked to the timing constraints as follows: given that timing constraints are defined using fraction of naturals, we can normalize the time unit used in the automaton to use only naturals. Let $C_1,...,C_k$ be the constant used in the enabling condition attached to the edges of the automaton with such normalized constraints. Then let C be the least common multiple of $C_1,...,C_k$.
- n, the number of clocks involved in the automaton.

As the number of enabled edges is restricted by clock constraints, the number of edges leaving from one location, B, is at least greater than the number of event transitions enabled, in any location, at any time: (Br ≤ B). The link between C and TP_{max} is much more complicated. First notice that for scattered constant $C_1,...,C_k$, small changes on one of the C_i may entail a combinatorial explosion of C. The problem is that this explosion is often directly linked to a state explosion of the time abstraction as well.

In the region graph, the clock space related to a single location is divided in the worst case according to the hyper-planes $x_i=k$ and $x_i - x_j =k$ for k lower than C. In dimension 2, it generates the grid presented below. In these conditions TP_{max} is bounded by the number of elementary time regions that can be crossed by lines along the vector (1,1,1….1). With two dimensions, it can easily be visualized: the largest number of regions can be crossed close to the 1st diagonal.



Nevertheless the example, an efficient time abstraction generator would generate path of length 3 – at most . It is far smaller that n^2.C =16—n= 2, C=4.

The bound provided by C, is a very rough estimation of what happen with TP_{max}. Moreover, as we are only interested into reachable and correct nodes, many nodes involved in very long time transition paths can “disappear” once the time abstraction is truncated to fit our needs. Thus instead of relying on C, we preferred to measure actually the parameter TP_{max} . Computing the time abstraction and its related parameters seems a better strategy.

The relevance of monitoring the time abstraction instead of the automaton highly depends on the detection gain introduced by clock regions. In the case of the run-time monitor, the gain is of at most 5 time units. Nevertheless, it represents a quarter of one complete round of the resource manager. The time abstraction can be analyzed to decide that monitoring the time abstraction is more interesting than monitoring the automaton itself. Beside the computation involved in the current state and next deadline estimations for each new event, the monitor makes three system calls. The empirical analysis is designed to identify the relative cost of system calls, β, with respect to state estimation computations, α, in the overall overhead ($Oh=\alpha+\beta$). Such measurements will be performed for different values of TP_{max} and Br. The experimentations are meant to obtain these measurements easily.

4. The experimental setting.

In the previous section, the relationship between the monitoring overhead and the parameters Br and TP_{max} has been highlighted. It turns out that the overall overhead is made of two different kinds of computations: system calls and graph browsing. Whereas the second source of overhead can be optimized, the first is directly related to the efficiency of the underlying real time operating system (which is `Xenomai v2.3` in our case). Thus the first source of overhead is the minimal price to use the monitor The experimental setting is made of two steps.

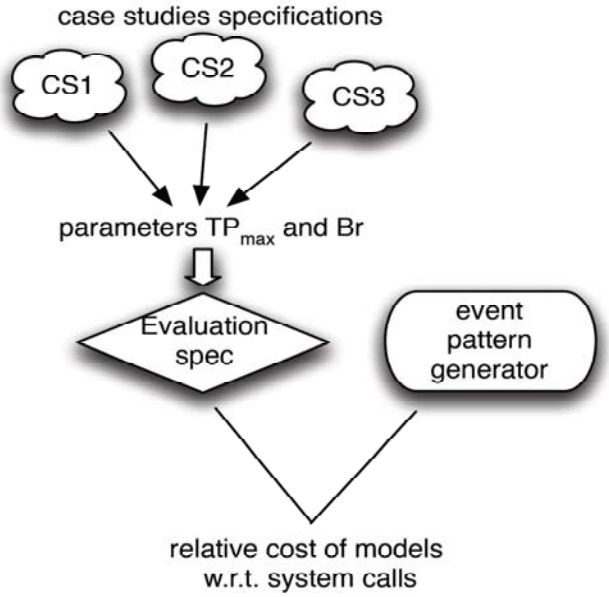


Figure 5: Experimental process.

First, a sample of case studies has been gathered from various sites of well known model checkers that accept timed automaton as inputs [9,10]. Given this sample, intervals of interest have been defined for parameters Br and TP_{max}. Then fake specifications have been written, together with dedicated

applications with respect to these values. Such pairs (application, specification), would cause the monitor to follow predefined execution path with parameters Br and TP_{max}. Thus the overhead measured along these paths is likely to represent the overhead we could have measured on a real application. The complete process is summed up in Figure 5.

4.1 Collecting the parameters of the case studies.

Most of the model checkers propose several academic case studies together with real-life case studies drawn from system actually produced --less frequent. Such specifications have been analyzed to get the values of TP_{max} and Br. This part of the process has been done semi-automatically but can be improved.

Let us first analyze the main difference between academic examples and real-life case studies. In most of academic examples, the systems are made of identical components interacting in a peer-to-peer fashion: Fisher mutual exclusion, CSMA/CD, token ring networks. We are convinced that real-life examples are less symmetric. For instance, one can consider the interactions between a driver, an asynchronous application and a real time application bound to both. Nevertheless, the case studies available do not exhibit such architectures. Thus, we gathered parameter values from freely distributed case studies that have often a regular structure. The main issue with regular architecture lays in the concurrency introduced by weakly synchronized components. It leads to consider an arbitrarily high value for Br corresponding to the number of components involved in the system. Moreover, by coupling components with distinct periodicity, it is likely that C would explode exponentially. The specifications considered to get those values are the following:

- Single component in CSMA/CD, Fisher's mutual exclusion, audio bus controller protocol.
- Few cooperating (about 3) components for CSMA/CD.

The motivation for such choices is twofold. First, the specification provided as input for the monitoring process can be considered as the result of a model checking process that proved that whenever all the components behave correctly the system is correct — that is the case of most protocol specifications. With component off-the-shelf, it may happen that the specifications of each component cannot be changed easily without rewriting completely the component. Then the monitor could be used as a wrapper in order to enforce a fail silent model for the assembly. In this context the size of the considered systems would not be larger than few components. Notice that for the largest systems, the state space was as large as thousands of states and thousands of transitions. The following table presents a summary of values observed for various specifications:

Br		TP_{max}	
Min	Max	Min	Max
2	8	2	16

First, the minimal values of both parameters correspond to simple components for which using the proposed monitor is irrelevant. The descriptions of non-trivial timed behaviours often involves automaton with at least one time transition path of length three. Nevertheless, even for very complex systems, the maximal branching degree is still rather small. The maximal values for TP_{max} have been obtained for compositional specifications with highly independent components. There, the clock space is highly fragmented due to the nature of the application (mutual exclusion). The ranges observed for these two parameters will still be considered for the experiment.

4.2 Generating the pair model / real-time code.

The experimental part of this work has mostly be automated so that experimentation can be run on other platform supporting `Xenomai`. The problem of deploying actual applications is rooted in its cost. It was too costly for each of these specifications to implement those protocols for real and then plug the detector. Moreover, the strategy to assess the monitor would be unclear as we are not interested in average performances. Thus compiling the overhead along a complete execution would smooth all the measures. Instead of considering a "real-life" model, an artificial specification is generated for which the parameters can be tuned at will. Thus we can generate a model with the same topological properties than in the most complicated part of each of the case study specifications. This will allow collecting at low cost a large number of measures for parameters α and β -- recall that they represent the state computation overhead and the system calls overhead respectively, as presented in the analytical approximation of the overhead per event.

Given the parameters Br and TP_{max}. We generate a fake time abstraction with the following requirements:

- The system accepts in each node at least Br events.
- The last node has a bounded duration for any of its clocks.
- Any event sends the system back in node 0 and reset all the clocks.
- From node 0 there is a unique path of time transitions of length TP_{max}.

Thus the model described in Figure 6 is generated for several pairs of parameters Br and TP_{max}. Notice that TP_{max} grows faster than Br with the complexity of the model.

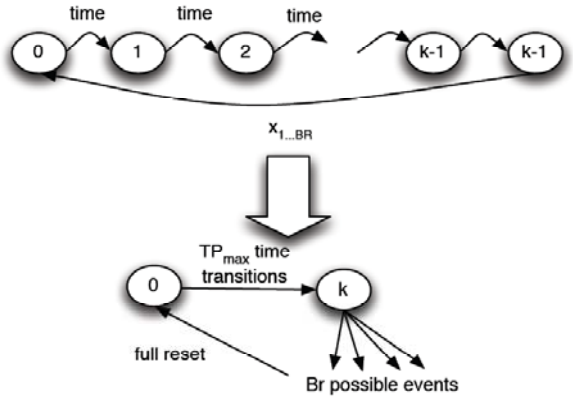


Figure 6: The model used to measure the overhead.

In order to ensure that using this model to assess the overhead is realistic, we need to generate deterministic execution paths. Here the idea is to generate periodically one of the (x_i)s, say x_p, so that the system is in the node "k" when x_p occurs.

The model above will be generated so that at any time in [T-Δ,T+Δ] from the last event, the current node would always be k. Thus the application that will be used as reference to measure the overhead will only have to generate the "last event" among the x_i periodically at time kT.

The time constant Δ is used to ensure that the generated code activates the right path in the artificial model. Along this path, the monitor would generate the same kind of overhead than it would have for the actual application. The parameter Δ will be chosen with respect to the context switching latency of the operating system `Xenomai 2.3`. If the application is configured to trigger an alarm at time k*T, and Δ is ten times greater than the usual context switching latency, then only the desired path would be followed in the artificial model.

The experimental platform was made of a Pentium© III at 800Mhz on which we used the real time `Linux Xenomai 2.3.0` (`Xenomai` is a patch for the `Linux` kernel, currently used with the version 2.6.19.) The distribution used to install the operating system is a `Debian Sarge` with only few modules. The interested reader can find the monitor generation framework on-line[4]. The measurements have been done under several different settings for which the following parameters have been changed: TP_{max}, Br.

Notice that load conditions shall not interfere with the application and monitor behaviours. Unlike load conditions, the base time unit modifies the ratio between T, the application period, and Δ, the duration of the last time abstract node. Thus the relative cost of systems calls used in the monitor increases with the application frequency. It can be used to determine frequency bounds for the relevance of monitoring the application.

The measurement of the overhead under different load conditions is more related to the assessment of the underlying operating system, than the one of the monitoring framework. The load conditions interfere with the application and monitor behaviours together by changing the reactivity of the operating system.

4.3 Measurement points in the monitoring engine.

In order to estimate the scale factors binding M0, M1, M2, M3 and ΔSysCall, the following measurements are carried out for each event occurrence. Only the mean value is kept: storing each event overhead would require large tables that may disturb the overhead computation. The steps durations are measured incrementally: first, only step 1, then steps 1 plus step 2 and so on. It allows estimating the average cost of each step. The duration of the last step cannot be measured with high confidence, so we do not integrate it to the computation. Instead, it is assumed it lasts as long as step 1 does.

Figure 7 represents the average value of the total measured overhead with respect to the parameters Br and TP_{max} for the same number of clocks –2 clocks.

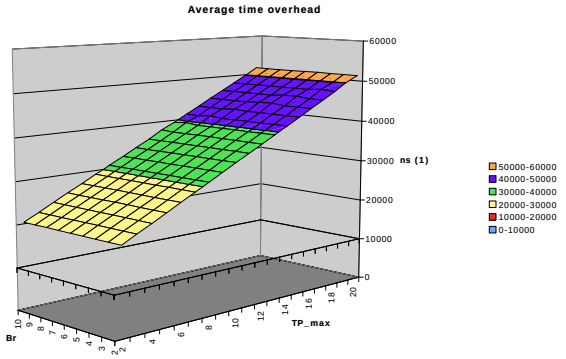


Figure 7: Measured overhead w.r.t. different pairs (Br,TP_{max}).

The graph plotted in Figure 7 is provided in order to decide which parameter between Br and TP_{max} has the greater impact on the overhead. Beside the direct relationship between the overhead and the parameters, we are interested in the grow ratios of the overhead with respect to each parameter. A session of experimentations provides the growth factor of the overhead with respect to parameters. Several sessions have been performed and the table below report their average values:

[4] `http://www.laas.fr/~trobert/RTRV`

ΔOh /ΔBr	$\Delta Oh/\Delta Tp_{max}$
17,12 ns/unit	1560,65

The results obtained through the proposed experimentation are pointing out that the most costly step is the state estimation, which is not surprising. Besides, the measured overheads related to model computations are of the same magnitude than the system calls used to enforce the detection policy. In addition to these quantitative results, this study pointed out the steps that actually require optimizations (step 2).

Nevertheless for models with non-trivial timing behaviours (TP_{max} between 3-10), the use of such monitor is not costly: the price to pay (40 microseconds) is only twice the worst case of a context switch of `Xenomai 2.3` (20 microseconds). This relationship is essential as deadline detection latency is directly connected to context switching latency. Thus for such models, the expected worst-case detection latency will be between 20 and 40 microseconds: 20 for deadline misses and 40 for unexpected events. The key parameter to estimate the cost of such a system is TP_{max}. When deploying the tool, one should focus on trying to improve the step 2 or on minimizing TP_{max}

5. Conclusion.

An experimental setting has been proposed to assess the overhead introduced by the real-time monitoring framework `RTRV`. Instead of selecting few "real-life" applications and testing the monitor on these applications, another assessment strategy has been followed. This work has been carried out according to several external requirements: the assessment methods should be simple but relevant. It leads us to consider first the theory to decide which parameters should be measured. Finally, we identified several steps in the monitoring algorithms for which duration measurements have been performed. The theory for some of these steps provided the dependencies between their durations and the parameters of the automaton. The results of the experimentations point out the impact on the overhead of the richness of the model with respect to event and/or timing constraints. As stated from the beginning this work is a fast and simple assessment of the overhead of a monitoring tool for elaborated specifications. Besides, like most of the monitoring frameworks for complex real-time specifications, run-time overhead highly depends on the specification. Thus a fair assessment strategy for various monitoring frameworks would require that the respective dependencies (model/overhead) are identified to avoid extreme bias. It is a prerequisite to propose a suitable measurement strategy with reduced cost.

The measurements showed that our monitoring framework is very efficient for non-trivial timing behaviours (i.e. TP_{max} between 3-10). As it has been pointed out in the current result, optimizing step 2 would benefit the monitor performances, since it represents a bottleneck for performance in our implementation. More elaborated data structures, such as binary search tree, would improve the performances at the expense of the storage cost. Independently, one could investigate approximation methods on models to reduce the value of TP_{max}. As in `Xenomai`, "pods" allow inserting applications written for other real time operating systems, (*RTOS*), such as `VxWorks`. Enabling such technologies for a real-time operating system like `Xenomai` seems to be a key feature to take advantage of the off-the-shelf components paradigms applied to real time softwares.

6. References.

[1] K. Havelund and G. Rosu: "Runtime Verification", vol. 70. Elsevier Science, 2002.

[2] A. Bauer, M. Leucker, and C. Schallhart : "Monitoring of real-time properties", in FSTTCS, (Kolkata, INDIA) , 2006.

[3] R. Alur and D. L. Dill: "A theory of timed automata", Theoretical Computer Science, vol. 126, 1994.

[4] S. Tripakis and S. Yovine: "Analysis of timed systems using timeabstracting bisimulations", Formal Methods in System Design, vol. 18, Springer, 2001.

[5] "Xenomai homepage," https://www.xenomai.org.

[6] N. Delgado, A. Q. Gates, and S. Roach: "A taxonomy and catalog of runtime software-fault monitoring tools," IEEE Transactions Software Engineering, vol. 30, IEEE Transactions, 2004.

[7] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky : "Formally Specified Monitoring Of Temporal Properties," in Proceedings of 11th Euromicro Conference on Real-Time Systems, 1999, pp. 114–22.

[8] K. Havelund and G. Rosu : "Monitoring Java programs with Java PathExplorer", Runtime Verification, (Paris, France), 2001.

[9] "Kronos home page": "http://www.verimag.imag.fr/TEMPORISE/kronos/.

[10] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson and Wang Yi: "Uppaal - a Tool Suite for Automatic Verification of Real-Time Systems", 4th DIMACS Workshop, (New Brunswick, USA), 1995.