Monitoring Arbitrary Activation Patterns in Real-Time Systems

Moritz Neukirchner, Tobias Michaels, Philip Axer, Sophie Quinton, Rolf Ernst
Institut für Datentechnik und Kommunikationsnetze
Technische Universität Braunschweig
Braunschweig, Germany
neukirchner|michaels|axer|quinton|ernst@ida.ing.tu-bs.de

Abstract—Model-based verification of timing properties has become industrial practice in design processes of safety-critical hard real-time systems. To validate the correctness of the used verification model, systems are additionally monitored during regular operation.

With a growing variety of activation patterns considered in verification, some of them with infinite range capturing arbitrary activation patterns, the known approaches to monitoring, which assume periodic streams, have become inapplicable or they suffer from large overhead due to piecewise continuous time monitoring.

In this paper we present a lightweight monitoring approach for arbitrary activation patterns. It profits from the discrete time property of a minimum distance event representation which is used instead of the continuous time representation used in earlier approaches. The method has a configurable constant runtime overhead in terms of memory and computation and allows conservative monitoring of a given arbitrary minimum distance function. Furthermore, we provide conditions under which the monitoring function is exact.

I. INTRODUCTION

Design processes for safety-critical systems often employ formal model-based verification algorithms to prove adherence to constraints, as e.g. timing deadlines. To ensure, that the models used during verification actually reflect actual system behavior, monitoring mechanisms, which check this consistency at runtime, are used.

This is especially important if safety- and time-critical applications are executed on the same platform as non-critical applications. Non-critical applications do not have the same requirements on certification and thus, their models may not be as accurate. As a consequence current safety standards (e.g. avionics DO-178B [1]) require “sufficient isolation” between critical and non-critical components. Such isolation w.r.t. timing can be achieved by strict timing segregation in scheduling (e.g. in the ARINC 653 standard) or by runtime monitoring and control, that stops or throttles tasks that exceed the expected resource usage. Both are established techniques. While strict segregation achieves a high degree of determinism, run-time monitoring and control allows for dynamically changing operation conditions, leading to higher overall performance [2]. However, runtime monitoring requires to check the adherence of execution behavior to a specified verification model with no false negatives.

Monitoring of timing properties, as regarded in this paper, involves checking runtime properties (e.g. task execution time) against verification models (e.g. worst-case execution time). This can be achieved e.g. by means of watchdog timers, timed monitoring tasks [3] or heartbeat monitors [4].

Complex activation patterns used in e.g. automotive systems [5], suggest use of more general event stream models than the classical PJd. Over the past years the capabilities of system-level timing verification have been extended tremendously into this direction, e.g. current commercial tools can consider arbitrary activation patterns, as specified through minimum distance functions [6]. However, monitoring capabilities lag behind. As many monitoring functions have to be executed frequently (e.g. during each context switch), these functions must be deterministic in their timing behavior (i.e. execution time independent of input data) and cause only minimal overhead. Furthermore, for implementation in software on COTS processors the use of hardware resources as e.g. timers has to be minimal. This makes monitoring according to highly expressive and complex verification models hard and it was even considered impossible [7].

In this paper we show that it is indeed possible to monitor task activations according to arbitrary activation patterns at minimal overhead. We assume that these activation patterns are modelled with minimum distance functions, which describe lower bounds on the temporal distance between consecutive activations. Generally, such specifications are infinitely long. In this paper, we provide methods to reduce minimum distance functions to relevant sections of limited length. We deduce how a system can be monitored efficiently and with constant overhead w.r.t. these bounded minimum distance functions. From this, we derive a method which yields a conservative monitoring function for a desired maximum monitoring overhead, which can be specified by a system designer. This allows to conservatively monitor arbitrary activation patterns with user-specifiable overhead. With the specification of the allowed overhead, the system designer may trade monitoring accuracy for memory and computation overhead.

The remainder of this paper is structured as follows. In sec. II we review existing work on monitoring of real-time properties. Then, we introduce the system model (sec. III) and the general monitoring setup (sec. IV). In sec. V we explain how a system can generally be monitored according to minimum distance functions. In sec. VI we show how monitoring overhead can be limited to constant values and in
sec. VII we apply the introduced methods to an example. In the following we provide details on an implementation of our monitoring scheme (sec. VIII) and provide an evaluation w.r.t. to memory and computation overhead (sec. IX). We finally conclude the paper in sec. X.

II. RELATED WORK

Monitoring real-time properties in safety-critical systems is common industrial practice, e.g. the automotive OSEK operating system standard provides for deadline monitoring facilities [8], [9] and the AUTOSAR standard [10] foresees measures to monitor execution time of software components and handle violations with a diagnostic event manager [11]. Common mechanisms to perform deadline and execution time monitoring are through watchdog timers - either in hardware or in software [4].

[4] identifies two main scenarios that may cause timing constraint violations at runtime which require to be monitored: hanging of a software component (e.g. due to excessive blocking or execution time) and excessive activation frequency (i.e. violation of the activation pattern specification). [4] proposes to monitor both scenarios with heartbeat monitors, which require the monitored component to periodically call a heartbeat function. Aliveness and activation frequency are monitored through the minimum and maximum number of heartbeats within a period. While this can be efficiently implemented it assumes a periodic timing model and thus is unsuitable to monitor arbitrary activation patterns. An extension of such monitoring through polling is proposed in the scope of sampling-based runtime verification [12], which however requires detailed code analysis to derive appropriate sampling periods.

A more expressive means for specifying and monitoring timing constraints is provided in [13], [14]. The high-level constraints to be monitored, however, require a clock synchronization between processors and the overhead of the monitoring scheme cannot clearly be bounded. Additionally, constraints are defined for pairs of events and require a watchdog timer each. This leads to high overhead when monitoring arbitrary minimum distance functions.

Another approach, which allows conservative monitoring of arbitrary activation patterns, is presented in [15], [16]. Both abstract activations through arrival curves, which are approximated through the minimum of several linear staircase segments (parametrized through period and jitter). Each such segment can be monitored with a leaky bucket mechanism. Due to the monitoring and its modelling according to arrival curves (Δt domain) this requires a timer per staircase segment. The overhead seems acceptable for an FPGA implementation, as proposed in [16]. For an implementation on a COTS processor with a limited number of timers and under consideration of timer interrupts, this is impractical - especially if several event streams modelled with several segments need to be monitored. Furthermore, the proposed modelling only allows to monitor the convex hull of arrival curves, which reduces accuracy for relevant traffic types, as e.g. periodic bursts.

An alternative to monitoring, which is however related, is the use of servers [7] (e.g. Constant-Bandwidth Servers [17], [18] or Latency-Rate Servers [19]). Here a task is assigned a certain execution or invocation budget. The execution/invocation of the task is monitored and this way the budget consumed. The budget is replenished according to the server strategy. This allows to control the amount of computation time a task receives. However, it usually requires a specific scheduling policy.

In this paper we present a monitoring scheme for arbitrary activation patterns, which runs at constant user-specifiable overhead, while only requiring one timebase for all monitored streams combined. It does not depend on any specific scheduling strategy and is suitable for implementation on COTS processors.

III. SYSTEM MODEL

In this section we introduce the system model used throughout this paper. We use the system model of [6], which is common in Compositional Performance Analysis. In this model the platform \( \mathcal{P} \) consists of multiple processors interconnected by communication media. Processors and communication media are referred to as resources \( r_j \in \mathcal{P} \). We make no assumption on the scheduling policy used on these resources.

On a platform a set of communicating tasks \( \mathcal{T} = \{ \tau_1, \ldots, \tau_n \} \) is executed. Communication going over communication media is modeled as a task on the respective communication resource. Tasks are interconnected by a set of directed event streams \( \mathcal{E} = \{ e_1, \ldots, e_n \} \) to denote their connectivity. When we talk about any event stream of \( \mathcal{E} \) we drop the subscript and denote it simply by \( e \). A task is activated by an event at an incoming event stream. We refer to an activation of a task as instance. Upon completion of an instance of a task an event is produced at all of the task’s outgoing event streams.

In order to have a notion on the temporal occurrence of events at an event stream we define an event trace, which describes a specific sequence of event occurrences.

**Definition 1** An event trace of an event stream \( e \) is an increasing function

\[
\sigma : \mathbb{N}^+ \rightarrow \mathbb{N}
\]

where \( \sigma(n) \) denotes the time of the \( n \)-th occurrence of an event in the trace of event stream \( e \).

Such a trace accurately describes a specific activation sequence of a task. An example trace is shown in figure 1a, where vertical arrows denote event occurrences.

In many systems task activations are not strictly deterministic. This may be because e.g. tasks may be activated through external events. Also in such systems bounds on the worst-case response times (WCRT) of tasks have to be guaranteed. To determine the WCRTs despite such uncertainties analytical analyses, such as e.g. Modular Performance Analysis (MPA)
[20] or Compositional Performance Analysis (CPA) [6], use worst-case abstractions of possible traces as input. These are often referred to as event models. Event models may be rather simple parametric descriptions as e.g. periodic with jitter [21]. A more expressive event model is given by the minimum distance function \( d \), which provides a lower bound on the minimum distance between a specified number of subsequent events. We will use the terms minimum distance function and \( d \) function interchangeably. We formally define it as:

**Definition 2** The minimum distance function \( d \) of an event stream \( e_i \) is a function

\[
d : \mathbb{N}^+ \rightarrow \mathbb{N}
\]

where \( d(n) \) describes a lower bound on the time interval between the occurrence of the first and the last event in any sequence of \( n + 1 \) consecutive events in the trace of event stream \( e_i \).

Note, that our definition of the minimum distance function \( d \), slightly differs from the minimum distance function \( \delta^- \) in e.g. [22]. Both are related through

\[
d(n) = \delta^-(n + 1)
\]

Minimum distance functions \( d \) allow us to specify arbitrary event models, including e.g. periodic, periodic with jitter or sporadic. An example minimum distance function is shown in figure 1b. It shows, that e.g. any three events are separated by a time-window of at least \( d(2) = \Delta t_3 \), i.e. the distance between the first and the third event, the distance between the second and fourth, between the third and fifth event and so on, may not exceed \( \Delta t_3 \). Conversely, speaking in terms of arrival curves [20], this can also be formulated as the requirement, that in any time-window of length \( \Delta t_3 \) no more than three events may occur.

Based on these event models, analyses, such as MPA [20] or CPA [6], guarantee that the worst-case response time of a task does not exceed a certain value, if the activation pattern of all tasks adhere to their respective event models. In this paper we provide methods to monitor task activations according to \( d \) functions and thus to enforce this adherence within the system. Note, that we describe monitoring only according to minimum distance functions. For many scheduling analyses maximum distance functions (which describe the maximum time between consecutive events) are not required. However, if needed, the methodology presented in this paper can be easily adapted to provide monitoring mechanisms for maximum distance functions. Monitoring of task execution times is out of the scope of this paper, but can be achieved through e.g. watchdog timers [4].

In the following section we describe the general monitoring setup, which we assume.

**IV. MONITORING SETUP**

In monitoring we aim to determine, whether the actual execution within a system adheres to the models and constraints, that were used or derived in design-time verification. In the context of real-time properties we have to determine, whether a specific event trace, which can be observed in the running system, adheres to the event model, that was used during response time analysis. Such monitoring schemes may be required because the verification models may be inaccurate (e.g. for low criticality tasks [2]) or because side-effects, such as e.g. interrupt latencies or blocking times, were not considered properly in the analysis.

We assume a monitoring setup as shown in figure 2. On a resource several tasks, which are activated by events at incoming event streams, are executing. Before a task is activated through an event, a monitor checks, whether the event occurrence adheres to a specified event model, which is given as minimum distance function. To check adherence each monitor has a trace buffer of limited length, where the timestamps of the last events are stored. In case the event does not fit the model, the monitor can take appropriate action, as e.g. delaying or blocking an activation (i.e. perform traffic shaping), to enforce adherence to the event model. Such a monitor must exist for every incoming event stream of all tasks on the processor.

Monitors can additionally be used at outgoing event streams (as shown in fig. 2) to control outgoing events. This can be used to e.g. control access to communication media. Monitors for outgoing events are optional and do not have to be used at all outgoing event streams.

**V. MONITORING WITH \( d \) FUNCTIONS**

Until now, we have described the general monitoring setup and our notion of traces and minimum distance functions. In order to monitor traces according to minimum distance functions, we have to relate both, i.e. we have to define when a trace satisfies a \( d \) function. In this section we introduce the notion of satisfaction of \( d \) functions, provide a notion of
limited history in traces and show how monitoring can be performed on the basis of such limited history.

A. Event Model Satisfaction

To reason about monitoring formally we first define a condition when a trace satisfies an event model specified through a minimum distance function.

Definition 3 An event trace $\sigma$ satisfies an event model given through $d$ if

$$\forall m, n \in \mathbb{N}^+, m > n : \sigma(m) - \sigma(m - n) \geq d(n)$$

(4)

This means, that an event trace satisfies an event model, when for all $m, n$ the distance between the $m$-th and $(m - n)$-th event is larger or equal to the value specified in the event model.

This allows us to determine whether a given trace $\sigma$ can be abstracted through a given minimum distance function $d$ and thus, whether the timing guarantees, derived through analysis using the given $d$ function, are valid.

B. Event Model Satisfaction on Limited History

The above definition of event model satisfaction requires an infinite trace. When monitoring a system, we can only check whether the past events at an event stream satisfy an event model. To reason about this we define a bounded event trace containing only a limited number of events.

Definition 4 A bounded event trace is a trace of limited length, containing $k \in \mathbb{N}^+$ events, i.e. it is a function

$$\sigma_k : \{1, \ldots, k\} \rightarrow \mathbb{N}$$

(5)

where $\sigma_k(n)$ denotes the time of the $n$-th occurrence of an event in the trace.

Analogous to definition 3 we define satisfaction of a $d$ function for bounded traces. This definition only considers the history of past events.

Definition 5 A bounded event trace $\sigma_k$ satisfies a $d$ function if

$$\forall m \in [2, k] : \forall n \in [1, m - 1] : \sigma_k(m) - \sigma_k(m - n) \geq d(n)$$

(6)

While the condition is identical to definition 3, the domain is limited to those events, that exist in the trace.

With definition 5 a runtime monitor can check at any time whether the activation history of the task, which is given through the event trace, satisfies a given $d$ function. Checking a trace of $k$ events for satisfaction of a $d$ function requires to check relation 6 pairwise between all events in the trace, i.e. $\sum_{m=2}^{k} (m - 1) = \frac{k(k-1)}{2}$ times. It is one of the main goals of this paper to check satisfaction of a $d$ function with overhead independent of the length of the trace, i.e. independent of $k$. This will be regarded in sec. VI. Before this, we focus on continuity of traces and show how the the fact that events are observed one after the other can be exploited to reduce the monitoring overhead.

C. Monitoring of Continuous Bounded Traces

Technically event traces are continuous processes, where the arrival of an additional event does not alter the history of previous events, i.e. the arrival of a new event cannot affect any previous event arrival. Hence, only newly recorded events have to be checked for satisfaction to the $d$ function. To reason about this we define the continuation of a trace.

Definition 6 A bounded event trace $\sigma_{k+m} : k, m \in \mathbb{N}^+$ is a continuation of a bounded event trace $\sigma_k$, if

$$\forall n \in [1, k] : \sigma_k(n) = \sigma_{k+m}(n)$$

(7)

Thus, recording additional events in a runtime monitoring scheme simply continues the existing trace. We now show, that we only have to check relation 6 for new events, if the trace satisfied the event model before.

Theorem 1 If a bounded event trace $\sigma_{k-1}$ satisfies a minimum distance function $d$, then each continuation $\sigma_k$ satisfying

$$\forall n \in [1, k - 1] : \sigma_k(k) - \sigma_k(k - n) \geq d(n)$$

(8)

also satisfies $d$.

Proof: From the definition of continuation of an event trace (eq. 7) and because $\sigma_{k-1}$ satisfies the minimum distance function $d$ we know that

$$\forall i \in [2, k - 1] : \forall n \in [1, i - 1] : \sigma_k(i) - \sigma_k(i - n) \geq d(n)$$

(9)

Thus, it remains to show, that relation 6 is also satisfied for $i = k$, $\forall n \in [1, k - 1]$, i.e. that

$$\forall n \in [1, k - 1] : \sigma_k(k) - \sigma_k(k - n) \geq d(n)$$

(10)

This is required by the theorem (relation 8).

As a consequence from theorem 1 we can perform monitoring of events by only require to checking new events against the existing trace. Additionally, this theorem implies, that we do not have to anticipate the arrival of future events, and only have to check against the past.

With theorem 1 we have decreased the complexity of monitoring as we only need to check relation 6 between the new event and all existing events, i.e. $k$ times for a trace of $k$ events.
Fig. 3: Example of a 4-repetitive \(d\) function

VI. LIMITING TRACE BUFFERS

Although we have reduced the complexity to a linear dependency of the trace length, this is still not suitable for runtime monitoring. First, as we are dealing with real-time systems, the monitoring requires a bounded execution time. Thus, its maximum execution time must not depend on the size of the trace. Secondly, the trace to check for satisfaction with the \(d\) function needs to be stored within the system. So far we cannot neglect any event from the trace and thus, the required memory would grow during runtime.

In this section we first show, that for a class of minimum distance functions, which we call \(l\)-repetitive, monitoring can be performed with trace buffers of fixed finite size. In sec. VI-B we show how we can construct more restrictive \(l\)-repetitive minimum distance functions from arbitrary minimum distance functions, i.e. allowing conservative monitoring of arbitrary \(d\) functions. In sec. VI-C we show how we can further reduce the trace buffer by considering properties of worst-case response time analysis.

A. Monitoring of Repetitive \(d\) functions

First, we show that monitoring with limited and constant trace buffers is possible for a class of \(d\) functions which we call \(l\)-repetitive. \(l\)-repetitive minimum distance functions are analogous to \(k\)-subadditive extension of arrival curves in [15].

**Definition 7** We call a minimum distance function \(d\) \(l\)-repetitive if it satisfies

\[
d(n) = \begin{cases} 
  d_n \text{ (given)} & \text{for } n \leq l \\
  \max_{w \in [l, l]} (d(w) + d(n-w)) & \text{for } n > l
\end{cases}
\]

Thus, an \(l\)-repetitive minimum distance function is only explicitly defined for arguments smaller or equal to \(l\). Each value for arguments greater than \(l\) is inductively expressed through a sum of this minimum distance function with arguments no greater than \(l\). An example of a 4-repetitive minimum distance function is given in figure 3, which represents a burst of 4 events that may occur at most every 10ms. Here every value of \(d(n)\) with \(n > l\) can be expressed through a combination of \(d(m)\) with \(m \leq l\), e.g. \(d(7) = d(3) + d(4)\).

For this class of \(d\) functions we now show, that the trace buffer can be limited to a length of \(l\) events.

**Theorem 2** Let the bounded event trace \(\sigma_{k-1}\) satisfy the \(l\)-repetitive minimum distance function \(d\). Then, each continuation \(\sigma_k\) of \(\sigma_{k-1}\) also satisfies \(d\) if

\[
\forall n \in [1, l] : \sigma_k(k) - \sigma_k(k-n) \geq d(n) \tag{12}
\]

**Proof:** As in the proof of theorem 1 we have to show, that

\[
\forall n \in [1, k-1] : \sigma_k(k) - \sigma_k(k-n) \geq d(n) \tag{13}
\]

From relation 12 we require this to be satisfied for all \(n \in [1, l]\). Thus it remains to show, that it is also true for all \(n \in [l+1, k-1]\), i.e. that

\[
\forall n \in [l+1, k-1] : \sigma_k(k) - \sigma_k(k-n) \geq d(n) \tag{14}
\]

In relation 13 we introduce event \(\sigma_k(k-b)\) for a given \(b \in [1, l]\) and reformulate the distance between the \(k\)-th and \((k-n)\)-th as

\[
\sigma_k(k) - \sigma_k(k-n) = \sigma_k(k) - \sigma_k(k-b) + \sigma_k(k-b) - \sigma_k(k-n) \tag{15}
\]

We will now find upper bounds for the distances between the \(k\)-th and \((k-b)\)-th event and the \((k-b)\)-th and \((k-n)\)-th event. We start with the first. From the initial assumption eq. 12 we know

\[
\forall b \in [1, l] : \sigma_k(k) - \sigma_k(k-b) \geq d(b) \tag{16}
\]

We now regard the distance between the \((k-b)\)-th and \((k-n)\)-th event. As \(b > 0\) and with the definition of continuation of an event trace (def. 6) we know

\[
\forall b \in [1, l] : \sigma_k(k-b) - \sigma_k(k-n) = \sigma_{k-1}(k-b) - \sigma_{k-1}(k-n) \tag{17}
\]

\(\sigma_k\) is a continuation of \(\sigma_{k-1}\), thus

\[
\forall n \in [1, k-1] : \sigma_{k-1}(n) = \sigma_k(n) \tag{18}
\]

As \(\sigma_{k-1}\) satisfies \(d\) by requirement of the theorem and because \(b \in [1, l]\) and \(n \in [l+1, k-1]\)

\[
\forall b \in [1, l] : \sigma_k(k-b) - \sigma_k(k-n) \geq d(n-b) \tag{19}
\]

With equations 15, 16 and 19.

\[
\forall n \in [l+1, k-1], \forall b \in [1, l] : \sigma_k(k) - \sigma_k(k-n) \geq d(b) + d(n-b) \tag{20}
\]

Since \(\sigma_k(k) - \sigma_k(k-n)\) is independent of \(b\) and with the definition of \(l\)-repetitive minimum distance functions (eq. 11)

\[
\forall n \in [l+1, k-1] : \sigma_k(k) - \sigma_k(k-n) \geq \max_{b \in [1, l]} (d(b) + d(n-b)) = d(n) \tag{21}
\]

Theorem 2 states, that we only have to check a new event against the last \(l\) events, if the minimum distance function to monitor is \(l\)-repetitive. Thus, if the monitor shall e.g. allow a burst of 4 events every 10ms (4-repetitive), the trace buffer only has to contain the timestamps of the last 4 activations.
and any new event only has to be checked against the last 4 activations. Thus, for \( l \)-repetitive minimum distance functions, the monitoring overhead (in memory and execution time) is constant.

\( l \)-repetitive minimum distance functions are highly relevant in practice, as they describe e.g. periodic activations or periodic bursts, i.e. they allow monitoring of a wide range of realistic event models.

B. Limiting to More Restrictive Event Model

In the previous section we have shown, how the monitoring overhead can be bounded in case of \( l \)-repetitive minimum distance functions. Although many common parametrizations of event models are \( l \)-repetitive, in general \( d \) functions are not. Additionally, the monitoring overhead for \( l \)-repetitive \( d \) functions is directly given through \( l \), as \( l \) comparisons and a trace buffer of \( l \) events is required. Thus, large \( l \), lead to large monitoring overhead.

In this section we introduce a methodology that allows us to construct from an arbitrary event model another event model, that is more restrictive and \( l \)-repetitive. The method is parametrizable in the size of the desired trace buffer and thus allows us to define the monitoring overhead as design parameter.

We start with an initial assumption on the arbitrary minimum distance function to be monitored.

Assumption 1 For verification of timing constraints only a part of the \( d \) function is relevant. The relevant domain is \([1,n_{\text{max}}]\) where \( n_{\text{max}} \) depends on scheduling policy of the resource.

This assumption states, that the distance between two events, that are sufficiently far apart, does not influence the scheduling on a resource. This is a reasonable assumption for realistic scheduling policies. In sec. VI-C we show in particular, that this is assumption applies to all schedulers that can be analyzed with the busy-window approach [23], [24], i.e. work-conserving schedulers, such as e.g. static priority preemptive (SPP), rate monotonic scheduling (RMS) or earliest deadline first (EDF).

Before we construct a more restrictive \( l \)-repetitive event model, we formally define what it means for a \( d \) function to be more restrictive.

Definition 8 A minimum distance function \( d \) is more restrictive than another minimum distance function \( d' \) if
\[
\forall n \in \mathbb{N}^+ : d'(n) \geq d(n) \tag{22}
\]

Additionally, we call a function more restrictive in an interval \([1,n]\) if relation 22 is only satisfied in this interval. Naturally, if a trace satisfies a more restrictive minimum distance function \( d' \) it also satisfies the less restrictive minimum distance function \( d \). We show this in the following lemma.

Lemma 1 Let \( d' \) be more restrictive than \( d \) and let the bounded event trace \( \sigma_k \) satisfy \( d' \). Then \( \sigma_k \) also satisfies \( d \).

Proof: With the definition of satisfaction of an event model (def. 5) and the definition of more restrictive \( d' \) functions (def. 8), the lemma immediately follows.

\[
\forall m \in [2,k] : \forall n \in [1,m-1] : \\
\sigma_k(m) - \sigma_k(m - n) \geq d'(n) \geq d(n) \tag{23}
\]

Thus, any trace that satisfies a more restrictive event model also satisfies the original model. As a consequence we can monitor according to any more restrictive minimum distance function without risking violation of the original event model. The inverse naturally is not true. Thus, when monitoring according to a more restrictive minimum distance function instead of the specified one, the monitor may yield false positives, i.e. notify of an event model violation, although the trace adheres to the original \( d \) function. For example, the original \( d \) function might allow a burst of three events and the more restrictive function only a burst of two events. Consequently, conservatism is introduced. We address this conservatism later.

For now, we regard the possible design options resulting from lemma 1 and from theorem 2. We know, that we can safely monitor according to any more restrictive minimum distance function (lemma 1), and that it is sufficient to monitor the last \( l \) events for \( l \)-repetitive \( d \) functions (theorem 2). Thus, for a given \( d \) if we can construct an \( l \)-repetitive minimum distance function more restrictive than \( d \) for a user-specified \( l \), we can explicitly define the monitoring overhead as design parameter.

We now define such a construction as an optimization problem and derive properties on the solution quality. A minimum distance function \( d \) is \( l \)-repetitive and more restrictive than \( d \) in the interval \([1,n_{\text{max}}]\) if it satisfies the following constraints
\[
\forall n \in [1,n_{\text{max}}] : d'(n) \geq d(n) \tag{24}
\]
\[
\forall n \in [l+1,n_{\text{max}}] : d'(n) = \max_{w \in [1,l]} (d'(w) + d'(n-w)) \tag{25}
\]

Naturally, a construction of an \( l \)-repetitive more restrictive minimum distance function \( d'(n) \) should introduce minimal conservatism. To quantitatively express the overestimation we define a metric for this conservatism, which we will use for the objective function of the optimization problem. The weighted accumulated distance \( \Delta \) between a conservative monitoring function \( d' \) and the original function \( d \) within the relevant interval \([1,n_{\text{max}}]\) is
\[
\Delta(d'(n)) = \sum_{n \in [1,n_{\text{max}}]} \alpha_n * (d'(n) - d(n)) \tag{26}
\]
where \( \alpha_n \) are predefined weights. Recall, that \( d(n) \) denotes the minimum distance between any \( n+1 \) consecutive events (def. 2). Thus, the difference \( d(n) - d(n) \) specifies, how much more restrictive \( d'(n) \) is for the distance between \( n+1 \) events.
Let \( d(n) \) be a given function and \( \Delta(n) \) an optimal w.r.t. to a quality metric \( \Delta \) and a given function \( d(n) \) that is relevant in the interval \([1, n_{\text{Max}}]\), if it is the solution to the minimization problem

\[
\min(\Delta(d^\Delta(n)))
\]

subject to

\[
\forall n \in [1, n_{\text{Max}}] : d^\Delta(n) \geq d(n)
\]

\[
\forall n \in [l + 1, n_{\text{Max}}] : d^\Delta(n) = \max_{w \in [1,l]} (d^\Delta(w) + d^\Delta(l + 1 - w))
\]

Thus, an optimal \( l \)-repetitive more restrictive minimum distance function has minimal conservatism for a given \( l \). Now, we show, that the optimal solution can only yield less conservative results, if the trace buffer is increased, i.e. by increasing the allowed monitoring overhead, we can only increase the accuracy of the monitoring.

**Lemma 2** Let \( d^\Delta \) and \( d^{\Delta+1} \) be optimal w.r.t. to a quality metric \( \Delta \) and a given function \( d \). Then

\[
\Delta(d^\Delta(n)) \geq \Delta(d^{\Delta+1}(n))
\]

**Proof:** The proof is through the definition of the optimization problem in def. 9. The functions \( d^\Delta(n) \) and \( d^{\Delta+1}(n) \) only differ in the definition through the constraint

\[
d^\Delta(l + 1) = \max_{w \in [1,l]} (d^\Delta(w) + d^\Delta(l + 1 - w))
\]

which appears in the definition of \( d^\Delta(n) \) but not for \( d^{\Delta+1}(n) \). Thus, the definition of \( d^{\Delta+1}(n) \) is a relaxed optimization problem of the definition of \( d^\Delta(n) \). As a consequence, the solution can only be better w.r.t. to the quality metric. □

As a result from lemma 2 we can conclude, that the conservatism of an optimal construction of an \( l \)-repetitive more restrictive minimum distance function can only be decreased if the allowed trace buffer size \( l \) is increased. As a result, a system designer can perform a trade-off between monitoring accuracy and overhead.

Although the construction of an optimal monitoring function can be performed at design time, the computational complexity may be high. Thus, we now provide an efficient heuristic to construct a more restrictive \( l \)-repetitive minimum distance function from a given \( d \). We illustrate the approach graphically and then derive a formal description of the method. Figure 4 shows in black dots a non-repetitive minimum distance function that is relevant up to \( n_{\text{Max}} = 7 \). For \( d \) we wish to define a \( 4 \)-repetitive more restrictive minimum distance function \( d^4 \). As \( l \)-repetitive functions are implicitly defined for \( n > l \) (see def. 7), we only have to define \( d^4 \) with \( i \in [1,4] \).

We start with the informal graphical description of the heuristic construction. We first find the largest slope from the origin to any point of \( d \). We denote this slope \( a \) (see fig. 4). By definition of \( a \) any point of \( d(n) \) for \( n \leq n_{\text{Max}} \) is below this tangent. Thus, we can initialize all \( d^4_1 = j \cdot a \) and the resulting \( d^4(n) \) will be larger than \( d(n) \) in the relevant section \([1, n_{\text{Max}}]\). Next, we subsequently decrease the individual values \( d^4_1 \), starting, with \( d^4_1 \), such that \( d^4(n) \geq d(n) \) still holds. This is illustrated in fig. 4.

Now, we formulate this construction formally. The maximum slope from the origin to any point of \( d \) in the relevant interval is given through

\[
a = \max_{j \in [1,n_{\text{Max}}]} \left( \frac{d(j)}{j} \right)
\]

With \( a \) the construction is defined as:

**Definition 10** Let \( d \) be a minimum distance function which is relevant in the interval \([1, n_{\text{Max}}]\) and let \( a \) be the maximum slope of \( d \). The parameterized \( l \)-repetitive minimum distance function \( d^l \) of \( d \) is defined as

\[
\tilde{d}^l(n) = \begin{cases} 
\bar{d}_n & \text{for } n \leq l \\
\max_{w \in [1,l]} (d(w) + d(n - w)) & \text{for } n > l
\end{cases}
\]

Where \( \bar{d}_n = \tilde{d}^{(l)}_n \) is calculated iteratively (in \( l \) iterations) with

\[
\begin{cases} 
\tilde{d}^{(l)}_j = \min_{d_i \in [d(i), a \cdot i]} (\tilde{d}^{(l-1)}_i : \forall n \leq n_{\text{Max}}, \tilde{d}^l(n) \geq d(n)) : i = j \\
\tilde{d}^{(l)}_j = \tilde{d}^{(l-1)}_j & : i \neq j
\end{cases}
\]

Note, that this definition of the parameterized \( l \)-repetitive minimum distance function \( d^l \) is only given explicitly in the domain \([1, l]\). For any larger argument it is specified through \( l \)-repetitiveness (eq. 11). In the section \([1, l]\) \( d^l \) is given through
the iterative formula eq. 34. Here each value \( d_j \) is initialized with the corresponding value from the maximum tangent \( a \ast i \). Then, starting from \( d_1 \), each value is reduced such that \( d^j(n) \geq d(n) \) still holds, assuming the current assignment of all other \( d_i \).

Now, we show that the resulting function \( d^j(n) \) is indeed more restrictive than \( d \).

**Lemma 3** Any parametrized \( l \)-repetitive minimum distance function \( d^j \) is more restrictive than its corresponding \( d \) function.

**Proof:** By definition (eq. 34) all \( d^j \) \((n \leq l)\) are initialized with \( a \ast n \) and the values are decreased only such that \( d^j(n) \geq d(n) \) still holds. Consequently, we only have to show that \( d^j(n) \geq d(n) \) already holds for the initial assignment, i.e. that

\[
\forall n \in [1, n_{max}]: a \ast n \geq d(n) \tag{35}
\]

Trivially, we know

\[
\forall n \in [1, n_{max}]: \max_{j \in [1, n_{max}]} \left( \frac{d(j)}{j} \right) \ast n \geq \frac{d(n)}{n} \ast n \tag{36}
\]

Thus, with definition of \( a \) (eq. 32)

\[
\forall n \in [1, n_{max}]: a \ast n \geq d(n) \tag{37}
\]

Thus, lemma 3 shows that \( d^j \) is by construction more restrictive than \( d \). Thus, it can safely be used for monitoring. Its conservatism can be explicitly calculated with \( \Delta \).

**C. Determination of Relevant Section of \( d \) Function**

In the previous two sections we have shown how we can achieve constant monitoring overhead for \( l \)-repetitive minimum distance functions and how we can construct an \( l \)-repetitive minimum distance function from an arbitrary \( d \) function. As pointed out above, this construction requires to specify which section of the original \( d \) function is relevant. In this section we further elaborate what our notion of relevant is and how it can be determined for any scheduler that can be analyzed with the busy-window analysis.

As described in sec. IV the goal of monitoring is to ensure, that the experienced behavior of the system does not exceed its verification model. For timing, monitoring shall ensure that tasks do not experience any larger worst-case response times than verified and that their output event traces still satisfy the output event model. Thus, in contrast to previous sections, where we regarded solely isolated event streams and their event model, now we have to consider the scheduling of the tasks that are activated by events at an event stream.

**Assumption 2** We assume that the execution sequence of tasks on a resource only depends on the number of pending activations of all tasks and the state of the scheduler (if it is stateful).

For stateless scheduling policies, such as e.g. SPP, RMS or EDF, this assumption states, if the resource reaches its initial state (no activations pending), the scheduling does not depend on the execution history. In terms of traces this means, that we can neglect all trace data (i.e. flush trace buffers), whenever the resource becomes idle. Consequently, if we can bound the maximum time until a resource with a stateless scheduler becomes idle, we can determine the maximum number of events in this time window. This maximum number of events, that have to be considered, directly specify the relevant section of a \( d \) function.

Any work-conserving scheduling policy, i.e. every scheduling policy that does not leave the resource idle if any task activations are pending, is analyzable with the busy-window approach [23], [24]. The busy-window approach calculates the maximum time-window \( w_i(q) \) during which the resource is busy processing until completion of the \( q \)-th instance of task \( \tau_i \). Thus, the longest time \( w \) a resource can process task activations without becoming idle is given through the maximum over all tasks and their activations, i.e.

\[
w = \max_{i,q} (w_i(q)) \tag{38}
\]

Thus, after at most \( w \) time units, the resource becomes idle and we can neglect all history, i.e. trace data. From this maximum busy-window \( w \) we can now determine the relevant section \([1, n_{max}]\) of the minimum distance function \( d \) describing the activation of task \( \tau_i \).

\[
n_{max} = \max_{n \in \mathbb{N}^+} (n : d(n) \leq w) \tag{39}
\]

This function yields the maximum value \( n \) for which \( d(n) \) is still smaller than the worst-case busy-window \( w \). We do not have to check any values \( d(n) \) for \( n > n_{max} \) as they will not influence scheduling on this resource. Thus, the trace buffer of an event stream requires to store no more than \( n_{max} \) events. If the required trace buffer shall be reduced through use of a more restrictive monitoring function as shown in the sec. VI-B, \( n_{max} \) can be used in the construction of the monitoring function.

**VII. CASE-STUDY**

In this section we illustrate on an example, how the methods presented in sec. VI can be used to design an activation pattern monitor for an arbitrary event model.

Assume we want to monitor the activation pattern of a task on a work-conserving resource. The activation pattern is modeled through a given minimum distance function \( d \), which is shown up to \( n = 8 \) in fig. 5a. The given \( d \) is not \( l \)-repetitive and thus cannot be monitored with constant overhead.

To limit the overhead we can first perform a busy-window analysis of the resource. Assume that the maximum number of task activations within the maximum busy-window is 7, i.e. \( n_{max} = 6 \). Consequently, from sec. VI-C we know, that a trace buffer needs to contain at most \( n_{max} = 6 \) events and from theorem 1 we only need to perform \( n_{max} = 6 \) comparisons. However, we may decide that the overhead of a trace buffer containing a trace of 6 events is too large. Thus, we aim to construct a parametrized \( l \)-repetitive minimum
distance function with the methodology provided in sec. VI-B. Start with a value of \( l = 3 \). The values of \( d^3(n) \) in the relevant interval are shown in fig. 5b and table I. In order to evaluate the conservatism of \( d^3(n) \) we calculate \( \Delta(d^3) \) (with all weights equal to 1) which yields a value of 3. Thus, monitoring according to \( d^3(n) \) may yield false positives. To reduce conservatism we evaluate the conservatism that would be introduced when monitoring according to \( d^4(n) \). Again, the values are summarized in tab. I. We see, that \( d^4(n) \) is equal to \( d \) in the relevant section, i.e. that \( \Delta(d^4) = 0 \). Thus, we can safely monitor the corresponding event trace according to \( d^4 \) without introducing conservatism. The required trace buffer size holds for \( 4 \) events and a total number of \( 4 \) comparisons is required upon arrival of a new event. As \( \Delta(d^4) = 0 \) we know, that increasing the trace buffer size further, cannot provide additional accuracy.

VIII. IMPLEMENTATION

In the previous sections, we have shown that monitoring according to arbitrary event models is possible with constant overhead and how the desired overhead can be explicitly defined and reduced by the system designer at the cost of higher conservatism. In this section we focus on the implementation of a monitoring scheme as outlined in this paper.

Algorithm 1 shows in pseudocode the function that has to be executed upon arrival of each new event. It takes as input the current time, i.e. the timestamp of the occurrence of the new event, and a trace buffer of size \( l \). Here, the value in \( \text{trace buffer}[0] \) denotes the timestamp of the most recent and \( \text{trace buffer}[l-1] \) that of the least recent event occurrence. We assume, that the timestamp is either provided by the operating system or a dedicated hardware timer. Upon bootup, the trace buffer is initialized with values of \(-\infty\). Additionally, the algorithm requires the \( d \) function to be given in an array. Here \( d[i] \) denotes \( d(i+1) \).

Upon arrival of a new event the algorithm checks for all \( l \) elements in the trace buffer (line 1) whether the distance between the new event and the event in the trace complies with the value specified in the \( d \) function (line 2). If this is not the case the algorithm triggers a notification, that the monitoring bound has been violated (line 3). At the end of all checks, the trace buffer is shifted, i.e. the least recent event is discarded, and the current timestamp is saved in \( \text{trace buffer}[0] \) to denote the most recent event occurrence (lines 6-7).

The runtime of the entire algorithm only depends on the size of the trace buffer, which is statically configured at design time. Thus, the monitoring is performed at constant runtimes.

IX. EVALUATION

In this section we evaluate the feasibility of the proposed monitoring method. We have implemented this monitoring scheme on top of the real-time kernel MicroC/OS-II [25] on a STM3220G-EVAL board featuring a Cortex-M3 microcontroller running at 120MHz. We have analyzed the induced overhead in terms of memory consumption and runtime overhead.

The entire implementation requires only two functions. The first function initializes and configures the monitor. This includes specification of the \( d \) function to be monitored, registration of an exception handler to be called in case of violations and initialization of the trace buffer. The second function implements the actual monitoring functionality and checks whether a new event may occur at the given time (alg. 1). In our implementation we use the operating system time ticks as time base. Overall the implementation requires 508 bytes of program code.

The required data depends on the size of the trace buffer. Each element in the buffer requires 4 bytes. Additionally, every value in the \( d \) function also requires 4 bytes. Trace buffer and \( d \) function both contain \( l \) elements. The monitor itself requires 20 bytes of data. Thus, the total data memory overhead of a monitor is given through

\[
\text{Mem}_{\text{data}} = 20 + l \times 8 \tag{40}
\]

In order to evaluate the runtime overhead, we have measured the latency from calling the monitor until the first instruction of the exception handler, i.e. the detection latency of the monitor. For this purpose we have controlled the activation pattern such that always the last comparison in the monitor leads to a violation of the \( d \) function, i.e. that the maximum detection latency was observed. We have performed this experiment for...
a range of different trace buffer sizes. For each buffer size
200 samples were obtained. The results are shown in fig. 6.
The diagram shows, the average detection latency in number of
clock cycles over the trace buffer size in number of events.
We see that the monitoring runtime overhead is linear in terms
of the size of the used trace buffer and is in the order of
magnitude of ∼10 clock cycles per buffer entry. The standard
deviation among the samples for each trace buffer size always
was below 2.3%. This deviation and the slight non-linearity
in the graph are due to caching effects on the processor.
From this evaluation we see, that the proposed monitoring
according to arbitrary activation patterns is feasible and can be
performed at minimal overhead. The overhead scales linearly
w.r.t. the size of the used trace buffer and is constant for a
given buffer size.

X. CONCLUSION
In this paper we have presented a novel method to monitor
arbitrary activation patterns in real-time systems. We have
provided a definition of event model satisfaction and shown,
that monitoring can be performed continuously, only requiring
to check satisfaction of the event model for the most recent
event.
Based on this continuous monitoring scheme we have shown
that the monitoring overhead can be bounded by a constant
for the broad class of 1-repetitive event models, which are
highly relevant in practice. Furthermore, we have provided a
methodology, by which a designer can explicitly specify the
tolerable monitoring overhead and based on this parametriza-
tion can construct a conservative activation pattern monitor
based on an arbitrary activation pattern. The overestimation of
the constructed monitor can explicitly be calculated. Finally,
we have shown that the overhead can be further reduced by
considering scheduling analysis in the design of the monitoring
scheme.
We have shown the feasibility of the proposed monitoring
scheme in an implementation and evaluated the memory and runtime overhead.

REFERENCES
[1] “DO-178B - software considerations in airborne systems and equipment
watchdog as a dependability software service for automotive safety
relevant systems,” in Intl. Conf. on Dependable Systems and Networks
(DSN), 2007.
und gateway-systemen im kraftfahrzeug,” Ph.D. dissertation, Karl-
sruher Institut für Technologie (KIT), 2010.
“System level performance analysis - the syntax/s approach,” Computers
development,” in Design, Automation and Test in Europe Conference
[10] AUTOSAR - Automotive Open System Architecture,
http://www.autosar.org/.
O. Weis, T. Weber, J. Ruh, L. Lundh, T. Sandén, P. Heitkämper,
R. Rimkus, J. Leflour, A. Gilber, U. Virmich, S. Voget, K. Nishikawa,
K. Kajio, T. Schurnhorst, and B. Kunkel, “Autosar – current results and
preparations for exploitation,” in 7th EUROFORUM conference Software
timing constraints in distributed real-time systems,” Real-Time Syst.,
with confidence threshold requirements,” IEEE Transactions on
networks with a deterministic service,” in Proc. of Annual Joint Con-
ference of the IEEE Computer Societies, INFOCOM, 1996.
inputs for hard real-time embedded systems,” in Proc. of the 49th Design
Automation Conference (DAC), 2012.
real-time systems,” in Real-Time Systems Symposium, 1998. Proceed-
server,” in Proc. of Real-Time Systems Symposium. IEEE Computer
for analysis of traffic scheduling algorithms,” IEEE/ACM Trans. on
scheduling hard real-time systems,” in Symp. on Circuits and Systems
(ISCAS), 2000.
models,” Ph.D. dissertation, Technical University of Braunschweig, De-
[22] M. Jersak, “Compositional scheduling analysis using standard event
real-time systems,” Journal of Real-Time Systems, vol. 6, pp. 133–152,
1994.
network (can) schedulability analysis: Refuted, revisited and revised,”

Fig. 6: Detection Latency