CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

Conference Paper

REVERT: Runtime Verification for Real-Time Systems

Sangeeth Kochanthara
Geoffrey Nelissen
David Pereira
Rahul Purandare

CISTER-TR-161006

REVERT: Runtime Verification for Real-Time Systems

Sangeeth Kochanthara, Geoffrey Nelissen, David Pereira, Rahul Purandare

*CISTER Research Centre

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail:

http://www.cister.isep.ipp.pt

Abstract

Real-time systems are becoming more complex andopen, thus increasing their development and verification costs.Although several static verification tools have been proposedover the last decades, they suffer from scalability and precisionproblems. As a result, the tools fail to cover all the necessarysafety properties for realistic real-time applications involving alarge number of components and tasks. Runtime verification is aformal technique that verifies properties during system executionwith the support of monitors. The monitors are generatedfrom formal languages using correct-by-construction generationmethods. Runtime verification can thus be used as a complementor replacement for static verification approaches. The currentstate-of-the-art tools either do not have notion of time, or sufferfrom the potential blowup of states at run-time. In this paper,we propose REVERT, a framework developed with a focus onthe verification of functional and non-functional properties withtiming constraints. The contribution of this work is twofold: (i) adomain-specific specification language allowing the definition ofrequirements for real-time applications; (ii) a novel mechanism togenerate monitors, with state-space and time guarantees, capableof identifying and reacting to timing properties defined with theproposed specification language.

REVERT: Runtime Verification for Real-Time Systems

Sangeeth Kochanthara
IIIT-Delhi,
India

Geoffrey Nelissen
CISTER/INESC TEC, ISEP,
Portugal

David Pereira
CISTER/INESC TEC, ISEP,
Portugal

Rahul Purandare
IIIT-Delhi,
India

***Abstract*—Real-time systems are becoming more complex and open, thus increasing their development and verification costs. Although several static verification tools have been proposed over the last decades, they suffer from scalability and precision problems. As a result, the tools fail to cover all the necessary safety properties for realistic real-time applications involving a large number of components and tasks. Runtime verification is a formal technique that verifies properties during system execution with the support of monitors. The monitors are generated from formal languages using correct-by-construction generation methods. Runtime verification can thus be used as a complement or replacement for static verification approaches. The current state-of-the-art tools either do not have notion of time, or suffer from the potential blowup of states at run-time. In this paper, we propose REVERT, a framework developed with a focus on the verification of functional and non-functional properties with timing constraints. The contribution of this work is twofold: (i) a domain-specific specification language allowing the definition of requirements for real-time applications; (ii) a novel mechanism to generate monitors, with state-space and time guarantees, capable of identifying and reacting to timing properties defined with the proposed specification language.**

I. INTRODUCTION

The time at which a result is produced by Real-Time Systems (RTS) is as important as their functional correctness. Depending on the application domain and on the level of criticality associated with the application, failing to meet some timing constraints can lead to drastic consequences for the system's environment and the agents involved in the system's operation. Due to the strong reliability and predictability demanded from these systems, verification and validation are fundamental activities often required to be performed according to directives of legal certification entities [7].

Static verification is one of the means to address the strong reliability and predictability demands. However, static verification experiences practical limitations such as undecidability of properties of the underlying formal model, or blowup of the potential states to track. Moreover, extra-functional properties like the time at which events occur are only available at run-time. This scenario makes Runtime Verification (RV) techniques the natural candidates to address the current limitations of static approaches [9].

The earliest works in the RV literature focused on *event-triggered* monitoring in which monitors are invoked on each event occurrence that is being monitored. RMOR [5] and MOP [4], for instance, use aspect-oriented programming to instrument the target application's source code. Such methods, however, have unpredictable overheads [10] making them unsuitable to RTS. Moreover, aspect-oriented programming may impact the timing and correctness of the target system and may interfere with certifiability constraints.

In order to make RV suitable to RTS, Zhu et al. [12] proposed predictable monitoring which ensures temporal non-interference of the system being monitored. More recently, Navabpour et al. introduced Rithm [10] for RV on many-core platforms using LTL 3-valued logic to specify properties. Rithm is based on a time-triggered framework. Rithm can use a GPU to improve the responsiveness of the monitors by parallel execution of monitors on accumulated traces. However, it may face a trade-off between responsiveness and efficiency since execution of parallel monitor for each event occurrence in real-time will reduce efficiency. Furthermore, there is a significant overhead incurred while transferring data between the host monitoring process on the CPU and the monitoring threads on the GPU. In comparison, self-monitoring [2], where monitoring code is directly inserted in the application code, has a better response time, but with the potential drawback of hampering the timing properties of the program being monitored, as well as linking the behavior of the monitors to the behavior of the monitored tasks. The main limitation of Rithm though is its lack of notion of time. It relies on the relative ordering of events but cannot specify timing constraints on a sequence of events.

RuleR [6], RT-Mac [11], and Copilot [8] are examples of tools with notion of time. RuleR has a highly expressive monitoring architecture which models constraints as rules. Yet, RTS properties may be difficult to model as rules which makes RuleR hard to comprehend, error-prone, and better suited for domain experts rather than for industrial developers. Notably, Copilot is one of the RV tools designed to handle ultra critical systems and uses Satisfiability Modulo Theories (SMT) based k-induction [8] to prove invariant properties of generated monitors. Due to the non-deterministic properties of timed automata models, the tools with notion of time may have to keep track of multiple possible states at each time instant. It was shown that, under such models, the number of states that need to be tracked by the monitor may grow exponentially [1]. Therefore the memory space and the computing time required by those tools are hard to predict.

Contributions. In this paper, we propose a new specification

language named REVERT. It supports timing constraints on top of events relative ordering. It is designed to be simple and easy to learn. As a second contribution, we also present a new method to generate complete finite deterministic timed automata from the specification written with REVERT.

We argue that this new method guarantees that the generated monitor has to keep track of only one state at any point in run-time, thus avoiding the potential blowup in the number of states of the generated monitor that most RV frameworks encounter. This makes REVERT an efficient and expressive inline runtime verification framework for safety critical systems.

II. SPECIFICATION LANGUAGE

REVERT is a new specification language for real-time applications designed with usability and easiness in mind. REVERT is a combination of state machine, extended regular expressions, boolean expressions, and timing constraints.

REVERT relies on external events to reason about traces. Properties on execution patterns or execution order of events, that must be enforced during the application run-time, are specified using extended regular expressions. To express timing constraints on a sequence of events we use three high-level operators: `time`, `duration` and `jitter` (refer to section III for a formal definition). These operators are then automatically converted to finite timed automata. The syntactic structure of a monitor specification in REVERT is presented below:

```
monitor m_i {
  observe { ev_1, ..., ev_l }
  variables {v_1 : type, ..., v_j : type }
  jobs{
    j_1{
     start: {ev_1, ev_2}
     suspend: {ev_3}
     resume: {ev_1}
     complete: {ev_6}
    },
    ...,
    j_p{...}
  }

  nodes { n_1, ..., n_k }
  initial { n_q }

  node n_1 { init_1 prop_1 trans_1 }
  ...
  node n_k { init_k prop_k trans_k }
}
```

Listing 1: Structure of a monitor specification in REVERT

The `observe` statement specifies the events that are monitored by the monitor m_i, out of the complete set of events produced by the monitored application. The complete set of events is specified in external files included in the specification.

The `variables` statement defines variables local to the monitor m_i; The `jobs` statement declares the set of jobs associated with different tasks. Each job specification is defined by the set of events associated with its lifecycle (from its release to its completion). Each job is defined using the following four sets of events; `start`, `suspend`, `resume` and `complete`, which contain events related to the release, suspension (for instance, due to preemption, unavailability of a shared resource or a self-suspension), resumption, and completion of the job, respectively.

The `nodes` statement declares identifiers of all the nodes of the monitor. Those nodes model the different states that can be reached by a finite state machine, which determine how the properties to be monitored evolve with the system state. The `initial` statement declares the node in which the monitor will be active when it starts its execution. The behavior of the monitor for each node n_i is specified in a `node` block with the corresponding name. A node block n_i includes some initialization code $init_i$, the set of properties that need to be monitored $prop_i$, and the set of transitions $trans_i$ from the current node n_i to any other node defined in the monitor. The transitions between the nodes are guarded by guards based on the success or failure of the properties monitored in that node.

The structure of the specification language is built on top of two main observations. First, real-time systems may be dynamic, adapting to the changes in the environment, their workload, and the type of operations that must be performed at a given time or reacting to detected anomalies. Consequently, the properties that must be verified by the monitors may change over time, and it should be possible to specify different modes of operations that are activated depending on some constraints. In the monitor specification as presented in Listing 1, different nodes can be seen as different modes of execution. Transitions can be used to specify mode changes or the activation of corrective measures in case of a specification violation. The definition of a corrective action and the mechanism for its execution are not in the scope of this paper.

As a second observation, we realized that the number of timing properties that must be verifiable are rather limited and can all be expressed with a combination of the three operators `time`, `duration` and `jitter`, which return the time taken by a sequence of events, the execution time of a job, and the jitter on a timing property, respectively. The `time` operator may, for instance, be used to verify that a deadline, a period or a minimum separation time between two events is respected. The `duration` operator is useful to check that the execution time of a job does not exceed its budget or estimated worst-case execution time, or to ensure that the interference suffered by one task due to other tasks is bounded. Finally, the `jitter` operator may be used to bound the variation on any timing property. It can be argued that similar properties can be encoded in existing frameworks such as RULER and RT-MaC. However, they are not all intrinsic constructs of the language, which renders their specification difficult and error-prone to inexperienced users.

The specification language does not explicitly impose but expects the guards on the transitions to be mutually exclusive. If some non-determinism exists in the specification due to non-mutually exclusive node transitions, it is resolved during the monitor generation using implicit priorities. Transitions are prioritized in the order of their declarations, thereby ensuring that there is only one active node at any time.

III. MODEL

Let Σ be the set of all observable events in the application. The monitoring model considers a finite set of monitors $\mathcal{M} = \{m_1, \ldots, m_k\}$, where each monitor $m_i \in \mathcal{M}$ is a tuple (P_i, E_i, A_i) such that $E_i \subseteq \Sigma$ specifies the subset of

events of interest for the monitor m_i, P_i is a collection of properties over E_i, and A_i is a structure (N_i, ν_i) such that N_i is the set of states that the monitor m_i can reach, and $\nu_i : N_i \rightarrow \mathcal{G}_i \rightarrow N_i$ is a transition function dependent on a transition guard that is a member of the set of guarded expressions $\mathcal{G}_i$. Each guarded expression is expressed as the success or the failure of one property in P_i. Properties in P_i can be expressed as logical expressions and *extended regular expressions* (ERE) inductively defined over E_i.

Logical expressions extend the traditional propositional logic with the time-related predicates `time`$(\alpha) \odot$ `val`, `duration`$(j_i) \odot$ `val` and `jitter`$(\rho) \odot$ `val`, where α is an ERE, j_i is the identifier of a job (see section II), ρ is either a `time` or a `duration` predicate, $\odot \in \{<, \leq, =, \geq, >\}$, and `val` is a natural number. Assuming that the function Δ returns the timestamp associated with any event in Σ, the semantics of the previous three predicates are defined as follows: if $first$ and $last$ are the events that denote the start and the end of α, respectively, then `time`$(\alpha) \odot$ `val` holds iff $(\Delta(last)$ - $\Delta(first)) \odot$ `val`; similarly, let $start$, $susp_k$, res_k, and $comp$ be the events that denote the start, the k^{th} suspension, the k^{th} resumption, and the completion of the job j_i, then `duration`$(j_i) \odot$ `val` holds iff $(\Delta(comp)$ - $\Delta(start)$ - $\sum_k(\Delta(_{res_k}) - \Delta(_{susp_k}))) \odot$ `val`; finally for the case of `jitter`$(\rho) \odot$ `val`, the predicate holds iff $(\max_t(\rho) - \min_t(\rho)) \odot$ `val` where $\max_t$ and $\min_t$ return the maximum and minimum value of ρ until time t.

Formally, *Extended Timed Regular Expressions* (ETREs) used in REVERT are defined as follows. Let Σ be a nonempty finite set of alphabets and let I be a closed interval $[a, b]$ with $a, b \in \mathbb{N}^+$ and $a \leq b$. The set of *Extended Timed Regular Expressions (ETRE)* is inductively defined by the following BNF grammar:

$$\alpha \ ::= \ 0 \,|\, 1 \,|\, e \in \Sigma \,|\, \alpha \vee \alpha \,|\, \alpha.\alpha \,|\, \alpha^\star \,|\, \langle\alpha\rangle_I \,|\, \square\,\alpha.$$

where 0 is the empty set, 1 is the set containing the null string, '$\vee$' is the logical or, '.' is the concatenation, '$\star$' is the Kleene's star operator, $\langle\alpha\rangle_I$ is defined as `time`$(\alpha) \in I$, and $\square$ is a newly introduced operator. The introduction of the operator $\square$ is based on the observation that regular expressions may become extremely complex when (i) the number of monitored event increases but (ii) some properties refer only to a small subset of those events. For instance, specifying that the response time of a task must be smaller than its deadline would require to express all possible sequence of events that do not comprise the completion event between the start and completion of the task. However, using the $\square$ operator, the same property can simply be written as `time`$(start \,\square\, comp) < deadline$. $\square$ operator is formally defined as follows: considering that $\mathcal{L}(\alpha) \subseteq \Sigma^\star$ is the language denoted by the event expression α, the language of $\mathcal{L}(\square\,\alpha)$ is defined as the set of all words $w = w_1 w_2$ such that $w_2 \in \mathcal{L}(\alpha)$, and w_1 does not contain any word in the language denoted by α. Note that we did not use the complement operator to ensure determinism [1].

Except for the newly introduced operator $\square\,\alpha$, the syntax of ETRE is the same as of classic timed regular expressions, as well as their semantic interpretation in the domain of timed languages.

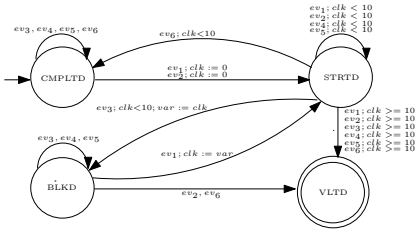

Fig. 1: FSM of the expression `failure`(`duration`(j_1)<10)

IV. MONITOR GENERATION

In order to generate the monitor that will be running beside the application, we transform the specification to a complete deterministic finite automaton with the notion of time. Note that the automaton generation occurs before run-time and the automaton has a maximum of one transition per event occurrence. This enables to generate a monitor with time and space guarantees by avoiding the potential blowup of states in run-time, irrespective of the size and complexity of autmaton from which the monitor is generated. The determinism and finiteness of the automaton ensures that the generated monitor will be tracking one single state at any time. To the best of our knowledge no other RV tool with the notion of time gives state-space and time guarantees.

The generation of monitors is achieved through the following steps: 1) Generating an automaton for each transition; 2) Generating an automaton for each node n_i by applying a product operation on the automata obtained for each transition from n_i to any other node. As mentioned in section II, implicit priority is used to resolve potential conflicts on the final state; 3) Generating the monitor automaton by concatenating the automata of all nodes. The monitor automaton is then converted to XML format which can be used to produce code.

As discussed in Section III, each individual constraint can either be an ETRE or a logical expression on the duration of job or the jitter of a time property. The transformation of a logical expression `duration` or `jitter` in a timed automaton, is implemented using predefined templates. For example, Fig. 1 shows `failure`(`duration`(j_1) < 10), where j_1 is defined in Listing 1 and the list of observed events are ev_1 to ev_6.

For transitions based on ETRE however, the traditional automaton construction methods were incapable of generating complete deterministic finite timed automaton from timed expressions as the ones offered by REVERT. We extended the notion of derivative of a regular expression that was introduced in the 60's by Brzozowski [3], with a notion of *pseudo-integral*. In this extension, as presented here, we do not consider the $\square$ operator. Given an ETRE α and a timestamped event $\rho_i = (ev_i, t_i)$, informally the derivation process will return a new ETRE that removes the event ev_i from the head of all traces that are members of the language denoted by α; pseudo-integration process will give an ETRE as result

which will accept the language formed by appending event ev_i to the language of α. By applying these methods finitely many times with respect to all events of interest, the result will be a finite automaton that recognizes all the words of the original expression α. We now provide the formal definition of this method, but first we need to provide a syntactic function that can decide whether or not the empty trace belongs to the language of the expression given to be derived or integrated.

Definition 1 (Empty trace membership): Let Σ be a non-empty finite set of events, and let α be an ETRE defined over Σ. The syntactic emptiness function is inductively defined as follows:

$$E(0) = \textbf{false} \quad E(1) = \textbf{true} \quad E(a) = \textbf{false}, a \in \Sigma$$
$$E(\alpha \vee \beta) = E(\alpha) \vee E(\beta) \qquad E(\alpha \cdot \beta) = E(\alpha) \wedge E(\beta)$$
$$E(\alpha^\star) = \textbf{true} \quad E(\langle\alpha\rangle_I) = E(\alpha)$$

Definition 2 (Derivative): Let Σ be a non-empty finite set of events, let α be an ETRE, and let $\rho = (ev, t)$ be a timed symbol with $ev \in \Sigma$ and $t \in \mathbb{T}$, where $\mathbb{T}$ is a time domain. The derivative of α with respect to ρ, denoted as $\mathcal{D}_\rho(\alpha)$, is inductively defined as follows:

$$\mathcal{D}_\rho(0) = 0 \quad \mathcal{D}_\rho(1) = 0 \qquad \mathcal{D}_\rho(\alpha^\star) = \mathcal{D}_\rho(\alpha) \cdot \alpha^\star$$
$$\mathcal{D}_\rho(\alpha) = \begin{cases} 1, & \text{if } \alpha = ev; \\ 0, & \text{otherwise.} \end{cases} \qquad \mathcal{D}_\rho(\langle\alpha\rangle_I) = \begin{cases} \langle\mathcal{D}_\rho(\alpha)\rangle_{I-t}, & \text{if } I - t \neq \emptyset; \\ 0, & \text{otherwise.} \end{cases}$$
$$\mathcal{D}_\rho(\alpha_1 \vee \alpha_2) = \mathcal{D}_\rho(\alpha_1) \vee \mathcal{D}_\rho(\alpha_2)$$
$$\mathcal{D}_\rho(\alpha_1 \cdot \alpha_2) = \mathcal{D}_\rho(\alpha_1) \cdot \alpha_2 \ \vee \ E(\alpha_1) \cdot \mathcal{D}_\rho(\alpha_2)$$

Definition 3 (Pseudo Integral): Let Σ be a non-empty finite set of events, let α be an ETRE, and let $\rho = (ev, t)$ be a timed symbol with $ev \in \Sigma$ and $t \in \mathbb{T}$, where $\mathbb{T}$ is a time domain. The integral of α with respect to ρ, denoted as $\mathcal{I}_\rho(\alpha)$, is inductively defined as follows:

$$\mathcal{I}_\rho(0) = 0 \quad \mathcal{I}_\rho(1) = ev \qquad \mathcal{I}_\rho(\alpha_1 \cdot \alpha_2) = \alpha_1 \cdot \mathcal{I}_\rho(\alpha_2)$$
$$\mathcal{I}_\rho(\alpha) = \begin{cases} \alpha, & \text{if } \alpha = ev^\star; \\ \alpha \cdot ev, & \text{otherwise.} \end{cases} \qquad \mathcal{I}_\rho(\alpha_1 \vee \alpha_2) = \mathcal{I}_\rho(\alpha_1) \vee \mathcal{I}_\rho(\alpha_2)$$
$$\mathcal{I}_\rho(\langle\alpha\rangle_I) = \begin{cases} \langle\mathcal{I}_\rho(\alpha)\rangle_{I-t}, & \text{if } I - t \neq \emptyset; \\ 0, & \text{otherwise.} \end{cases} \qquad \mathcal{I}_\rho(\alpha^\star) = \alpha^\star \cdot \mathcal{I}_\rho(\alpha)$$

We propose Algorithm 1 to build a complete deterministic finite timed automata from the logical expression `time`(α).

Algorithm 1:

every state $state_i$ is associated with two variables $REex_i$ and $REel_i$;
add start state ($state_0$) to the set $waiting\_states$;
reset clock variable $main\_clock$;
$REex_0 := \alpha$;
$REel_0 := 0$;
for *all* $state_i \in waiting\_states$ **do**
  for *all* $\rho \in \Sigma$ **do**
    if $\mathcal{D}_\rho(REex_i) \neq 0$ **then**
      if $\exists state_j \in waiting\_states$ *s. t.* $\mathcal{D}_\rho(REex_i) \in REex_j$ **then**
        $REel_j := REel_j \vee \mathcal{I}_\rho(REel_i)$;
      else if $\mathcal{D}_\rho(REex_i) = 1$ **then**
        create a new final state $state_j$;
        $REex_j := 1$;
        $REel_j := \mathcal{I}_\rho(REel_i)$;
      else
        add a new state $state_j$ to $waiting\_states$;
        $REex_j := \mathcal{D}_\rho(REex_i)$;
        $REel_j := \mathcal{I}_\rho(REel_i)$;
      end
      create a transition from $state_i$ to $state_j$;
    else
      LSI := longest suffix of $\mathcal{I}_\rho(REel_i)$ matched with $REel_j$ for any state $state_j \in waiting\_states$;
      if LSI *is empty* **then**
        create a transition from $state_j$ to $state_0$;
      else if *length of* LSI *= 1* **then**
        create a self-loop on $state_i$ with $main\_clock$ reset;
      else
        add an auxiliary clock aux_clk_i;
        RE_{pre} := longest prefix of $\mathcal{I}_\rho(REel_i)$ before LSI;
        reset aux_clk_i at $state_k \in waiting\_states$ s. t. $RE_{pre} = REel_k$;
        create a transition from $state_i$ to $state_j$ with $main\_clock$ set to value of aux_clk_i;
      end
    end
  end
end

V. Conclusion and Future Work

In this paper, we have presented REVERT, a specification language for performing RV on RTS. We proposed a novel method to generate complete deterministic timed automata from the specification, that avoids blowup in the number of states at run-time suffered by other state-of-the-art tools. As future work, we first plan to formally prove the correctness of the algorithm presented in this paper and extend it to support the $\square$ operator. Secondly, we will compare the expressivity of our language with state-of-the-art tools. Finally, we will bound the time and space complexity of the generated monitors.

Acknowledgments

This work was partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within the CISTER Research Unit (CEC/04234); also by ARTEMIS/0001/2013 - JU grant nr. 621429 (EMC2).

References

[1] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[2] Borzoo Bonakdarpour, Johnson J Thomas, and Sebastian Fischmeister. Time-triggered program self-monitoring. In *RTCSA*, pages 260–269. IEEE, 2012.

[3] Janusz A Brzozowski. Derivatives of regular expressions. *JACM*, 11(4):481–494, 1964.

[4] Feng Chen and Grigore Roşu. Mop: An efficient and generic runtime verification framework. OOPSLA, New York, NY, USA, 2007. ACM.

[5] Klaus Havelund. Runtime verification of c programs. In *ICTSS: 8th International Workshop*, TestCom '08 / FATES '08, pages 7–22, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] Klaus Havelund. Rule-based runtime verification revisited. *STTT*, 17(2):143–170, 2015.

[7] Andrew Kornecki and Janusz Zalewski. Software certification for safety-critical systems: A status report. In *IMCSIT 2008.*, pages 665–672. IEEE, 2008.

[8] Jonathan Laurent, Alwyn Goodloe, and Lee Pike. Assuring the guardians. In *Runtime Verification*, pages 87–101. Springer, 2015.

[9] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009.

[10] Samaneh Navabpour, Yogi Joshi, Wallace Wu, Shay Berkovich, Ramy Medhat, Borzoo Bonakdarpour, and Sebastian Fischmeister. Rithm: a tool for enabling time-triggered runtime verification for c programs. In *FSE*. ACM, 2013.

[11] Usa Sammapun, Insup Lee, and Oleg Sokolsky. RT-MaC: Runtime monitoring and checking of quantitative and probabilistic properties. In *RTCSA 2005*, pages 147–153. IEEE Computer Society, 2005.

[12] Haitao Zhu, Matthew B Dwyer, and Steve Goddard. Predictable runtime monitoring. In *ECRTS*, pages 173–183. IEEE, 2009.