**Noname manuscript No.**
(will be inserted by the editor)

# Time-triggered Runtime Verification

**Borzoo Bonakdarpour · Samaneh Navabpour · Sebastian Fischmeister**

**Abstract** The goal of runtime verification is to monitor the behavior of a system to check its conformance to a set of desirable logical properties. The literature of runtime verification mostly focuses on *event-triggered* solutions, where a monitor is invoked when an event of interest occurs (e.g., change in the value of some variable). At invocation, the monitor evaluates the set of properties of the system that are affected by the occurrence of the event. This constant invocation introduces two major defects to the system under scrutiny at run time: (1) significant *overhead*, and (2) *unpredictability* of behavior. These defects are serious obstacles when applying runtime verification on safety-critical systems that are time-sensitive by nature.

To circumvent the aforementioned defects in runtime verification, in this article, we introduce a novel *time-triggered* approach, where the monitor takes samples from the system with a constant frequency, in order to analyze the system's health. We describe the formal semantics of time-triggered monitoring and discuss how to optimize the sampling period using minimum auxiliary memory. We show that such optimization is NP-complete and consequently introduce a mapping to *Integer Linear Programming*. Experiments on a real-time benchmark suite show that our approach introduces bounded overhead and effectively reduces the involvement of the monitor at run time by using negligible auxiliary memory. We also show that in some cases it is even possible to reduce the overall overhead of runtime verification by using our time-triggered approach when the structure of the system allows choosing a long enough sampling period.

**Keywords** Runtime verification, monitoring, time-triggered, predictability, overhead, real-time, embedded systems.

B. Bonakdarpour
School of Computer Science
University of Waterloo
200 University Avenue West, Waterloo, Ontario, Canada, N2L 3G1
E-mail: borzoo@cs.uwaterloo.ca

S. Navabpour and S. Fischmeister
Department of Electrical and Computer Engineering
University of Waterloo
200 University Avenue West, Waterloo, Ontario, Canada, N2L 3G1
E-mail: {snavabpo, sfischme}@uwaterloo.ca

## 1 Introduction

In a computing system, *correctness* refers to the assertion that a system satisfies its specification. *Runtime verification* [5, 6, 16, 21, 23, 43] refers to a technique, where a *monitor* checks at run time whether or not the execution of a system under inspection satisfies a given correctness property. Runtime verification complements exhaustive verification methods such as model checking and theorem proving, as well as incomplete solutions such as testing and debugging. Exhaustive verification often requires developing a rigorous abstract model of the system and suffers from the infamous *state-explosion problem*. Testing and debugging, on the other hand, provide us with under-approximated confidence about the correctness of a system as these methods only check for the presence of defects for a limited set of scenarios.

Constructing a monitor for runtime verification normally involves synthesizing an automaton for each property that the system under scrutiny must satisfy [34]. Then, by composing the monitor with the system, the monitor observes the occurrence of each transition and decides whether the specification has been met, violated, or impossible to tell. Thus, the monitor is invoked by the execution of every event in the system which may affect the valuation of the properties (e.g., change in the value of a variable). We call this type of monitoring *event-triggered*. The main challenge in augmenting a system with runtime verification is dealing with the runtime *overhead* of monitor invocations. Several techniques have been introduced in the literature for reducing and controlling runtime monitoring overhead. Examples include:

- improved instrumentation (e.g., using aspect-oriented programming [14, 46]),
- combining static and dynamic analysis techniques (e.g., using typeset analysis [8] and PID controllers [28]), and
- efficient monitor generation and management (e.g., in the case of parametric monitors [39]).

Although the aforementioned approaches assist in reducing the overhead, the event-triggered monitor has two characteristics which may cause defects in the behaviour of the system under scrutiny: (1) *unpredictable* invocation of the monitor due to different patterns in the occurrence of monitored events for different execution scenarios of the program, and (2) possible bursts of monitoring invocation due to the uneven distribution of the occurrence of monitored events throughout the program execution. These characteristics can lead to undesirable transient overload situations in time-sensitive systems. This is because time predictability is the key ingredient in designing real-time systems. Time predictability makes it easier to respect timing constraints and hard real-time deadlines, and hence, bursts of monitoring invocation can cause the monitored program to violate its timing constraints. We currently lack a rigorous method to design and deploy runtime monitors suitable for time-sensitive systems. Such a monitor should intervene with the program execution in a predictable and timely fashion, making it possible for a system designer to reason about the timing constraints of the monitored program in a straightforward manner.

With this motivation, in this paper, we propose an alternative and novel approach for runtime verification of *sequential* systems, where the monitor is *time-triggered*. The idea is that the monitor wakes up with a *fixed* frequency and takes samples from the system under inspection in order to analyze the system's correctness. This way, the involvement of the monitor is time-bounded and predictable. Such predictability makes a time-triggered monitor a perfect module in real-time systems, especially when

they are constrained by hard real-time deadlines. However, the main challenge in this mechanism is accurate reconstruction of the system's state between two samples. For instance, if the value of a variable that should be monitored changes more than once between two samples, the monitor may fail to detect violations of some properties. Hence, the problem boils down to finding the *longest possible sampling period* that allows state reconstruction.

Given a program and a set of (desired) variables to be monitored, in order to calculate the longest sampling period, one has to consider three factors: (1) the instructions that change the value of the desired variables, (2) execution paths of the program, and (3) the time interval between the execution of instructions that change the value of the desired variables. Our method first constructs the program's control-flow graph. A vertex in this graph is a basic block of one instruction and an edge between two vertices exist if execution of one may immediately lead to the execution of the other. Each edge is associated with a weight, which is the best-case execution time of the source vertex. The longest sampling period for runtime monitoring is the minimum shortest path between two vertices which incorporate instructions that change the value of a desired variable. This sampling period ensures that *all* state changes (i.e., changes in the value of desired variables) vital to sound evaluation of properties are observed at run time.

We employ the longest sampling period to define the formal semantics of time-triggered runtime verification using the timed automata formalism [2]. We define in formal terms the behavior of a time-triggered monitor and how it interrupts and evaluates a set of properties at runtime by using the 3-valued Linear Temporal Logic [7] and parallel composition of timed automata. We also argue that our method can be extended for monitoring systems with respect to real-time extensions of temporal logics such as the 3-valued Timed Linear Temporal Logic [7].

The longest sampling period extracted from a control-flow graph tend to be short, and hence, precipitates highly frequent invocations of the monitor even in branches of the program that do not require monitoring. In other words, the system under inspection by a time-triggered monitor is likely to suffer from the *redundant sampling* phenomenon. To tackle this problem, we propose a method for increasing the sampling period by incorporating auxiliary memory, where we store a history of state changes. As a result, extending the sampling period does not result in overlooking state changes vital to the evaluation of properties, as they are kept in the history. Thus, when the monitor wakes up and samples the program state, it also reads the recorded history of events. Naturally, it is desirable to use the least amount of history while allowing the least monitoring invocation. More formally, we face a tradeoff between minimizing the size of auxiliary memory to build the state history versus maximizing the sampling period. We show that the corresponding optimization problem is NP-complete.

In order to cope with the exponential complexity of the optimization problem, we map the problem onto *integer linear programming* (ILP). Our tool chain RiTHM[1] takes a C program as input, instruments the program to build optimal history and constructs a monitor that takes samples with the optimal sampling period. In particular, we generate the control-flow graph of a given C program using the tool LLVM [35]. Next, we generate the critical control-flow graph, which encapsulates the state changes caused by monitored variables. This graph and the optimization problem are then transformed

[1] To access the tool, please visit `http://uwaterloo.ca/embedded-software-group/projects/rithm`.

into an ILP model. The model is given to the tool lp_solve [37] to obtain the optimal sampling period and the size of auxiliary memory. Moreover, the solution to the ILP model specifies what instrumentation instructions must be added to the input program for building up history. Finally, a time-triggered monitor is automatically generated for an MCB1700 board. It interrupts the program execution with respect to the obtained sampling period, reads the program state and history variables and evaluates properties.

We report the results of comprehensive experiments on the SNU [1] benchmark suite (designed for real-time systems) to study the effect of different factors on time-triggered monitoring. These factors include the longest sampling period, extended sampling period using history, and desired variables for monitoring. We measure the impact of a time-triggered monitor on metrics important to deploying a system augmented with a time-triggered monitor. These metrics include the added instrumentation, amount of monitoring overhead, the monitoring overhead pattern and predictability, overhead jitter, program execution time, auxiliary memory usage, the number of variables stored in the history, and redundant sampling. Our experimental results are highly encouraging. First, the size of ILP models for real-world applications are quite manageable. Second, we observe that in event-triggered implementations, the system suffers from bursts of monitor involvement, whereas our time-triggered monitor adds bounded, and hence, predictable overhead. Finally, we observe that the memory usage overhead is negligible and our method effectively increases the sampling period, which results in adding less overall monitoring overhead at runtime, and in some cases obtaining faster execution of the program as compared to event-triggered methods.

**Organization.** The rest of the paper is organized as follows. We present the preliminary concepts in Section 2. Formal semantics of time-triggered runtime monitoring is discussed in Section 3. Then, in Section 4, we introduce our method for optimizing the sampling period using auxiliary memory and analyze its complexity. Section 5 presents our transformation to ILP. Section 6 explains our implementation, tool chain, and experimental settings, while Section 7 is dedicated to experimental results. Related work is discussed in Section 8. Finally, we make concluding remarks and discuss future work in Section 9.

## 2 Preliminaries

In this section, we present the preliminary concepts. In Subsection 2.1, we present the notion of *control-flow graphs* for analyzing timing characteristics of programs written in high-level programming languages. In Subsections 2.2 and 2.3, we present the concept of timed automata [2] and 3-valued linear temporal logic [7, 32, 36], respectively, as basis for presenting the semantics of time-triggered runtime verification.

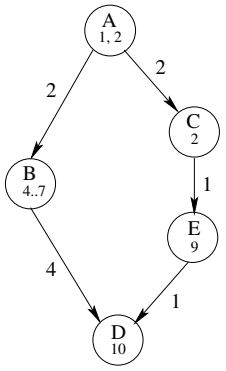
### 2.1 Control-flow Graphs

**Definition 1** The *control-flow graph* of a program $P$ is a weighted directed simple graph $CFG_P = \langle V, v^0, A, w \rangle$, where:

- $V$: is a set of *vertices*, each representing a basic block of $P$. Each basic block consists of a sequence of instructions in $P$.

```
1:      a = scanf(...);
2:      if (a % 2 == 0) goto 9
3:      else {
4:            printf(a + "is odd");
5:*           b = a/2;
6:*           c = a/2 + 1;
7:            goto 10;
8:      }
9:      printf(a + "is even");
10:     end program
```

(a) A simple C program




(b) Control-flow graph

**Fig. 1** A C program and its control-flow graph.

- $v^0$: is the *initial vertex* with indegree 0, which represents the initial basic block of $P$.
- $A$: is a set of *arcs* of the form $(u, v)$, where $u, v \in V$. An arc $(u, v)$ exists in $A$, if and only if the execution of basic block $u$ can immediately lead to the execution of basic block $v$.
- $w$: is a function $w : A \to \mathbb{N}$, which defines a *weight* for each arc in $A$. The weight of an arc is the *best-case execution time* (BCET) of the source basic block[2]. ■

*Notation:* Let $v$ be a vertex of a control-flow graph. Since the weight of all outgoing arcs from $v$ are equal, we denote the weight of the arcs that originate from $v$ by $w(v)$.

For example, consider the C program in Figure 1(a). If each instruction takes one time unit to execute, the resulting control-flow graph is shown in Figure 1(b). Vertices of the graph in Figure 1(b) are annotated by the corresponding line numbers of the C program in Figure 1(a).

### 2.2 Timed Automata

Let $AP$ be a finite set of *atomic propositions* and $\Sigma = 2^{AP}$ be a finite *alphabet*. A letter $a$ in $\Sigma$ is interpreted as assigning truth values to the elements of $AP$; i.e., elements in $a$ are assigned true (denoted $\top$) and elements not in $a$ are assigned false (denoted $\bot$). A *timed word* over $\Sigma$ is a sequence $(a_0, t_0), (a_1, t_1) \cdots (a_k, t_k)$, where each $a_i \in \Sigma$ and each $t_i$ is in non-negative real numbers $\mathbb{R}_{\geq 0}$ and the occurrence times increase monotonically. Let $X$ be a set of *clock variables*. A *clock constraint* over $X$ is a Boolean combination of formulae of the form $x \preceq c$ or $x - y \preceq c$, where $x, y \in X$, $c \in \mathbb{Z}_{\geq 0}$, and $\preceq$ is either $<$ or $\leq$. We denote the set of all clock constraints over $X$ by $\Phi(X)$. A *clock valuation* is a function $\nu : X \to \mathbb{R}_{\geq 0}$ that assigns a real value to each clock variable. For $\tau \in \mathbb{R}_{\geq 0}$, we write $\nu + \tau$ to denote $\nu(x) + \tau$ for every clock variable $x$ in $X$. Also, for $\lambda \subseteq X$,

[2] In Section 3, we will compute the longest sampling period of a CFG based on BCET of basic blocks. This computation is quite realistic, as (1) all hardware vendors publish the BCET of their instruction set in terms of clock cycles, and (2) BCET is a conservative approximation and no execution occurs faster than that.

$\nu[\lambda := 0]$ denotes the clock valuation that assigns 0 to each $x \in \lambda$ and agrees with $\nu$ over the rest of the clock variables in $X$.

**Definition 2** A *timed automaton* is a tuple $\mathcal{A} = \langle L, L^0, X, \Sigma, E, I \rangle$, where

- $L$ is a finite set of *locations*.
- $L^0 \subseteq L$ is a set of *initial locations*.
- $X$ is a finite set of clock variables.
- $\Sigma$ is a finite set of labels.
- $E \subseteq (L \times \Sigma \times 2^X \times \Phi(X) \times L)$ is a set of *switches*. A switch $\langle l, a, \lambda, \varphi, l' \rangle$ represents a transition from location $l$ to location $l'$ labelled by $a$, under clock constraint $\varphi$. The set $\lambda \subseteq X$ gives the clocks to be reset with this switch.
- $I : L \to \Phi(X)$ assigns a *delay invariant* to a location. ■

The semantics of a timed automaton $\mathcal{A}$ is as follows. A *state* is a pair $(l, \nu)$, where $l \in L$ and $\nu$ is a clock valuation for $X$. A state $(l, \nu)$ is an initial state if $l \in L^0$ and $\nu(x) = 0$ for all $x \in X$. There are two types of *transitions*:

1. *Location switches* are of the form $\langle l, a, \lambda, \varphi, l' \rangle$, such that $\nu$ satisfies $\varphi$, $(l, \nu) \xrightarrow{a} (l', \nu[\lambda := 0])$, and $\nu[\lambda := 0]$ satisfies $I(l')$.
2. *Delay transitions* are of the form $(l, \nu) \xrightarrow{\tau} (l, \nu + \tau)$, which preserves the location $l$ for time duration $\tau \in \mathbb{R}_{\geq 0}$, such that for all $0 \leq \tau' \leq \tau$, $\nu + \tau'$ satisfies the delay invariant $I(l)$.

For a timed word $w = (a_0, t_0), (a_1, t_1) \cdots (a_k, t_k)$, a *run* over $w$ is a sequence

$$q_0 \xrightarrow{t_0} q_0' \xrightarrow{a_0} q_1 \xrightarrow{t_1 - t_0} q_1' \xrightarrow{a_1} q_2 \xrightarrow{t_2 - t_1} q_2' \xrightarrow{a_2} q_3 \to \cdots \xrightarrow{a_k} q_{k+1}$$

such that $q_0$ is an initial state.

Let $\mathcal{A}_1 = \langle L_1, L_1^0, X_1, \Sigma_1, E_1, I_1 \rangle$ and $\mathcal{A}_2 = \langle L_2, L_2^0, X_2, \Sigma_2, E_2, I_2 \rangle$ be two timed automata, where $X_1 \cap X_2 = \emptyset$. The *parallel composition* of $\mathcal{A}_1$ and $\mathcal{A}_2$ is $\mathcal{A}_1 || \mathcal{A}_2 = \langle L_1 \times L_2, L_1^0 \times L_2^0, X_1 \cup X_2, \Sigma_1 \cup \Sigma_2, E, I \rangle$, where $I(l_1, l_2) = I(l_1) \ \wedge \ I(l_2)$, and $E$ is defined by:

1. for $a \in \Sigma_1 \cap \Sigma_2$, for every $\langle l_1, a, \lambda_1, \varphi_1, l_1' \rangle$ in $E_1$, and $\langle l_2, a, \lambda_2, \varphi_2, l_2' \rangle$ in $E_2$, $E$ contains $\langle (l_1, l_2), a, \lambda_1 \cup \lambda_2, \varphi_1 \wedge \varphi_2, (l_1', l_2') \rangle$.
2. for $a \in \Sigma_1 \backslash \Sigma_2$, for every $\langle l, a, \lambda, \varphi, l' \rangle$ in $E_1$, and every $m \in L_2$, $E$ contains $\langle (l, m), a, \lambda, \varphi, (l', m) \rangle$.
3. for $a \in \Sigma_2 \backslash \Sigma_1$, for every $\langle l, a, \lambda, \varphi, l' \rangle$ in $E_2$, and every $m \in L_1$, $E$ contains $\langle (m, l), a, \lambda, \varphi, (m, l') \rangle$.

### 2.3 3-Valued Linear Temporal Logic (LTL)

*Linear temporal logic* (LTL) [42] is a popular formalism for specifying properties of (concurrent) programs. The set of well-formed linear temporal logic formulas is constructed from a set of atomic propositions, the standard Boolean operators, and temporal operators. A *word* is a finite or infinite sequence of letters $w = a_0 a_1 a_2 \ldots$, where $a_i \in \Sigma$ for all $i \geq 0$. We denote the set of all finite words over $\Sigma$ by $\Sigma^*$ and the set of all infinite words by $\Sigma^\omega$. For a finite word $u$ and a word $w$, we write $u \cdot w$ to denote their *concatenation*.

**Definition 3 (LTL Syntax)** LTL formulas are defined inductively as follows:

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \mathbf{U} \varphi_2$$

where $p \in \Sigma$, and $\bigcirc$ (next) and $\mathbf{U}$ (until) are temporal operators. ■

**Definition 4 (LTL Semantics)** Let $w = a_0 a_1 \ldots$ be an infinite word in $\Sigma^\omega$, $i$ be a non-negative integer, and $\models$ denote the *satisfaction* relation. Semantics of LTL is defined inductively as follows:

$$\begin{array}{lll}
w, i \models \top & & \\
w, i \models p & \text{iff} & p \in a_i \\
w, i \models \neg\varphi & \text{iff} & w, i \not\models \varphi \\
w, i \models \varphi_1 \vee \varphi_2 & \text{iff} & w, i \models \varphi_1 \ \vee \ w, i \models \varphi_2 \\
w, i \models \bigcirc\varphi & \text{iff} & w, i+1 \models \varphi \\
w, i \models \varphi_1 \,\mathbf{U}\, \varphi_2 & \text{iff} & \exists k \geq i : w_k \models \varphi_2 \ \wedge \ \forall i \leq j \leq k : w, j \models \varphi_1
\end{array}$$

In addition, $w \models \varphi$ holds iff $w, 0 \models \varphi$ holds. ■

Notice that an LTL formula $\varphi$ defines a set of words (i.e., a *language* or a *property*) that satisfies the semantics of that formula. We denote this language by $L(\varphi)$. For simplicity, we introduce abbreviation temporal operators. $\Diamond\varphi$ (*eventually* $\varphi$) denotes $\top \,\mathbf{U}\, \varphi$, and $\Box\varphi$ (*always* $\varphi$) denotes $\neg\Diamond\neg\varphi$.

Implementing runtime verification boils down to the following problem: given a *finite* word $\sigma = a_0 a_1 a_2 \ldots a_n$, check whether or not $\sigma$ belongs to a set of words defined by some property $\varphi$. This is a complex problem, because LTL semantics is defined over infinite words and a running program can only deliver a finite word at a verification point. For example, given a finite word $\sigma = a_0 a_1 \ldots a_n$, it may be impossible for a *monitor* to decide weather the property $\Diamond p$ is satisfied.

To formalize satisfaction of LTL properties at run time, in [7], the authors propose semantics for LTL, where the evaluation of a formula ranges over three values '$\top$', '$\bot$', and '?' (denoted LTL$_3$). The latter value expresses the fact that it is not possible to decide on the satisfaction of a property, given the current finite trace of the program.

**Definition 5 (LTL$_3$ semantics)** Let $u \in \Sigma^*$ be a finite word. The truth value of an LTL$_3$ formula $\varphi$ with respect to $u$, denoted by $[u \models \varphi]$, is defined as follows:

$$[u \models \varphi] = \begin{cases} \top & \text{if } \forall w \in \Sigma^\omega : u \cdot w \models \varphi, \\ \bot & \text{if } \forall w \in \Sigma^\omega : u \cdot w \not\models \varphi, \\ ? & \text{otherwise.} \end{cases}$$

■

Note that the syntax $[u \models \varphi]$ for LTL$_3$ semantics is defined over finite words as opposed to $u \models \varphi$ for LTL semantics, which is defined over infinite words. For example, given a finite program trace $\sigma = a_0 a_1 \cdots a_n$, property $\Diamond p$ holds iff $a_i \models p$, for some $i$, $0 \leq i \leq n$ (i.e., $\sigma$ is a good prefix). Otherwise, the property evaluates to ?.

**Definition 6 (Good and Bad Prefixes)** Given a language $L \subseteq \Sigma^\omega$ of infinite words over $\Sigma$, we call a finite word $u \in \Sigma^*$

– a *good prefix* for $L$, if $\forall w \in \Sigma^\omega : u \cdot w \in L$

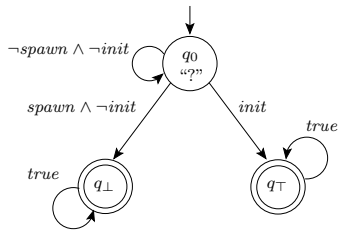



**Fig. 2** The monitor for property $\varphi \equiv (\neg spawn\ \mathbf{U}\ init)$

– a *bad prefix* for $L$, if $\forall w \in \Sigma^\omega : u \cdot w \notin L$
– an *ugly* prefix otherwise. ■

Implementing runtime verification for an LTL$_3$ property involves synthesizing a monitor that realizes the property. In [7], the authors introduce a stepwise method that takes an LTL$_3$ property $\varphi$ as input and generates a deterministic finite state machine (FSM) $\mathcal{M}^\varphi$ as output. Intuitively, simulating a finite word $u$ on this FSM reaches a state that illustrates the valuation of $[u \models \varphi]$.

**Definition 7 (Monitor)** Let $\varphi$ be an LTL$_3$ formula over alphabet $\Sigma$. The *monitor* of $\varphi$ is the unique FSM $\mathcal{M}^\varphi = (\Sigma, Q, q_0, \delta, \lambda)$, where $Q$ is a set of states, $q_0$ is the initial state, $\delta$ is the transition relation, and $\lambda$ is a function that maps each state in $Q$ to a value in $\{\top, \bot, ?\}$, such that:

$$[u \models \varphi] = \lambda(\delta(q_0, u)).$$

■

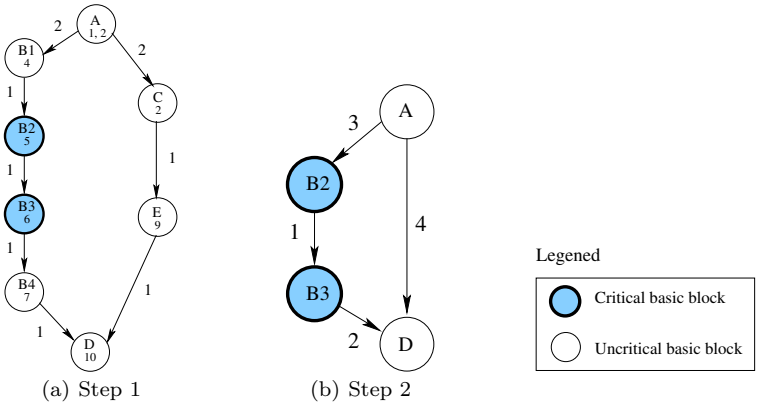
For example, consider the property $\varphi \equiv (\neg spawn\ \mathbf{U}\ init)$ (i.e., a thread is not spawned until it is initialized). The corresponding monitor is shown in Figure 2 [7]. The proposition *true* denotes the set $AP$ of all propositions. We use the term a *conclusive state* to refer to monitor states $q_\top$ and $q_\bot$; i.e., a state where $\lambda(q) = \top$ and $\lambda(q) = \bot$, respectively. Other states are called *inconclusive states*. A monitor $\mathcal{M}^\varphi$ is constructed in a way that it recognizes good, bad, and ugly prefixes of $L(\varphi)$. Hence, a conclusive state is in fact also a *trap* state. In other words, if $\mathcal{M}^\varphi$ reaches a conclusive state, it stays in this state.

## 3 Formal Semantics of Time-triggered Monitoring

Given a program $P$, we describe the semantics of time-triggered monitoring in two steps: (1) identifying the longest sampling period, and (2) constructing a time-triggered monitor and composing it with $P$. Then, we show that the obtained composition can effectively verify a rich fragment of LTL$_3$ properties at run time.

### 3.1 Calculating the Longest Sampling Period

Let $P$ be a program and $\varphi$ be an LTL$_3$ property, where $P$ is expected to satisfy $\varphi$. Let $\mathcal{V}_\varphi$ denote the set of variables that can change the valuation of the atomic propositions




(a) Step 1 (b) Step 2

**Fig. 3** Obtaining a critical CFG and calculating the sampling period.

in $\varphi$. For example, in property $\varphi \equiv \Box\Diamond(x \geq 0 \ \wedge \ y = 10)$, we have $\mathcal{V}_\varphi = \{x, y\}$. Generally, in our time-triggered monitoring, a *sampler* process periodically wakes up with some sampling period, reads the value of variables in $\mathcal{V}_\varphi$ from program $P$, and passes them to a monitor $\mathcal{M}^\varphi$ (as described in Subsection 2.3) to evaluate $\varphi$. The sampler process is discussed in more detail in Subsection 3.2. The main challenge in this mechanism is accurate reconstruction of the states that $P$ takes in between two consecutive samples from the sampler process. For instance, if the value of a variable in $\mathcal{V}_\varphi$ changes more than once between two samples, the sampler process can only extract the last value of the variable. Hence, the monitor may fail to evaluate property $\varphi$ correctly.

In order to accurately sample all the changes in the value of variables in $\mathcal{V}_\varphi$, we modify $CFG_P$ as follows. In the first step, we ensure that each *critical instruction* (i.e., an instruction that modifies the value of a variable in $\mathcal{V}_\varphi$) is in a basic block that contains no other instructions. We refer to such a basic block as *critical basic block* or *critical vertex*. Formally, let $inst_v = \langle v^1 \cdots v^n \rangle$ denote the sequence of instructions in a basic block $v$ of $CFG_P$. Let $v^i$, where $1 < i < n$, be the one and only critical instruction in $inst_v$. We split vertex $v$ into three vertices $v_1$, $v_2$, and $v_3$, such that $inst_{v_1} = \langle v_1^1 \cdots v_1^{i-1} \rangle$, $inst_{v_2} = \langle v_2^i \rangle$, and $inst_{v_3} = \langle v_3^{i+1} \cdots v_3^n \rangle$. Incoming arcs to $v$ now enter $v_1$. We add arc $(v_1, v_2)$, where $w(v_1, v_2)$ is equal to the best-case execution time of $\langle v_1^1 \cdots v_1^{i-1} \rangle$. We also add arc $(v_2, v_3)$, where $w(v_2, v_3)$ is equal to the best-case execution time of $\langle v_2^i \rangle$. Outgoing arcs from $v$ now leave $v_3$ with weight equal to the best-case execution time of $\langle v_3^{i+1} \cdots v_3^n \rangle$. Obviously, if $i = 1$ or $i = n$, we split $v$ into two vertices. We continue this procedure until each critical instruction is in a separate basic block. For example, in the program in Figure 1(a), if variables `b` and `c` are in $\mathcal{V}_\varphi$, then instructions 5 and 6 are critical, and hence, we obtain the control-flow graph in Figure 3(a).

Since non-critical vertices do not play a role in determining the sampling period, in the second step, our method collapses non-critical vertices as follows. Let $CFG = \langle V, v^0, A, w \rangle$ be a control-flow graph. *Transformation* $T(CFG, v)$, where $v \in V \backslash \{v^0\}$ and outdegree of $v$ is positive, obtains $CFG' = \langle V', v^0, A', w' \rangle$ via the following ordered steps:

1. Let $A''$ be the set $A \cup \{(u_1, u_2) \mid (u_1, v), (v, u_2) \in A\}$. Observe that if an arc $(u_1, u_2)$ already exists in $A$, then $A''$ will contain parallel arcs (such arcs can be distinguished by a simple indexing or renaming scheme). We eliminate the additional arcs in Step 3.
2. For each arc $(u_1, u_2) \in A''$,

$$w'(u_1, u_2) = \begin{cases} w(u_1, u_2) & \text{if } (u_1, u_2) \in A \\ w(u_1, v) + w(v, u_2) & \text{if } (u_1, u_2) \in A''\backslash A \end{cases}$$

3. If there exist parallel arcs from vertex $u_1$ to $u_2$, we only include the one with minimum weight in $A''$.
4. Finally, $A' = A''\backslash\{(u_1, v), (v, u_2) \mid u_1, u_2 \in V\}$ and $V' = V \setminus \{v\}$.

We clarify a special case of the above transformation, where $u$ and $v$ are two non-critical vertices with arcs $(u, v)$ and $(v, u)$ between them. Deleting one of the vertices, e.g., $u$, results in a self-loop $(v, v)$, which we can safely remove. This is simply because a loop that contains no critical instructions has no effect on the calculation of the longest sampling period.

We apply the above transformation on all non-critical vertices. We call the resulting graph a *critical control-flow graph*. Such a graph includes (1) a non-critical initial basic block, (2) possibly a non-critical vertex with outdegree zero (if the program is terminating), and (3) a set of critical vertices. Figure 3(b) shows the critical CFG of the graph in Figure 3(a).

**Definition 8** Let $CFG = \langle V, v^0, A, w\rangle$ be a critical control-flow graph. The *longest sampling period* (LSP) for $CFG$ is

$$LSP_{CFG} = \min\{w(v_1, v_2) \mid (v_1, v_2) \in A \ \wedge \ v_1 \text{ is a critical vertex}\}$$

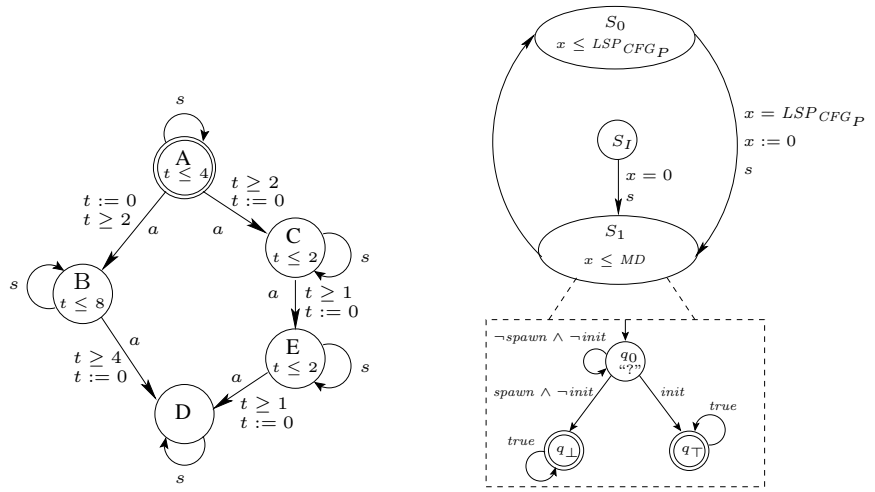
■

Intuitively, the longest sampling period is the minimum time interval between the execution of two critical instructions that change the value of a variable in $\mathcal{V}_\varphi$. For example, the longest sampling period of the control-flow graph in Figure 3(b) is $LSP = 1$. Later in this section, we show that by applying this sampling period, one can verify the correctness of a rich fragment of LTL$_3$ properties at run time.

### 3.2 Constructing and Composing a Time-triggered Monitor

We now explain the semantics of time-triggered monitoring using timed automata. We note that our implementation (as described in Section 6) does not explicitly use the transformation presented in this subsection; i.e., we solely use the timed automata formalism to describe the semantics and our implementation is a refinement of the transformation. Transformation of a control-flow graph $CFG = \langle V, v^0, A, w\rangle$ into a timed automaton $\mathcal{A}_{CFG} = \langle L, L^0, X, \Sigma, E, I\rangle$, where $X = \{t\}$ and $\Sigma = \{a, s\}$, is as follows:

- $L = \{l_v \mid v \in V\}$
- $L^0 = \{l_{v^0}\}$
- $E = \{\langle l_v, a, \{t\}, t \geq w(v, v'), l_{v'}\rangle \mid (v, v') \in A\} \cup \{\langle l_v, s, \{\}, \mathit{true}, l_v\rangle \mid v \in V\}$.




(a) Timed automaton of the CFG in Figure 1(b)

(b) Time-triggered sampler and monitor

**Fig. 4** Formal semantics of time-triggered monitoring.

– $I(l_v)$ = worst-case execution time of basic block $v \in V$.

Intuitively, $\mathcal{A}_{CFG}$ works as follows. Each location of $\mathcal{A}_{CFG}$ corresponds to one and only one vertex of *CFG*. The initial location corresponds to the initial basic block of *CFG*. Each location is associated with a delay invariant; the execution can stay in a location no longer than the worst-case execution time of the corresponding basic block. $\mathcal{A}_{CFG}$ has two types of switches. The first set of switches (labelled by $a$) change the location. Each such switch takes place when the execution of the corresponding basic block is complete. Obviously, this can happen not earlier than the best-case execution time of the basic block. The other set of switches (labelled by $s$) are self-loops and are meant to synchronize with the automaton representing the sampler process. The timed automaton obtained from the control-flow graph in Figure 1(b) is shown in Figure 4(a), where the worst-case execution time of each instruction is 2.

The relation between execution of a program $P$ and runs of the timed automaton $\mathcal{A}_{CFG_P}$ is as follows. Intuitively, a delay transition in $\mathcal{A}_{CFG_P}$ corresponds to the execution of a set of instructions in $P$. Formally, let $q = (l, t = 0)$ be a state of $\mathcal{A}_{CFG_P}$, where location $l$ hosts instructions $\{l^1 \cdots l^n\}$. An outgoing transition from this state with delay $\tau$ reaches a state $(l, t + \tau)$ which leads to executing zero or more instructions. Thus, starting from $(l, t = 0)$, a run of $\mathcal{A}_{CFG_P}$ is of the form:

$$(l, t = 0) \xrightarrow{\tau_1} (l^i, t + \tau_1) \xrightarrow{\tau_2} (l^j, t + \tau_1 + \tau_2) \xrightarrow{\tau_3} \cdots \xrightarrow{\tau_m} (l^n, t + \textstyle\sum_{k=1}^{m} \tau_k) \xrightarrow{a} (l', t = 0),$$

such that:

– $i \leq j \leq m$,
– $l \neq l'$, where $(l, l')$ is a location switch in $E$,
– $(l^i, t + \tau_1)$ denotes the fact that instructions $\langle l^1 \cdots l^i \rangle$ have been executed within $\tau_1$ time units,
– $\sum_{k=1}^{m} \tau_k \geq w(l, l')$ in $CFG_P$, and

– $(t \leq \sum_{k=1}^{m} \tau_k) \Rightarrow I(l)$.

Note that an $s$-transition may occur in such a run, but such transitions obviously do not change the current location or the value of $t$. This also holds in practice, because when the sampler process intervenes with the program execution to extract the value of variables in $\mathcal{V}_\varphi$, the execution of the program halts until the sampler process finishes its data extraction and resumes the normal operation of the program.

A sampler process $\mathcal{S}_P$ of a time-triggered monitor for program $P$ works as follows (see Figure 4(b)). From the initial location $S_I$ the only outgoing switch is enabled immediately (i.e., when $x = 0$). This switch is labeled by $s$ and the sampler synchronizes with $\mathcal{A}_{CFG_P}$ on the switch in order to read the variables in $\mathcal{V}_\varphi$ and evaluate $\varphi$. Consequently, the sampler reaches location $S_1$ and may remain in this location for at most $MD$ time units, where $MD$ is the worst-case execution time for reading the variables in $\mathcal{V}_\varphi$ and property evaluation, using the technique presented in Subsection 2.3. That is, the sampled program state is simulated on the monitor (e.g., the monitor depicted in location $S_1$ from Figure 2) automaton for property evaluation. We assume that sampling never occurs in the middle of the execution of an instruction. Normally, this assumption is already implemented, as hardware interrupts to generate time ticks are generally handled after completion of fetched instructions. After the verification step in location $S_1$, the sampler reaches location $S_0$, where it sleeps until the sampling period is complete (i.e, $x = LSP_{CFG_P}$). Thus, the parallel composition $\mathcal{A}_{CFG_P}||\mathcal{S}_P$ constructs the entire monitored system. For example, the following is a run of the automaton in Figure 4(a), composed with a sampler with sampling period $LSP = 1$ and $MD = 0$:

$$AS_I \xrightarrow{s} AS_1 \to AS_0 \xrightarrow{1} A^1S_0 \xrightarrow{s}$$

$$A^1S_1 \to A^1S_0 \xrightarrow{1} A^2S_0 \xrightarrow{s} A^2S_1 \to A^2S_0 \xrightarrow{a} BS_0 \xrightarrow{1} B^4S_0 \to \cdots$$

We call the combination of such a time-triggered sampler and a monitor (as illustrated in Figure 4(b)) a *time-triggered monitor*.

### 3.3 Correctness of Time-triggered Runtime Monitoring

In this section, we show that runtime verification using a time-triggered monitor is *sound* and *complete* for a rich fragment of LTL$_3$.

**Assumption 1** *We assume that $MD \leq LSP$.* ■

This assumption is quite realistic. That is, the time a time-triggered monitor needs to read the state of the program and evaluate properties is less than the sampling period. This can be, for instance, guaranteed by scheduling all verification tasks at run time on a different computing core. We now show that our monitor construction method is sound and complete with respect to observing all value changes of variables.

**Lemma 1** *Let $P$ be a program and $w = (a_0, t_0), (a_1, t_1) \cdots$ be a timed word of $\mathcal{A}_{CFG_P}||\mathcal{S}_P$. For all $i$ and $j$, where $i < j$, $a_i = a_j = s$, and there does not exist an $s$-transition between $a_i$ and $a_j$ in $w$, no run over $w$ contains delay transitions between $a_i$ and $a_j$ that includes two critical instructions.*

*Proof* The lemma holds by construction of $\mathcal{S}_P$, as it enforces sampling period $LSP$. We only describe three cases for the sake of clarity:

- Note that if all locations of $\mathcal{A}_{CFG_P}$ show their worst-case execution time, the monitor still observes all critical state changes. One can think of this scenario similar to a sliding window with fixed size (equal to $LSP$) that can move over a run. Since the window can never observe two critical state changes, worst-case executions are irrelevant to sampling points.
- The above argument also clarifies why the delay invariant of location $S_1$ in $\mathcal{S}_P$ causes no incorrectness.
- Finally, removing self-loops from non-critical vertices do not create any problems, since those loops do not contain critical instructions. Thus, no matter how many times such loops iterate, the longest sampling period guarantees correctness. ■

Lemma 1 has several consequences important to deploying and running a time-triggered monitor:

- Once a time-triggered monitor starts its execution, it can soundly re-construct the state of the inspected program, regardless of the time the time-triggered monitor started executing. This implies that even when the time-triggered monitor has execution offset from the execution start of the inspected program, the time-triggered monitor starts sampling the program correctly as soon as the it starts its execution. Thus, if a time-triggered monitor crashes and recovers, it will work correctly from the point it restarts sampling.
- If the inspected program does not exhibit best-case execution time (which is normally the case), then the program executes at a slower pace. In this case, the time-triggered monitor still samples the program with $LSP$ and this ensures that no critical instructions are overlooked. Thus, if the inspected program is later augmented by new code that does not decrease the minimum time interval between the execution of two consecutive critical instructions and hence, there is no need to change the sampling period or the time-triggered monitor structure.
- Unlike worst/average-case execution, best-case execution time analysis is a straightforward procedure. Most hardware vendors publish the best-case execution time of instructions sets based on CPU clock cycles. Thus, our method for constructing a time-triggered monitor is a conservative, but robust approach for deployment.

A valid question in the context of our approach is whether any LTL$_3$ property can be soundly verified when the time-triggered sampler is in location $S_1$. To intuitively answer this question, first, consider the following property $\varphi_1 \equiv \Diamond p$; i.e., eventually proposition $p$ holds. Since the sampler reads the state of the program after any change in the value of variables in proposition $p$ (i.e., variables in $\mathcal{V}_{\varphi_1}$), when it reaches state $S_1$, simulation of proposition $p$ on $\mathcal{M}^{\varphi_1}$ can trivially determine whether the valuation of $\varphi$ is '?' or conclusive.

On the other hand, consider LTL property $\varphi_2 \equiv (p \Rightarrow \bigcirc q)$; i.e., if $p$ holds, then proposition $q$ must hold in the next state. In this case, a time-triggered monitor cannot soundly evaluate $\varphi_2$. The reason is simple: the sampler only reads the state of the program when it wakes up. Thus, if proposition $p$ becomes true in the next program state, the sampler may wake up several states after $q$ becomes true. Hence, when the sampler wakes up, if $q$ is false, then the monitor can evaluate $\varphi_2$ to $\bot$. However, if $q$ is true, then the monitor cannot deduce whether $q$ became true in the immediate state after $p$ become true, or in some other state.

In the next theorem, we show that a time-triggered monitor is sound for verification of the fragment of LTL$_3$ which excludes the next operator (denoted LTL$_3^{-\bigcirc}$). To this end, we first set our terminology based on standard concepts. A *state* of a program is a valuation of its variables. Notice that each state of a program determines a set of propositions that hold in that state. Thus, a finite word of a program is trivially defined by a finite sequence of states of the program. Given a finite word $u$ of a program, where each letter in $u$ is read by the time-triggered monitor, we denote the complete finite word of the program by $\hat{u}$. Formally, let $u = a_0a_1a_2 \cdots a_n$ and $\hat{u} = b_0b_1b_2 \cdots b_m$. If $a_i$ and $a_{i+1}$ are two letters in $u$, where $0 \leq i \leq n$, then (1) there exist $j$ and $k$, such that $0 \leq j \leq k \leq m$, $b_j = a_i$, and $b_k = a_{i+1}$, and (2) if there exists $l$, where $j < l < k$, then state $b_l$ is not sampled by the time-triggered monitor.

**Lemma 2** *Let $p$ be a proposition in AP and $u = a_0a_1 \cdots a_n$ be a finite word of a program $P$ that is sampled by a time-triggered monitor with sampling period LSP. Let $\hat{u} = b_0b_1 \cdots b_m$. It is the case that $p \notin b_i$ and $p \in b_{i+1}$, where $0 \leq i \leq m$, if and only if there exists $j$, $0 \leq j \leq n$, such that $p \notin a_j$ and $p \in a_{j+1}$.*

*Proof* The proof follows trivially from Lemma 1. ■

Intuitively, Lemma 2 shows that if a proposition becomes true in a state, then the time-triggered monitor always detects it in the next immediate sample and vice versa.

**Lemma 3** *Let $p$ and $q$ be two propositions in AP and $u = a_0a_1 \cdots a_n$ be a finite word of a program $P$ that is sampled by a time-triggered monitor with sampling period LSP. Let $\hat{u} = b_0b_1 \cdots b_m$. It is the case that $p \in b_i$ and $q \in b_j$, where $0 \leq i \leq j \leq m$ if and only if there exists $i'$ and $j'$, $0 \leq i' \leq j' \leq n$, such that $p \in a_{i'}$ and $p \in a_{j'}$.*

*Proof* The proof follows trivially from Lemma 2. ■

Intuitively, Lemma 3 shows that the causal order of occurrence of events when detected by a time-triggered monitor is correct. Notice that in Lemmas 2 and 3 the *if* directions show soundness and the *only if* reverse directions show completeness.

**Theorem 1** *Let $P$ be a program, $\varphi$ be a property in LTL$_3^{-\bigcirc}$, and $u$ be a finite word of $P$ that is sampled by a time-triggered monitor with sampling period LSP and $\hat{u}$ be the corresponding complete program finite word. It is the case that $[u \models \varphi] = [\hat{u} \models \varphi]$.*

*Proof* Let $u = a_0a_1 \cdots a_n$ and $\hat{u} = b_0b_1 \cdots b_m$. We prove the theorem in an inductive fashion:

- Let $\varphi \equiv p$, where $p$ is an atomic proposition and $[\hat{u} \models p] = \top$. This implies that $p \in b_0$. Since the time-triggered monitor samples the program in the initial state (i.e., the switch from location $S_I$ to $S_1$ in Figure 4(b)), we have $a_0 = b_0$. Thus, the monitor can determine the truthfulness of $p \in a_0$. The same argument holds for other possible values of $[u \models p]$. Also, the same claim can be proved for any point of a finite word other than $a_0$ and $b_0$, in the same fashion by applying Lemma 2.
- For cases, where the property is of the form $\neg\varphi$ or $\varphi_1 \wedge \varphi_2$, the proof is implied by Lemma 2 and is identical to the proof of the previous case.
- Let $\varphi \equiv \varphi_1 \mathbf{U} \varphi_2$. We now show that $[u \models \varphi] = [\hat{u} \models \varphi]$. We distinguish three sub-cases:

– Let us assume that $[\hat{u} \models \varphi_1 \mathbf{U} \varphi_2] =?$. It follows that (1) there does not exist $k \leq m$, such that $[\hat{u}, k \models \varphi_2] = \top$, and (2) for all $i$, $0 \leq i \leq m$, $[\hat{u}, i \models \varphi_1] = \top$. By applying Lemmas 2 and 3, it is straightforward to see that there does not exist $l \leq n$, such that $[u, l \models \varphi_2] = \top$, and (2) for all $j$, $0 \leq j \leq n$, $[u, j \models \varphi_1] = \top$. Hence, we have $[u \models \varphi_1 \mathbf{U} \varphi_2] =?$.
– Let us assume that $[\hat{u} \models \varphi_1 \mathbf{U} \varphi_2] = \top$. It follows that (1) there exists $k \leq m$, such that $[\hat{u}, k \models \varphi_2] = \top$, and (2) for all $i$, $0 \leq i \leq k$, $[\hat{u}, i \models \varphi_1] = \top$. By applying Lemmas 2 and 3, it is straightforward to see that there exists $l \leq n$, such that $[u, l \models \varphi_2] = \top$, and (2) for all $j$, $0 \leq j \leq l$, $[u, j \models \varphi_1] = \top$. Hence, we have $[u \models \varphi_1 \mathbf{U} \varphi_2] = \top$.
– Let us assume that $[\hat{u} \models \varphi_1 \mathbf{U} \varphi_2] = \bot$. It follows that (1) there does not exist $k \leq m$, such that $[\hat{u}, k \models \varphi_2] = \top$, or (2) there exists $i$, $0 \leq i \leq m$, $[\hat{u}, i \models \varphi_1] = \bot$. By applying Lemmas 2 and 3, it is straightforward to see that there does not exist $l \leq n$, such that $[u, l \models \varphi_2] = \top$, or (2) there exists $j$, $0 \leq j \leq n$, $[u, j \models \varphi_1] = \bot$. Hence, we have $[u \models \varphi_1 \mathbf{U} \varphi_2] = \bot$. ■

Notice that in Theorem 1, the logical equivalence between $[u \models \varphi]$ and $[\hat{u} \models \varphi]$ show soundness and completeness of time-triggered runtime verification. Finally, we note that one can prove that a time-triggered monitor can soundly evaluate the 3-valued version of timed linear time temporal logic (TLTL$_3$) [7, 44] as well. We choose to focus on LTL$_3$, since reasoning about TLTL$_3$ would require presenting a fairly large background, which would distract our main goal in this paper.

## 4 Optimizing the Sampling Period and its Complexity

Employing the longest sampling period as identified in Subsection 3.1 results in highly frequent involvement of the time-triggered monitor in the program execution at run time. Obviously, increasing the sampling period naively may lead to the inability of the monitor to reconstruct the state of the program at a sampling point. Thus, in order to reduce the number of sampling points (i.e., reduce monitor involvement), we use auxiliary memory to build a history of program state changes between two samples. More specifically, let $(u, v)$ be an arc and $v$ be a vertex in a critical control-flow graph $CFG$, where $inst_v = \langle i \rangle$ and $i$ changes the value of a variable, say $a$. We apply transformation $T(CFG, v)$ introduced in Subsection 3.1 and add an instruction $i'$ : $a' \leftarrow a$, where $a'$ is an auxiliary memory location. Thus, we obtain $inst_u = inst_u.\langle i, i' \rangle$. We call this process *instrumenting transformation* and denote it by $IT(CFG, v)$. For example, in Figure 5(a), assuming that `x` is a variable in $\mathcal{V}_\varphi$, all three basic blocks are critical. Thus, the sampling period is 1 time unit. Figure 5(b) shows the CFG obtained by applying the $IT$ transformation on the shaded vertex, where the corresponding instruction is added to the previous vertex and the value of `x` is stored in `x'`.

Unlike uncritical vertices, the issue of loops involving critical vertices need to be handled differently. Suppose $u$ and $v$ are two critical vertices with arcs $(u, v)$ and $(v, u)$ between them and we intend to apply $IT$ on vertex $u$. This results in a self-loop $(v, v)$, where $w(v, v) = w(u, v) + w(v, u)$. Since we do not know how many times the loop may iterate at run time, it is impossible to determine the upperbound on the size of auxiliary memory needed to collapse vertex $v$. Hence, to ensure correctness, we do not allow applying transformation $IT$ on critical vertices that have self-loops.

Before we elaborate on the formulation of optimal instrumentation of a program, two issues need to be addressed with regard to instrumenting a program using the

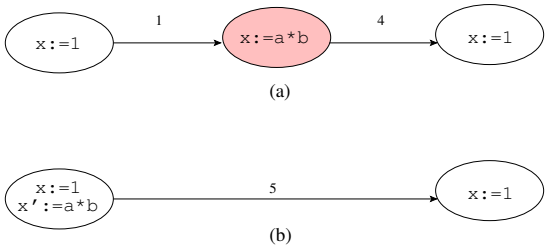



**Fig. 5** *IT* transformation applied on the middle basic block.

*IT* transformation. The first issue is whether instrumenting a program affects the calculation of the longest sampling period as described in Subsection 3.1. Observe that adding an extra instruction to a critical basic block only extends the execution time of that basic block. As argued in Section 3, if the execution of a basic block gets extended for any reason, it does not affect the correctness of a time-triggered monitor. This is due to the fact that adding instrumentation only increases the best-case execution time of a basic block and by maintaining the calculated sampling period, we are guaranteed that no critical instruction is overlooked.

The second issue is that whether the transformed program inspected by a time-triggered monitor exhibits the same semantics. In particular, let $(u, v)$ be an arc of a critical control-flow graph and $v$ be the basic block, on which transformation $IT(CFG, v)$ is applied. Let the critical instruction in $v$ be $inst_v = \langle i \rangle$, which updates a variable $a$ and the added instruction by $IT$ be $i' : a' \leftarrow a$. Let us assume that variable $a$ participates in valuation of a proposition $p$. In this setting, in order to enable a monitor to evaluate $\text{LTL}_3$ properties soundly, one only needs a simple rewriting procedure, so that a property that involves proposition $p$ is rewritten by $p'$ when variable $a'$ is read. Hence, valuation of any $\text{LTL}_3^{-\bigcirc}$ property is preserved by applying *IT* transformations; i.e., the instrumentation instructions do not change the functional properties of the inspected program (obviously with no next operator).

We now analyze the complexity of achieving optimal instrumentation. Given a critical control-flow graph, our goal is to optimize two factors through a set of *IT* transformations: (1) minimizing auxiliary memory, and (2) maximizing the sampling period. We now analyze the complexity of such optimization.

**Instance.** A critical control-flow graph $CFG = \langle V, v^0, A, w \rangle$ and positive integers $X$ and $Y$.

**Transformation optimization decision problem (TO).** Does there exist a set $U \subseteq V$, such that after applying transformation $IT(CFG, u)$ for all $u \in U$, we obtain a critical control-flow graph $CFG' = \langle V', v^0, A', w' \rangle$, where $|U| \leq Y$ and for all arcs $(u, v) \in A'$, $w'(u, v) \geq X$?

**Theorem 2** *TO is NP-complete.*

*Proof* Since showing membership to NP is straightforward, we only need to prove that TO is NP-hard. To this end, we reduce the *Minimum Vertex Cover Problem* (VC) [29] to TO. The minimum vertex cover problem is as follows: Given a (directed or undirected)

graph $G = \langle V, E \rangle$ and a positive integer $K$, the problem is to find a set $V' \subseteq V$, such that $|V'| \leq K$ and each edge in $E$ is incident to at least one vertex in $V'$.

First, we present a mapping from an instance of VC to an instance of TO. Then, we illustrate a reduction using our mapping.

**Mapping.** Let digraph $G = \langle V_1, E \rangle$ and positive integer $K$ be an arbitrary instance of VC. We obtain an instance of TO as follows:

- We construct digraph $CFG = \langle V_2, v^0, A, w \rangle$ as follows:
  - $V_2 = V_1 \cup \{v^0\}$, where $v^0$ is an additional vertex representing the initial basic block of $CFG$.
  - $A = E \cup \{(v^0, u) \mid u \in V_2\}$,
  - $w(v_0, u) = 2$ for all $u \in V_2$ and $w(v, u) = 1$, for all $v \in V - \{v^0\}$.
- Finally, we let $Y = K$ and $X = 2$.

**Reduction.** Now, we show that the answer to an instance of VC is affirmative if and only if the answer to TO is positive:

- ($\Rightarrow$) Let $V_1' \subseteq V_1$ be the answer to VC for $G$, such that $|V_1'| \leq K$. We now show that the set $V_2'$ identical to $V_1'$ is the answer to TO. First, observe that $|V_2'| \leq Y$. Now, notice that deleting a vertex in $V_2'$ results in all pairs of incoming and outgoing arcs to be replaced by edges of weight 2. The only case where an edge of weight 2 is not created between two vertices, say $u$ and $v$, is when an edge of cost 1 already exists between $u$ and $v$. However, since all arcs are covered by a vertex in $V_2'$, the arc with weight 1 will be replaced by an arc of weight at least 2 through another vertex in $V_2'$ as well. Finally, since all vertices have indegree and outdegree of at least 1, all arcs are replaced by arcs of cost at least 2.

- ($\Leftarrow$) Let $V_2' \subseteq V_2$ be the answer to TO, such that $|V_2'| \leq Y$. We now show that the set $V_1'$ identical to $V_2'$ is the answer to VC. First, observe that $|V_1'| \leq K$. Now, since the weight of all arcs in $A$ are at least 2, all edges in $E_1$ must be incident to at least one vertex in $V_1'$. This simply implies that $V_1'$ is a cover for $E_1$. ■

Obviously, time-triggered monitoring and in particular, increasing the sampling period introduces *detection latencies*. To tackle this problem, one can specify a *tolerable detection delay* for variables in $\mathcal{V}_\varphi$. This factor can be easily incorporated in our transformation technique and optimization problem.

Theorem 2 clearly shows the tradeoff between minimizing the auxiliary memory size and maximizing the sampling period. For practical reasons, designers may have restrictions over the size of the auxiliary memory for building the histories. Nevertheless, one can increase the sampling period as much as possible to bound the runtime monitoring overhead. The extreme case is to take a sample in the beginning and one at the end of program execution. We now show that building optimized history even for this overly simplified problem remains NP-complete. We denote this problem by MTO.

**Theorem 3** *MTO is NP-complete.*

*Proof* Since showing membership to NP is straightforward, we only need to prove that MTO is NP-hard. To this end, our reduction is from the *Hamiltonian Path Problem*

(HP) [29]: Given a (directed or undirected) graph $G = \langle V, E \rangle$, the problem is to determine whether $G$ has a *simple* path that visits all vertices in $V$.

First, we present a mapping from an instance of HP to an instance of MTO. Then, we illustrate a reduction using our mapping.

**Mapping.** Let digraph $G_1 = \langle V_1, E_1 \rangle$ be an arbitrary instance of HP. We obtain an instance of MTO as follows:

- We construct digraph $G_2 = \langle V_2, E_2 \rangle$, such that $V_2 = V_1$ and $E_2 = E_1$.
- The cost function is defined as $C(e) = 1$ for all $e \in E_2$
- Finally, we let $Y = |V| - 2$ and $X = |V| - 1$.

**Reduction.** Now, we show that the answer to an instance of HP is affirmative if and only if the answer to MTO is positive:

- ($\Rightarrow$) If the answer to HP is affirmative, one can delete all vertices except for the first and the last along the Hamiltonian path. It follows that the number of deleted vertices is $|V| - 2$ and since each transformation selects edges with maximum cost, the cost of the final edge is $|V| - 1$.

- ($\Leftarrow$) Suppose that the answer to MTO is affirmative. This implies that the deleted vertices must be in a total order sequence to create edges of cost $|V| - 1$. This sequence creates a path that includes all but two vertices. Moreover, this path is simple, as deleting a vertex makes it impossible to consider a vertex more than once. Finally, since the cost of edges are at $|V| - 1$, the remaining two vertices are source and terminating vertices of a Hamiltonian path.

## 5 Mapping to Integer Linear Programming

In order to cope with the exponential complexity of our optimization problem, we transform it into *Integer Linear Programming* (ILP). ILP is a well-studied optimization problem and there exist numerous efficient ILP solvers. The problem is of the form:

$$\begin{cases} \text{Minimize} \quad c.\mathbf{z} \\ \\ \text{Subject to } A.\mathbf{z} \geq \mathbf{b} \end{cases}$$

where $A$ (a rational $m \times n$ matrix), $c$ (a rational $n$-vector), and $\mathbf{b}$ (a rational $m$-vector) are given, and, $\mathbf{z}$ is an $n$-vector of integers to be determined. In other words, we try to find the minimum of a linear function over a feasible set defined by a finite number of linear constraints. It can be shown that a problem with linear equalities and inequalities can always be put in the above form, implying that this formulation is more general than it might look.

We now describe how we map the optimization problem described in Section 4 to ILP. Our mapping takes the critical control-flow graph $CFG = \langle V, v^0, A, w \rangle$ of the inspected program and a desired sampling period $SP$ as input. Our objective is to find the minimum number of vertices that must be removed from $V$ through transformation $IT$.

**Integer variables.** Our ILP model employs the following sets of variables:

1. $\mathbf{x} = \{x_v \mid v \in V\}$, where each $x_v$ is a binary integer variable: if $x_v = 1$, then vertex $v$ is removed from $V$, whereas $x_v = 0$ means that $v$ remains in $V$.
2. $\mathbf{a} = \{a_{v_{(i)}} \mid v \in V \wedge 0 < i \leq \text{outdegree of } v\}$: where each $a_{v_{(i)}}$ is an integer variable which represents the weight of a unique arc originating from vertex $v$ (i.e., no two $a_{v_{(i)}}$ represent the weight of the same outgoing arc). This variable is needed to store the new weight of an arc created by merging a sequence of arcs. For example, in Figure 3(b), initially, variable $a_{B_{2_{(1)}}} = 1$. However, if $x_{B_3} = 1$ (i.e., vertex $B_3$ is removed), then $a_{B_{2_{(1)}}} = 3$.
3. $\mathbf{y} = \{y_{v_{(i)}}, y'_{v_{(i)}} \mid v \in V \wedge 0 < i \leq \text{outdegree of } v\}$, called *choice variables*, where each $y_{v_{(i)}}$ and $y'_{v_{(i)}}$ is an integer variable. The application of this set is described later in this section.

**Constraints for the initial basic block.** Since we always want a sample at the beginning of the program to extract the initial value of variables, we add the following constraints:

$$x_{v^0} = 0 \tag{1}$$

$$a_{v^0_{(i)}} = w(v^0, u) \tag{2}$$

where $0 < i \leq$ outdegree of $v^0$ and $(v^0, u)$ is an arc in $A$. Note that for each outgoing arc from $v^0$, the ILP model will have Constraint 2.

**Constraints for arc weights and internal vertices.** Since our goal is to ensure that the weight of all arcs become at least $SP$, if there exists an arc of weight less than $SP$, then the target vertex of the arc must be removed from the graph. Thus, for every arc $(u, v) \in A$, we add the following constraint:

$$a_{u_{(i)}} + SP.x_v \geq SP \tag{3}$$

where $a_{u_{(i)}}$ represents arc $(u, v)$.

Next, we add constraints for calculating the new weights of arcs when vertices are deleted from $CFG$. We distinguish two cases:

- **Case 1:** If $x_v = 0$, for some $v \in V$, then for each $a_{v_{(i)}}$, where $0 < i \leq$ outdegree of $v$, $a_{v_{(i)}}$ represents the weight of a unique arc originating from vertex $v$.
- **Case 2:** If $x_v = 1$, then for each $a_{v_{(i)}}$, where $0 < i \leq$ outdegree of $v$, $a_{v_{(i)}} = a_{v_{(i)}} + a_{u_{(j)}}$, where $a_{u_{(j)}}$ represents the weight of the arc $(u, v) \in A$. Note that in this case, although vertex $v$ is removed, for simplicity, we use variables $a_{v_{(i)}}$ as the weight of the newly created arcs. Also note that in this case, outgoing arcs from $u$ automatically satisfy Constraint 3.

In order to make these cases mutually exclusive in ILP, we use the choice variables with the following properties:

- **Prop. 1:** The values of $y_{v_{(i)}}$ and $y'_{v_{(i)}}$ are such that one of them is zero and the other is $a_{u_{(j)}}$. This property enforces mutual exclusiveness of the above cases.
- **Prop. 2:** If $x_v = 1$, then for all $i$, $0 < i \leq$ outdegree of $v$, $y_{v_{(i)}} = a_{u_{(j)}}$ and $y'_{v_{(i)}} = 0$. On the contrary, if $x_v = 0$, then $y_{v_{(i)}} = 0$ and $y'_{v_{(i)}} = a_{u_{(j)}}$.

In order to enforce Prop. 1, we use a special data structure implemented in our ILP solver called *Special Ordered Set Type 1*, where at most one variable can take a positive value while all others must have a value of zero. The following constraints enforce Prop. 1 and 2:

$$y_{v_{(i)}} + y'_{v_{(i)}} = a_{u_{(j)}} \tag{4}$$

$$sos_1(y_{v_{(i)}}, y'_{v_{(i)}}) \tag{5}$$

$$1 \leq x_v + y'_{v_{(i)}} \leq a_{u_{(j)}} \tag{6}$$

Note that Constraints 4-6 are duplicated for all $i$, $0 < i \leq$ outdegree of $v$. The following constraints implement Cases 1 and 2, respectively. These constraints target variable $a_{v_{(i)}}$ which represents an arc $(v, v') \in A$.

$$w(v, v') + a_{u_{(j)}} - y'_{v_{(i)}} = a_{v_{(i)}} \tag{7}$$

$$y_{v_{(i)}} + w((v, v')) = a_{v_{(i)}} \tag{8}$$

For example, if $v$ is deleted (i.e., $x_v = 1$), then we have $y_{v_{(i)}} = 0$ and $y'_{v_{(i)}} = a_{u_{(j)}}$ by Constraints 4-6. Moreover, when $v$ is deleted, the weight of the newly created arc $a_{v_{(i)}}$ will be $a_u + w(v)$. This is ensured by Constraints 7 and 8. Note that Constraints 7 and 8 are duplicated for all $i$, $0 < i \leq$ outdegree of $v$.

Now, we duplicate Constraints 4-8 for each incoming arc to vertex $v$. More specifically, for arcs $(u_1, v), (u_2, v) \cdots (u_n, v)$, we instantiate Constraints 4-8 by further duplicating each variable $a_{v_{(i)}}$ to $a^{u_1}_{v_{(i)}}, a^{u_2}_{v_{(i)}} \cdots a^{u_n}_{v_{(i)}}$. We note that existence of multiple incoming arcs in a control-flow graph is due to the existence of conditional and *goto* statements in the input program. Since the depth of nested conditional statements is not normally high, we do not expect to encounter an explosion in the number of $a$-variables in our ILP model.

**Handling loops.** Recall that in Section 4, we argued that vertices with self-loops cannot be removed. Self-loops are created when we apply the *IT* transformation on vertices of a cycle in a control-flow graph. To ensure that self-loops are not removed, we add a constraint to our ILP model, such that from each cycle $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n \rightarrow v_1$, only $n - 1$ vertices can be deleted:

$$\sum_{i=1}^{n} x_{v_i} \leq n - 1 \tag{9}$$

We note that cycles can be identified when we construct *CFG* and there is no need for graph exploration to enumerate them.

**Objective function.** Finally, we state our objective function, where we aim at minimizing the set of vertices removed from *CFG*:

$$\text{Minimize} \quad \sum_{v \in V} x_v \tag{10}$$

## 6 Experimental Settings

In this section, we present our tool chain and the experimental configurations in Subsections 6.1 and 6.2, respectively.

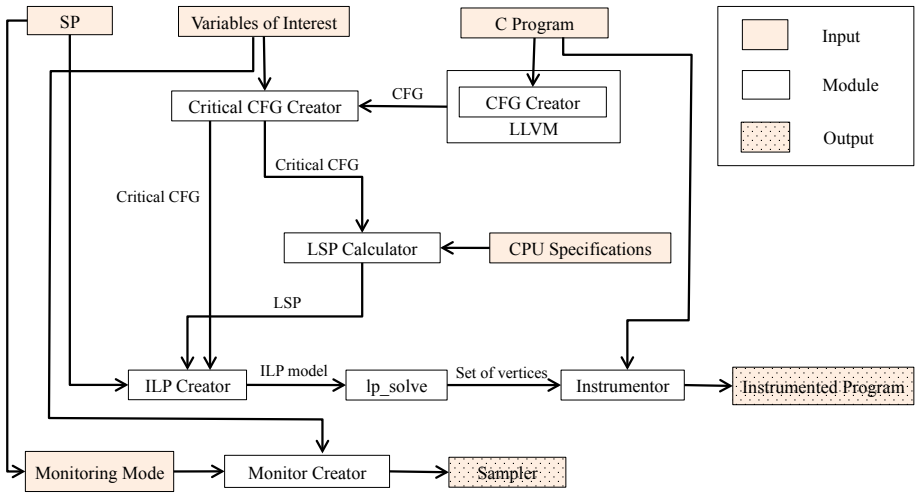



**Fig. 6** Tool chain.

### 6.1 Implementation and Tool Chain

Figure 6.1 shows our tool chain. Our tool chain consists of four main phases: (1) the *CFG* phase, (2) the *LSP* calculation phase, (3) the ILP phase, and (4) the monitoring phase. The *CFG* phase is responsible with creating the critical control-flow graph of the program. The *LSP* calculation phase finds the longest sampling period. The ILP phase is responsible for creating the ILP model and solving the optimization problem to find the minimum number of critical vertexes which need to be collapsed. The monitoring phase incorporates the sampler process of the time-triggered monitor which conducts the monitoring of the program. We note that in our experiments, we are not concerned with actual verification of LTL$_3$ properties. Our main objective is to study different aspect of monitoring overhead for data extraction only. The actual verification at runtime can be done using the tools introduced in [7].

The *CFG* phase contains two main components, the *CFG* creator and the critical *CFG* creator. The *CFG* creator is implemented over LLVM [35]. It takes the source code of the inspected program as input and produces the program's control-flow graph. Note that each vertex of the control-flow graph includes its best-case execution time and the line number of instructions incorporated within the vertex. The critical *CFG* creator is a Java application which receives the control-flow graph from the *CFG* creator and the list of variables of interest (i.e., variables in $\mathcal{V}_\varphi$) from the user. As a result, the critical *CFG* creator creates the critical control-flow graph and provides it to the *LSP* calculator phase.

The *LSP* calculator phase contains a Java application which receives the critical control-flow graph of the program from the critical *CFG* creator and calculates the longest sampling period of the program in the form of CPU cycles. The *LSP* calculator can also calculate the sampling period in the form of nano-seconds, if the user provides CPU specifications.

The ILP phase contains three main components, the ILP creator, ILP solver, and the instrumentor. The ILP creator is a Java application which receives the critical control flow graph from the critical *CFG* creator, the longest sampling period from the *LSP* calculator, and the intended sampling period *SP*. The ILP creator returns an ILP model within the format acceptable by the ILP solver. Our ILP solver is the mixed integer linear programming (MILP) solver, lp_solve [37]. lp_solve receives the ILP model and solves the ILP problem. As a result lp_solve returns the set of critical vertices which need to be collapsed. The instrumentor receives the set of collapsed critical vertices from lp_solve and finds the instructions in the program's source code that are incorporated within the collapsed critical vertices. Then, the instrumentor creates a duplicate of the program's source code and instruments the copied source code appropriately. In other words, after each instruction within a collapsed vertex, the instrumentor adds an instruction which stores the variable of interest updated by the instruction within the history. In the end, the instrumentor returns an instrumented copy of the program's source code.

The monitoring phase contains a Java application (monitor creator) which creates the sampler process. The sampler process is a C program which runs in parallel with the inspected program. The monitoring phase can create one of the following three monitoring modes for the sampler process:

- **Event-triggered (*ET* monitor):** The program execution halts when an instruction changes the value of a variable of interest and invokes the sampler. The sampler reads the new value of the variable of interest and informs the program to resume execution.
- **Time-triggered with no history (*TT* monitor):** The sampler is time-triggered. The sampler has a timer that represents the time-triggered monitor's sampling period. Hence, the sampler sets this timer to the sampling period calculated by the *LSP* calculator. When the timer goes off, the sampler halts the execution of the program and reads the value of all variables of interest. Then, the sampler resets its timer and informs the program to resume execution.
- **Time-triggered with history (*TTH* monitor):** This setting incorporates our ILP optimization. The monitoring is performed over the instrumented copy of the source code that is provided by the instrumentor. The sampler functions as of the *TT* monitor. In this setting, the sampler sets its timer to the intended sampling period (i.e., SP) provided by the user, and it also extracts the data in the history, in addition to the variables of interest.

In addition, the monitor creator receives the variables of interest and provides read access for these variables to the sampler process.

### 6.2 Experimental Configurations

Our case studies are from the SNU [1] benchmark suite. The experimental setting is as follows. In each program, the main function runs 500 times, where at each iteration the main function receives new input values from the environment. For each program, we find the top two variables which play the most part in the program's runtime behavior. In other words, we find the two variables which are most used by the program's instructions. Then, we consider these two variables as the variables of interest (i.e.,

variables in $\mathcal{V}_\varphi$). The program and the time-triggered monitor run on an MCB1700 board with the RTX real-time operating system.

To evaluate the $TT$ and $TTH$ monitors, we consider the following metrics:

1. The execution time of the monitored program. This value projects the amount of monitoring overhead at run time.
2. The absolute jitter (i.e., the difference between the minimum and maximum value) of the overhead of the monitor invocations throughout the program run. This metric is of importance, since when the absolute jitter of the overhead of the monitor invocations is smaller, the monitor has more predictable behavior.
3. The amount of memory used by the $TTH$ monitor.

## 7 Experimental Results

In this section, we describe the results of our Experiments. In particular, we analyze the runtime overhead of different monitors (i.e., $ET$, $TT$, $TTH$) in Subsection 7.1. Then, Subsection 7.2 studies the impact of employing different monitors on memory usage and history size. Finally, in Subsection 7.3, we present evidence that shows employing a time-triggered monitor results in more efficient allocation of resources at runtime.

### 7.1 Monitoring Overhead at Run Time

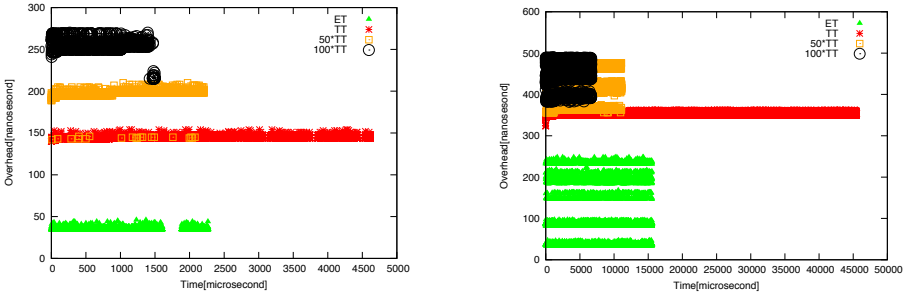
We analyze the runtime monitoring overhead based on the absolute jitter of the monitor invocations (cf. Subsection 7.1.1), the actual execution time of the monitored program, and redundant sampling (cf. Subsubsection 7.1.2).

#### *7.1.1 Absolute Jitter Analysis*

Notice that each monitor invocation overhead ($MIO$) at run time is caused by:

1. the overhead of interrupting/resuming the program execution and invoking the monitor, and
2. reading the values of the variables of interest as well as the variables stored in the history.

Recall that one of the main goals of designing a time-triggered monitoring is to obtain *bounded monitoring overhead* at run time. In other words, we hypothesize that the absolute jitter of MIO of a $TT$ monitor is less than the absolute jitter of MIO of an $ET$ monitor. To validate our hypothesis, consider Figures 7(a) and 7(b). These figures show MIO of the three monitoring modes throughout the program run of bs and qsort, respectively. In general, the $ET$ monitor invocations are irregularly distributed throughout the program execution. For instance, Figure 7(a) shows that program bs has an execution path which does not incorporate any critical instructions, and hence, the $ET$ monitor is not invoked at all (the execution between 1500$\mu$s and 2000$\mu$s). Moreover, recall that the overhead caused by each invocation is proportional to the type of the variable of interest read by the monitor. Hence, MIO may vary considerably from one invocation to another when the type of variables of interest vary. This, in turn, results in a large absolute jitter for the MIO (e.g., qsort in Figure 7(b)). Thus, the $ET$ monitor




(a) Monitor invocation overhead in bs

(b) Monitor invocation overhead in qsort

**Fig. 7** Monitoring invocation absolute overhead.

introduces probe-effects, which in turn may create unpredictable and even incorrect behavior from the monitored program. This anomaly is, in particular, unacceptable for real-time embedded and mission-critical systems.

On the contrary, the *TT* monitor invocations are evenly distributed throughout the program execution. In addition, since the number and type of variables of interest read at each sample remains constant, MIO is not subject to any bursts. Recall that the *TT* monitor reads all the variables of interest at each sample. Hence, the absolute overhead remains consistent and bounded which results in a small absolute jitter for MIO (see Figures 7(b) and 7(a)). Consequently, the monitored program exhibits a predictable behaviour. As can be seen in Figure 7, the *TT* monitor may potentially impose larger overhead compared to the *ET* monitor, which as a result, extends the overall execution time of the monitored program. Nonetheless, in many commonly considered embedded applications, designers prefer predictability at the cost of larger overhead.

For qsort, Figure 7(b) shows that the absolute jitter of MIO of the *TT* monitor is less than the absolute jitter of MIO of the *ET* monitor. In qsort, the variables of interest are of types array of int, int, and long. The different values of MIO of the *ET* monitor in Figure 7(b) shows how different types of variables can affect the value of MIO. For instance, when an entry in the array changes the *ET* monitor reads all the variables stored within the array. On the contrary, when the long variable changes, the *ET* monitor only reads the changed variable. Hence, the overhead of reading variables of interest may vary significantly from one variable to another. Thus, Figure 7(b) is a clear indication of how different types of variables of interest affect the value of MIO. As for bs, Figure 7(a) shows that the absolute jitter of MIO of the *TT* monitor and the absolute jitter of MIO of the *ET* monitor are approximately equal. For such programs, this is caused by the condition, where all variables of interest are of the same type. For instance in bs, all the variables of interest are of type int, and hence, the *ET* monitor extracts the same type of variables at each invocation. Thus, the *ET* monitor simulates the invocation condition of a *TT* monitor, where the type and number of variables extracted by the monitor are all similar. As a result, the absolute jitter of MIO of the *ET* monitor is similar to the absolute jitter of MIO of the *TT* monitor.

The situation is more complex for *TTH* monitors. A *TTH* monitor also reads the variables of interest stored in the history at each sample. Hence, the MIO of a *TTH* monitor depends on the overhead of reading the history as well. Since, the number and type of variables stored in the history can differ from one sample to another, the

absolute jitter of MIO may not be smaller to the absolute jitter of MIO of the $ET$ monitor. For qsort, the absolute jitter of MIO of the $TTH$ monitor with the intended sampling period of both $50*TT$ and $100*TT$ is less than the the absolute jitter of MIO of the $ET$ monitor (see Figure 8(a)). This is not the case in all programs, for instance, in program jfdctint, the absolute jitter of MIO of the $TTH$ monitor with the intended sampling period of both $50*TT$ and $100*TT$ is greater than the absolute jitter of MIO of the $ET$ monitor. Our deeper analysis shows that this is caused by a large difference between the number of *instrumentation* instructions executed between two samples. Recall from Section 5 that we use ILP to find the set of critical instructions to collapse as to achieve the intended sampling period $SP$, and as a result, the variables updated by these instructions are stored within the history. Hence, immediately after each of these critical instructions, the Instrumentor module in our tool chain adds an instruction which stores such updated variables into the history. We refer to these instructions as *instrumentation* instructions. Since our ILP model only focuses on finding the solution which incorporates the minimum amount of history, the instrumentation instructions may be unevenly distributed throughout the program run. Thus, there is a possibility that the number of executed instrumentations between two samples vary.

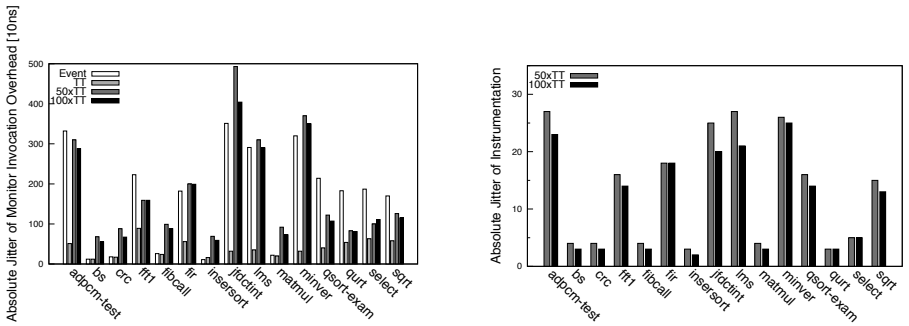
The fluctuation in the number of instrumentations executed between two samples causes an increase in the absolute jitter of the MIO. Figure 8(b) shows the absolute jitter of the executed instrumentations between two consecutive samples. It is clear that a larger absolute jitter for the executed instrumentation causes a larger absolute jitter for MIO. As can be seen, for jfdctint, the absolute jitter of the executed instrumentation to achieve the intended sampling period of $50 * TT$, is 25, which is a large value in comparison to the other SNU programs.

Figures 7(b) and 7(a) also show that in each monitoring mode, the execution time of the program changes. Note that the execution time of a monitored program depends on (1) the MIO of the monitor, and (2) the number of monitor invocations. Recall that at each monitor invocation, the sampler process stops the program execution and resumes its execution when the sampler finishes reading the variables of interest. Hence, the time the program *resumes* its execution depends on the MIO. For instance, the MIO of the $TT$ monitor is larger than the $ET$ monitor. In addition, the $TT$ monitor intervenes with the program more often. As a result, the execution time of qsort and bs monitored with the $TT$ monitor is longer than their execution time when monitored with the $ET$ monitor. We will discuss the characteristics of a monitored program's execution time in more detail in the next Subsection.

#### *7.1.2 Execution time and Redundant Sampling Analysis*

As for the affect of the monitoring overhead on program execution, Figures 9(a) and 9(b) show the execution time of the programs of SNU while being monitored with our three monitoring modes. The results show that the execution time of a program monitored with the $TT$ monitor is larger than the execution time of the program monitored with the $ET$ monitor. This excessive overhead is caused by the following characteristics of the $TT$ monitor.

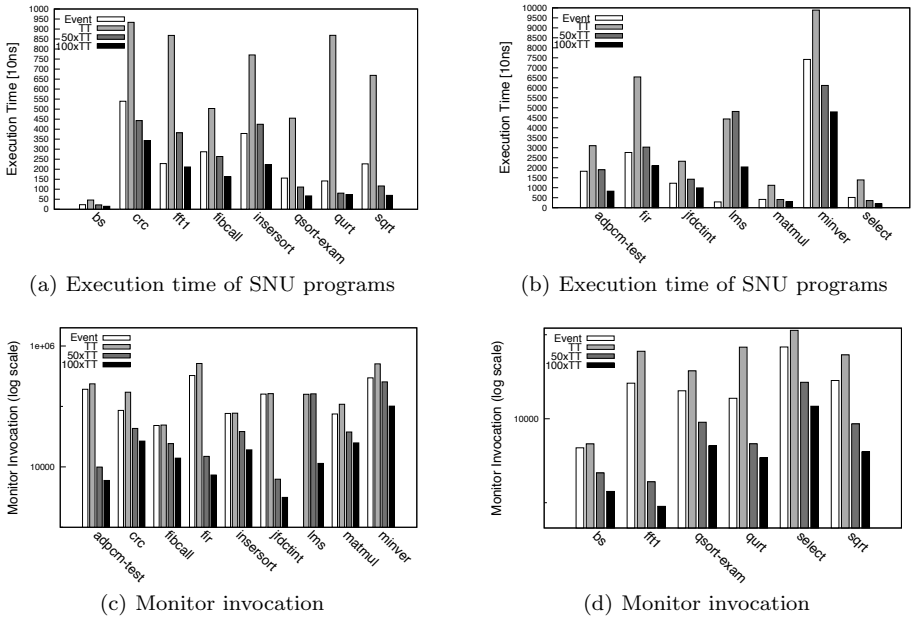
- The monitor invocation happens more often in the $TT$ monitor compared to the $ET$ monitor.
- The MIO of the $TT$ monitor is larger compared to the $ET$ monitor. This is caused by the fact that at each invocation, the $TT$ monitor reads all the variables of interest from the program while the $ET$ monitor reads only one variable of interest.




(a) Absolute jitter of monitor invocation overhead

(b) Absolute jitter of instrumentation

**Fig. 8** Absolute jitter of monitor invocation overhead and instrumentation.




(a) Execution time of SNU programs

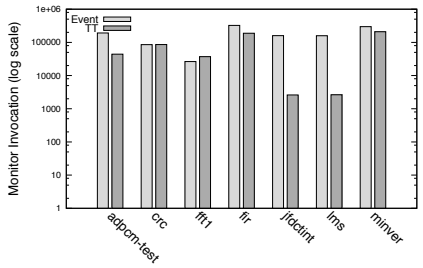
(b) Execution time of SNU programs

(c) Monitor invocation

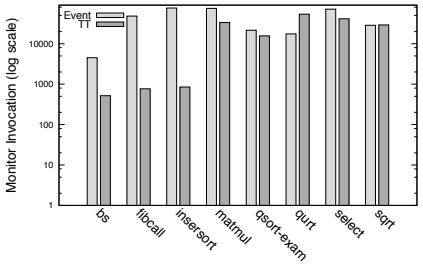
(d) Monitor invocation

**Fig. 9** Monitoring overhead and monitoring invocation.

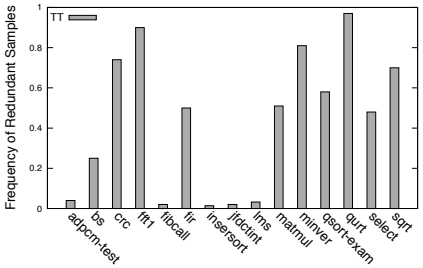
The side affect of high volume monitor invocation is *redundant sampling*. A redundant sample is when the monitor takes a sample while the program has not executed a critical instruction since the last monitor sample. The $TT$ bar in Figures 10(a) and 10(b) shows the number of redundant samples taken by the $TT$ monitor at run time, and the *Event* bar shows the number of critical instructions executed at run time. The ratio of redundant samples to the number of executed critical instructions is the metric which defines the excessive overhead of the $TT$ monitor. We refer to this ratio as the *redundant sample ratio*. Thus, a larger redundant sample ratio results in larger excessive monitoring overhead, and hence, a longer execution time for the monitored program.




(a) Redundant samples of $TT$ monitor




(b) Redundant samples of $TT$ monitor




(c) Frequency of redundant samples

**Fig. 10** Redundant samples and their frequency.

In a program such as minevr, the monitor takes 209213 number of redundant samples which results in a redundant sample ratio of 0.70, which is a considerably large value compared to the other SNU programs. Consequently, the execution time of minevr when monitored with the $TT$ monitor is 1.33 times longer than its execution time when monitored with the $ET$ monitor (see Figure 9(b)). Figure 10(c) shows the average frequency in which the monitor takes a redundant sample. Clearly, a larger redundant sample ratio results in a higher frequency of redundant samples. Hence, it is desirable to decrease the frequency of redundant samples. To this end, we use history to increase the sampling period, and hence, decrease the frequency of redundant samples. In the SNU programs, by using history to achieve the intended sampling periods of $50 * TT$ and $100 * TT$, the number of redundant samples reduces to zero. In other words, in the SNU programs, the intended sampling periods of $50 * TT$ and $100 * TT$ do not result in redundant samples, meaning that when the $TT$ monitor takes a sample, the program has executed at least one critical instruction. Note that this may not be the case for programs other than the SNU programs. In other words, by using history, the redundant samples of the $TT$ monitor does not reduce to zero for all possible programs.

In general, the monitoring overhead imposed by the $TTH$ monitor is less than the $TT$ monitor. Recall that the MIO is twofold. Our studies show that the overhead imposed by stopping/resuming the program execution and invoking the monitor makes up the majority of MIO. Consequently, when the $TTH$ monitor increases the sampling period, it also reduces the number of monitor invocations (i.e., samples). As a result, the overhead imposed by the $TTH$ monitor is less than the $TT$ monitor. Figures 9(a) and 9(b) show the reduction in the execution time of the programs when using the $TTH$ monitor compared to using the $TT$ monitor.

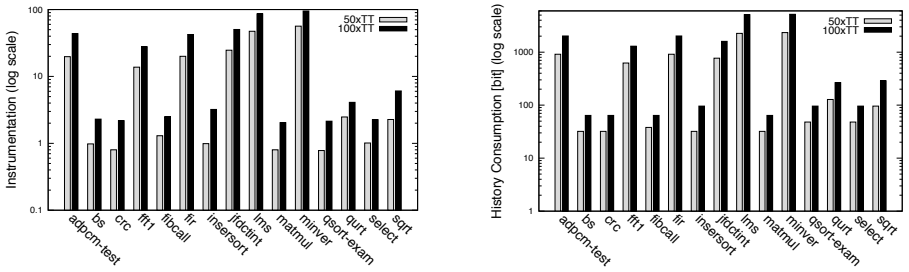
In some programs such as sqrt, select, and qurt, the execution times of the programs monitored with the $TTH$ monitor with sampling period of $50 * TT$ and $100 * TT$ are less than the execution time of the programs when monitored with the $ET$ monitor. Our studies show that in such programs, by using history, the number of monitor invocations reduces by more than 50% (e.g., in sqrt the reduction is 93% ). Figures 9(c) and 9(d) show the number of monitor invocations for each monitoring mode. These figures show that the number of monitor invocations of the $TTH$ monitor for both $50 * TT$ and $100 * TT$ are less than the monitor invocations of the $ET$ monitor. These figures show that the $TTH$ monitor does not introduce redundant samples, and hence, does not impose excessive and redundant overhead. Also, our studies show that in these programs, the overhead of stopping/resuming the program execution and invoking the monitor still makes up the majority of the MIO. Hence, the effect of the reduction in the number of invocations overcomes the effect of the increase in the overhead of reading the values of the variables of interest (Recall that the $TTH$ monitor must also read the variables of interest stored in the history). Thus, the monitoring overhead imposed by the $TTH$ monitor becomes less than the overhead of the $ET$ monitor.

On the other hand, in programs such as lms and insertsort, the execution time of the programs monitored with the $TTH$ monitor with sampling period of $50 * TT$ is more than their execution time when monitored with the $ET$ monitor. In these programs, our studies show that the overhead of reading the variables of interest exceeds the overhead of stopping/resuming the program execution and invoking the monitor. Hence, the effect of the increase in the overhead of reading the values of the variables of interest overcomes the effect of the decrease in the number of monitor invocations. Thus, the overhead imposed by the $TTH$ monitor is larger than the overhead of the $ET$ monitor.

### 7.2 History Size at Run Time

Regarding the $TTH$ monitor, recall that we prohibited deletion of self-loops from critical control-flow graphs. Hence, if some critical instructions reside in loop structures, the minimum sampling period of the loop structures, can determine the longest sampling period. For the SNU programs, the majority of the critical instructions reside in loops, and hence, in such a situation, employing history does not result in a considerable increase in the sampling period (e.g., for ficall the sampling period does not increase at all). To overcome this problem, we use profiling to estimate the upper bound of the number of times each loop structure takes for each program. We leverage gcov to carry out the profiling. With respect to the upper bound on the loops and type of variables of interest updated within the loops, we devise a size for the memory location of the history. For instance, if a loop structure runs at most 100 times and within it a variable of interest of type integer is updated, we devise a memory location of at least the size $int\_size * 100$ for the history. In addition, we note that solving the corresponding ILP problem for all programs of SNU take an average of 56 seconds. This clearly shows that we are not even close to the boundaries of ILP solving.

Figure 11(a) shows the average number of instrumentation executed between two consecutive samples. In other words, it shows the average number of data stored in the history between samples. Figure 11(b) shows the average amount of memory consumed by the history in between two consecutive samples. Note that the amount of history consumption depends on the number and type of variables stored within the history. The encouraging outcome from the experimental results shown in Figure 11(b) is that




(a) Number of variables stored in the history (b) Amount of history consumption

**Fig. 11** Number of variables stored in history and memory consumption.

with a small amount of additional memory, we can severely increase (e.g., by 50 and 100 times) the sampling period of the $TT$ monitor. For instance, program lms uses the most amount of extra memory (5088 bits) to increase its sampling period by a factor of 100. Hence, results show that by using approximately 5kbits of memory, the execution time of lms decreases by 57% (see Figure 9(b)). Thus, the experimental results encourage the use of $TTH$ monitors.

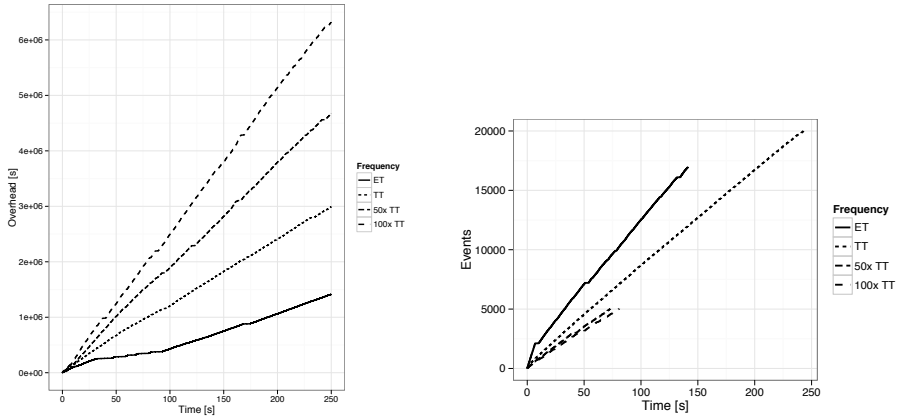
### 7.3 Resource Management

Although the relative data may seem to indicate that event-triggered approaches use less resources than time-triggered approaches, this is incorrect. Real-time applications must operate even under worst-case scenarios and as such, the worst-case behavior is of interested instead of the average case behavior.

To demonstrate this, let us consider the program sqrt and the associated measurements. Figure 12(a) shows the cumulative overhead for the program. The $x$-axis show the execution time of the application in seconds and the $y$-axis shows the overhead up to that execution time. It clearly shows that the event-triggered system has less overhead than the time-triggered approaches. However, this also depends on the worst-case behaviour.

Figure 12(b) shows the upper bound on the number of events for different durations. The $x$-axis shows the length of the duration in which events occurred. The $y$-axis shows the maximum number of events found in at least one time interval of the given duration. For example, the time-triggered approach resulted in an upper bound of 5000 events in at least one observed interval of 250 seconds. The interesting point is that the event-triggered approach consistently has a higher upper bound on the number of events for any given duration. This means that for a real-time application, in which developers have to reserve resource budgets to ensure the timeliness of the system, the event-triggered approach will require a larger reservation of resources than time-triggered approaches. This is due to, in the worst case, more events occurring and the need to reserve resources for the worst case.

The final decision on which scheme requires the lower resource partition depends on several factors such as interruption overhead, context-switch overhead, messaging overhead, etc. Thus Figure 12(b) reports the value on the $y$-axis as the number of observed events. However, this leaves the basic argument intact.




(a) Cumulative overhead measured for sqrt (b) Bound on monitoring events for sqrt

**Fig. 12** Resource management

## 8 Related Work

In classic runtime verification [43], a system is composed with an external observer, called the monitor. This monitor is normally an automaton synthesized from a set of properties under which the system is scrutinized. In general, runtime verification frameworks [15, 22, 24, 31, 32, 36, 46] use event-triggered monitoring, in the sense that every change in the state of the program invokes the monitor for analysis. These frameworks mainly consist of three components:

- The *logic* component converts a logical property into an internal representation.
- Using the internal representation, the *data extraction* component determines the data required for property evaluation and instruments the program accordingly.
- The *verification* component evaluates the property, using the property's internal representation and the data provided by the instrumentation at run time.

In these frameworks, each instrumentation added to the program, invokes the verification component and provides the component with data that reflect the changes in the program state (i.e., data required for property evaluation). Hence, these frameworks are unsuitable for time-sensitive systems because of their event-based monitoring.

On the other hand, our initial work [12] and the approach presented in this article use time-triggered monitoring. In particular, the approach in [12], calculates the longest sampling period which ensures sound program state reconstruction. This method may impose a large monitoring overhead. Hence, in [12] and [40], we propose using auxiliary memory to increase the sampling period, which in turn reduces the runtime monitoring overhead. In the same context, [20] introduces a sampling-based program monitoring technique. It proposes a framework that allows quantitative reasoning about issues involved in sampling-based techniques. [20] also discusses how to optimally instrument a program using a set of *markers*, such that different execution paths reachable from the same state are distinguishable. In addition, [47] also uses a sampling-based program monitoring technique to monitor the program execution. [47] introduces state estimation using Hidden Markov Models (HMM) to estimate the probability of satis-

faction/violation of a property in between samples. This method may suffer from false positives and false negatives.

From the logical and language point of view (i.e., logical component), runtime verification has mostly been studied in the context of Linear Temporal Logic (LTL) properties [5, 21, 24–26, 48] and in particular safety properties [27, 45]. Other languages along with their logical components have also been developed for facilitating specification of temporal properties [30, 33, 49]. Runtime verification of $\omega$-languages was considered in [17]. In addition, [19] addresses runtime verification of safety-progress [13, 38] properties. Also, [41] proposes the language Copilot for developing hard real-time monitors. The aim of this language is to develop programs where the monitor (1) does not change the functionality and schedule of the program, and (2) adds minimal overhead to the program. We, however, take a different approach by focusing on developing a new data-extraction component. We design a method where predictable monitors are added to observe the behaviour of existing programs. We also present optimization techniques and experimental evidence on the effectiveness of our approach.

Regardless of the type of monitor, runtime verification frameworks must impose low monitoring overhead to be considered practical. [4,27] reduce the overhead by rewriting safety properties such that the evaluation of properties (i.e., the verification component) requires the least information regarding the state of the program at run time. [10,11] reduce the number of instrumentation added by the data-extraction component, by determining locations in the program which do not affect property evaluation. [9] distributes the instrumentation cost added by the data-extraction component, among multiple users. Consequently, each copy of the program only extracts a subset of the data required to evaluate a property. [9] uses a central server to collect the partial data and provides the complete set of data to the verification component for property evaluation. [28] controls the overhead imposed by the framework by temporarily disabling monitoring of selected data, with the use of supervisory control theory of discrete event systems and PID-control theory of discrete time systems. [18] extracts only a subset of the data required to evaluate program properties, by removing/adding instrumentation relevant to the program state at run time. [3] discards instrumentation added by the data-extraction component with respect to the execution path of the program at run time. [3] manually predicts the execution path that the program will take at run time with respect to the inputs given to the program. Regarding the predicted path, [3] only keeps the instrumentation required for property evaluation.

## 9 Conclusion

In this article, we proposed a time-triggered approach for runtime verification. In this technique, a monitor interrupts the program execution within fixed time intervals to inspect the health of the program. We explored the problem by defining it in formal terms and then showed that a time-triggered monitor can soundly verify an $\text{LTL}_3$ [7] property (with no next operator) at run time. We also formulated an optimization problem for using minimum auxiliary memory to build a history of events occurred and maximize the sampling period. We showed that this problem is NP-complete. As a practical solution to cope with the complexity, we encoded our problem in integer linear programming (ILP). Our approach is fully implemented in the RiTHM[3] tool chain that

[3] To access the tool, please visit `http://uwaterloo.ca/embedded-software-group/projects/rithm`.

takes a C program as input and (1) constructs a time-triggered monitor with an optimal sampling period, and (2) instruments the input program in order to build a history of optimal size. Experimental results show that time-triggered monitoring provides a predictive overhead on the system. Moreover, using negligible auxiliary memory, one can increase the sampling period, which results in less overall overhead and faster execution of the system under scrutiny.

For future work, we are considering several research directions. We are currently working on adaptive monitoring, where the monitor adjusts its sampling period based upon the structure of the input program or the property under inspection. Such adaptive sampling will be highly beneficial to overcome loop problems. Another interesting direction is to develop efficient polynomial-time heuristics and approximation algorithms to solve our optimization problem. Also, one may consider developing hybrid monitors that take advantage of both event-triggered as well as time-triggered techniques.

## 10 Acknowledgement

This research was supported in part by NSERC Discovery Grant 418396-2012, NSERC DG 357121-2008, ORF-RE03-045, ORF-RE04-036, ORF-RE04- 039, APCPJ 386797-09, CFI 20314 and CMC, STPGP-430575, and the industrial partners associated with these projects.

## References

1. SNU Real-Time Benchmarks. `http://www.cprover.org/goto-cc/examples/snu.html`.
2. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
3. C. Artho, D. Drusinksy, A. Goldberg, K. H. andM. Lowry, C. Pasareanu, G. Roşu, and W. Visser. Experiments with test case generation and runtime analysis. In *Proceedings of the 10th International Conference on Advances in Theory and Practice of Abstract State Machines*, ASM'03, pages 87–108, 2003.
4. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'04, pages 44–57, 2004.
5. A. Bauer, M. Leucker, and C. Schallhart. Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2009. in press.
6. A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL Semantics for Runtime Verification. *Journal of Logic and Computation*, 20(3):651–674, 2010.
7. A. Bauer, M. Leucker, and C. Schallhart. Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14, 2011.
8. E. Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *nternational Conference on Software Engineering (ICSE)*, pages 5–14, 2010.
9. E. Bodden, L. Hendren, P. Lam, O. Lhoták, and N. Naeem. Collaborative runtime verification with tracematches. In *Proceedings of the 7th International Conference on Runtime Verification*, RV'07, pages 22–37, 2007.
10. E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, ECOOP'07, pages 525–549, 2007.
11. E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE'08, pages 36–47, 2008.
12. B. Bonakdarpour, S. Navabpour, and S. Fischmeister. Sampling-based runtime verification. In *Formal Methods (FM)*, pages 88–102, 2011.

13. E. Y. Chang, Z. Manna, and A. Pnueli. Characterization of Temporal Property Classes. In *Automata, Languages and Programming (ICALP)*, pages 474–486, 1992.
14. F. Chen and G. Roşu. Java-MOP: A monitoring oriented programming environment for java. In *Tools and Algorithms for the construction and analysis of systems (TACAS)*, pages 546–550, 2005.
15. F. Chen and G. Roşu. Java-mop: A monitoring oriented programming environment for java. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 546–550, 2005.
16. S. Colin and L. Mariani. *Run-Time Verification*, chapter 18. Springer-Verlag LNCS 3472, 2005.
17. M. d'Amorim and G. Rosu. Efficient Monitoring of omega-Languages. In *Computer Aided Verification (CAV)*, pages 364–378, 2005.
18. M. B. Dwyer, A. Kinneer, and S. Elbaum. Adaptive online program analysis. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 220–229, 2007.
19. Y. Falcone, J.-C. Fernandez, and L. Mounier. Runtime Verification of Safety-Progress Properties. In *Runtime Verification (RV)*, pages 40–59, 2009.
20. S. Fischmeister and Y. Ba. Sampling-based Program Execution Monitoring. In *ACM International conference on Languages, compilers, and tools for embedded systems (LCTES)*, pages 133–142, 2010.
21. D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Automated Software Engineering (ASE)*, pages 412–416, 2001.
22. K. Havelund. Runtime verification of c programs. In *Proceedings of the 20th IFIP TC 6/WG 6.1 International Conference on Testing of Software and Communicating Systems: 8th International Workshop*, TestCom '08 / FATES '08, 2008.
23. K. Havelund and A. Goldberg. Verify your Runs. pages 374–383, 2008.
24. K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. *Electronic Notes in Theoretical. Computer Science*, 55(2), 2001.
25. K. Havelund and G. Rosu. Monitoring Programs Using Rewriting. In *Automated Software Engineering (ASE)*, pages 135–143, 2001.
26. K. Havelund and G. Rosu. Synthesizing Monitors for Safety Properties. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 342–356, 2002.
27. K. Havelund and G. Rosu. Efficient Monitoring of Safety Properties. *Software Tools and Technology Transfer (STTT)*, 6(2):158–173, 2004.
28. X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S. A. Smolka, S. D. Stoller, and E. Zadok. Software monitoring with controllable overhead. *Software tools for technology transfer (STTT)*, 14(3):327–347, 2012.
29. R. M. Karp. Reducibility Among Combinatorial Problems. In *Symposium on Complexity of Computer Computations*, pages 85–103, 1972.
30. M. Kim, I. Lee, U. Sammapun, J. Shin, and O. Sokolsky. Monitoring, Checking, and Steering of Real-Time Systems. *Electronic. Notes in Theoretical Computer Science*, 70(4), 2002.
31. M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 114–122, 1999.
32. M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-mac: A run-time assurance approach for java programs. *Form. Methods Syst. Des.*, 24(2):129–155, 2004.
33. M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A Run-Time Assurance Approach for Java Programs. *Formal Methods in System Design (FMSD)*, 24(2):129–155, 2004.
34. O. Kupferman and M. Y. Vardi. Model Checking of Safety Properties. In *Computer Aided Verification (CAV)*, pages 172–183, 1999.
35. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization: Feedback Directed and Runtime Optimization*, page 75, 2004.
36. I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *Parallel and Distributed Processing Techniques and Applications(PDPTA)*, pages 279–287, 1999.
37. ILP solver lp_solve. `http://lpsolve.sourceforge.net/5.5/`.

38. Z. Manna and A. Pnueli. A Hierarchy of Temporal Properties. In *Principles of Distributed Computing (PODC)*, pages 377–410, 1990.
39. P. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. *Springer Journal of Automated Software Engineering*, 17(2):149–180, June 2010.
40. S. Navabpour, C. W. Wu, B. Bonakdarpour, and S. Fischmeister. Efficient techniques for near-optimal instrumentation in time-triggered runtime verification. In *International Conference on Runtime Verification (RV)*, pages 208–222, 2011.
41. L. Pike, A. Goodloe, R. Morisset, and S. Niller. Copilot: A Hard Real-Time Runtime Monitor. In *Runtime Verification (RV)*, 2010. 345-359.
42. A. Pnueli. The temporal logic of programs. In *Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, 1977.
43. A. Pnueli and A. Zaks. PSL Model Checking and Run-Time Verification via Testers. In *Symposium on Formal Methods (FM)*, pages 573–586, 2006.
44. J.-F. Raskin and P.-Y. Schobbens. The logic of event clocks - decidability, complexity and expressiveness. *Journal of Automata, Languages and Combinatorics*, 4(3):247–286, 1999.
45. G. Rosu, F. Chen, and T. Ball. Synthesizing Monitors for Safety Properties: This Time with Calls and Returns. In *Runtime Verification (RV)*, pages 51–68, 2008.
46. J. Seyster, K. Dixit, X. Huang, R. Grosu, K. Havelund, S. A. Smolka, S. D. Stoller, and E. Zadok. Aspect-oriented instrumentation with GCC. In *Runtime Verification (RV)*, pages 405–420, 2010.
47. S. D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. A. Smolka, and E. Zadok. Runtime verification with state estimation. In *Runtime Verification (RV)*, pages 193–207, 2011.
48. V. Stolz and E. Bodden. Temporal Assertions using Aspectj. *Electronic Notes in Theoretical Computer Science*, 144(4), 2006.
49. W. Zhou, O. Sokolsky, B. T. Loo, and I. Lee. *MaC*: Distributed Monitoring and Checking. In *Runtime Verification (RV)*, pages 184–201, 2009.