Sampling-based Runtime Verification*

Borzoo Bonakdarpour, Samaneh Navabpour, and Sebastian Fischmeister

Department of Electrical and Computer Engineering
University of Waterloo
200 University Avenue West
Waterloo, Ontario, Canada, N2L 3G1
Email:{borzoo,snavabpo,sfischme}@ece.uwaterloo.ca

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 by every change in the state of the system and evaluates properties of the system. This constant invocation introduces two major defects to the system under scrutiny at runtime: (1) significant *overhead*, and (2) *unpredictability*. To circumvent the latter defect, in this paper, we introduce a novel *time-triggered* approach, where the monitor frequently takes samples from the system in order to analyze the system's health. We propose formal semantics of sampling-based 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 real-world applications show that our approach introduces bounded overhead and effectively reduces involvement of the monitor at runtime using negligible auxiliary memory. We also show that in some cases it is even possible to reduce the overall overhead of runtime verification using our sampling-based approach when the structure of the system allows choosing long enough sampling period.

Keywords: Runtime verification, monitoring, time-triggered, predictability.

1 Introduction

Runtime verification [2, 3, 5, 10, 12, 26] is a complementary technique to exhaustive verification methods such as model checking and theorem proving, as well as incomplete solutions such as testing. Roughly speaking, in runtime verification, the objective is to ensure that at runtime, a system satisfies its desirable properties; i.e., the system under inspection is observed and analyzed by a decision procedure called the *monitor*.

* This is an extended version of the paper appeared in the 17th International Symposium on Formal Methods (FM'11).

In the literature of runtime verification, constructing a monitor involves synthesizing an automaton that realizes the properties that the system under scrutiny must satisfy [21]. 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 every event (e.g., change of value of a variable) triggered in the system. We call this type of monitoring *event-triggered*. The main drawback of event-triggered runtime verification is twofold: the monitor (1) imposes *unpredictable overhead*, and (2) may introduce *bursts* of interruptions to the system at runtime. This can lead to undesirable transient overload situations in time-sensitive systems.

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 *constant* frequency and takes samples from the system in order to analyze the system's soundness. This way, the involvement of the monitor is time-bounded and predictable. However, the main challenge in this mechanism is accurate reconstruction of the system's state between two samples; i.e., if the value of a variable of interest 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 re-constructivity.

We calculate the sampling period through building the system's control-flow graph. Then, we employ this sampling period to define the formal semantics of sampling-based runtime verification using the timed automata formalism. The sampling period extracted from control-flow graphs tend to be short and, hence, precipitates highly frequent involvement of the monitor even in branches of the program that does not require monitoring. 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. Obviously, we face a tradeoff between minimizing the size of auxiliary memory 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). We have developed a tool chain that takes C programs as input, instruments the program to build optimal history and constructs a monitor that takes samples with the optimal sampling period. Our experimental results show encouraging results. Firstly, the size of ILP models for real-world applications are quite manageable. Secondly, we observe that in event-triggered implementations, the system suffers from bursts of monitor involvement, whereas our sampling-based 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 overhead at runtime and in some cases obtaining faster execution of the system 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 sampling-based 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 is dedicated to experimental results. Related work is discussed in Section 7. Finally, we make concluding remarks and discuss future work in Section 8. Appendix A provides a summary of notations. All proofs appear in Appendix B.

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 Subsection 2.2, we present the concept of timed automata [1] to present the semantics of sampling-based 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.*
- *v^0: is the* initial vertex *with indegree 0, which represents the initial basic block of P.*
- *A: is a set of* arcs *(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*[1]*.* $\square$

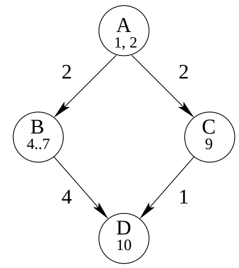
Notation: Let v be a vertex of a control-flow graph. Since the weight of all outgoing arcs from v are equal, $w(v)$ denotes the weight of the arcs that originate from v.

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).

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

```
a = scanf(...);
if (a % 2 == 0) goto 9
else {
      printf(a + "is odd");
      b = a/2;
      c = a/2 + 1;
      goto 10;
}
printf(a + "is even");
end program
```

(a) A simple C program



(b) Control-flow graph

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

2.2 Timed Automata

Let Σ be an alphabet. A *timed word* over Σ 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 $\lambda \subseteq X$, $\nu[\lambda := 0]$ denotes the clock valuation that assigns 0 to each $x \in \lambda$ and agrees with ν 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.*
- Σ *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* φ*. The set* $\lambda \subseteq X$ *gives the clocks to be reset with this switch.*
- $I : L \to \Phi(X)$ *assigns* delay invariants *to locations.* □

The semantics of a timed automaton $\mathcal{A}$ is as follows. A *state* is a pair (l, ν), where $l \in L$ and ν is a clock valuation for X. A state (l, ν) 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 ν satisfies φ, $(l, \nu) \xrightarrow{a} (l', \nu[\lambda := 0])$, 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 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

$$q_0 \xrightarrow{t_0} q'_0 \xrightarrow{a_0} q_1 \xrightarrow{t_1 - t_0} q'_1 \xrightarrow{a_1} q_2 \xrightarrow{t_2 - t_1} q'_2 \xrightarrow{a_2} q_3 \rightarrow \cdots \xrightarrow{a_k} q_{k+1}$$

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 $\langle (m, l), a, \lambda, \varphi, (m, l') \rangle$.

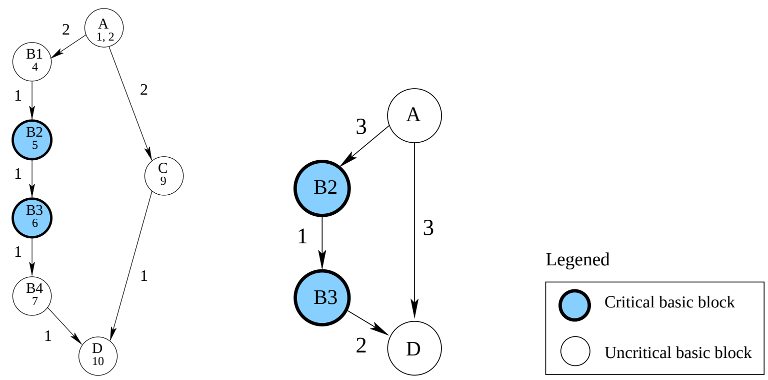
3 Formal Semantics of Sampling-based Monitoring

Given a program P, we describe the semantics of sampling-based monitoring in two steps: (1) identifying the minimum sampling period, and (2) constructing and composing a sampling-based monitor with P. Then, our goal is to show that the obtained composition never fails to detect property violations.

3.1 Calculating the Sampling Period

Let P be a program and Π be a logical property (e.g., in LTL), where P is expected to satisfy Π. Let $\mathcal{V}_\Pi$ denote the set of variables that participate in Π. In our idea of sampling-based monitoring, the monitor wakes up with some sampling period, reads the value of variables in $\mathcal{V}_\Pi$ and evaluates Π. The main challenge in this mechanism is accurate reconstruction of the state of P between two samples; i.e., if the value of a variable in $\mathcal{V}_\Pi$ changes more than once between two samples, the monitor may fail to detect violations of Π.

In order to handle value changes accurately, we modify CFG_P as follows. In the first step, we ensure that each *critical instruction* (i.e., an instruction that modifies a variable in $\mathcal{V}_\Pi$) 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 a critical instruction. 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 one basic block.



(a) Step 1 (b) Step 2

Fig. 2. Obtaining a critical CFG and calculating the sampling period.

For example, in the program in Figure 1(a), if variables `b` and `c` are of interest for verification of a property at runtime, then instructions 5 and 6 are critical and we obtain the control-flow graph in Figure 2(a).

Since uncritical vertices play no role in determining the sampling period, in the second step, we collapse uncritical 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 uncritical 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 does not affect the sampling period.

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

Definition 3. *Let* $CFG = \langle V, v^0, A, w\rangle$ *be a critical control-flow graph. The* minimum sampling period *for CFG is* $MSP_{CFG} = \min\{w(v_1, v_2) \mid (v_1, v_2) \in A \,\wedge\, v_1$ *is a critical vertex*$\}$. □

Intuitively, minimum sampling period is the minimum amount of time that two variables in $\mathcal{V}_\Pi$ get changed. For example the minimum sampling period of the control-flow graph in Figure 2(b) is $MSP = 1$. Later in this section, we will show that by applying this sampling period, no property violations can be overlooked.

3.2 Constructing and Composing Sampling-based Monitor

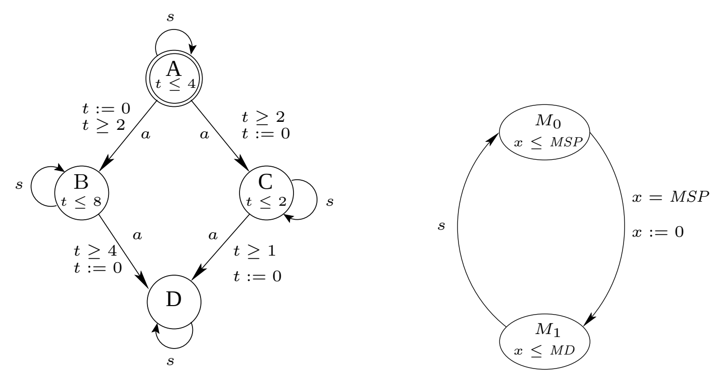
We now explain the semantics of sampling-based monitoring using timed automata[2]. Transformation of a control-flow graph $CFG = \langle V, v^0, A, w\rangle$ into a timed automaton $\mathcal{A}_{CFG} = \langle L, L^0, X, \Sigma, E, I\rangle$, where $X = \{t\}$ and $\Sigma = \{a, s\}$, is as follows:

- $L = \{l_v \mid v \in V\}$
- $L^0 = \{l_{v^0}\}$
- $E = \{\langle l_v, a, \{t\}, t \geq w(v, v'), l_{v'}\rangle \mid (v, v') \in A\} \cup \{\langle l_v, s, \{\}, true, l_v\rangle \mid v \in V\}$.
- $I(l_v)$ = 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 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 sampling-based monitor. The timed automaton obtained from the control-flow graph in Figure 1(b) is shown in Figure 3(a), where the worst-case execution time of each instruction is 2.

The relation between execution of a program P and runs of timed automaton $\mathcal{A}_{CFG_P}$ is as follows. Intuitively, a delay transition in $\mathcal{A}_{CFG_P}$ corresponds to 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 τ 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:

[2] We emphasize that our implementation does not involve the transformation presented in this subsection; i.e., we solely use the timed automata formalism to describe the semantics.



(a) Transformed Timed automaton

(b) Sampling-based Monitor

Fig. 3. Formal semantics of sampling-based monitoring.

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

such that $i \leq j \leq m$, $l \neq l'$, $(l^i, t+\tau_1)$ denotes the fact that instructions $\langle l^1 \cdots l^i \rangle$ have been executed within τ_1 time units, $\sum_{k=1}^{m} \tau_k \geq w(l, l')$ in CFG_P, and $\sum_{k=1}^{m} \tau_k$ satisfies $I(l)$. 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.

A sampling-based monitor $\mathcal{M}_P$ for program P works as follows (see Figure 3(b)). From the initial location M_0 the only outgoing switch is enabled when the sampling period is complete (i.e., $x = MSP_{CFG_P}$). The monitor may remain in location M_1 for at most MD time units, where MD is the maximum delay that can occur in execution of an instruction[3] (note that sampling never occurs in the middle of execution of an instruction). From location M_1, the monitor synchronizes with $\mathcal{A}_{CFG_P}$ on the switch labelled by s in order to read the variables of interest for evaluating properties. Thus, the parallel composition $\mathcal{A}_{CFG_P} || \mathcal{M}_P$ constructs the entire system[4]. For example, the following is a run of the automaton in Figure 3(a) composed with a monitor with sampling period $MSP = 1$ and $MD = 0$:

$$AM_0 \xrightarrow{1} A^1M_1 \xrightarrow{s} A^1M_0 \xrightarrow{1} A^2M_1 \xrightarrow{s} A^2M_0 \xrightarrow{a} BM_0 \xrightarrow{1} B^4M_1 \xrightarrow{s} B^4M_0 \xrightarrow{1} B^5M_1 \rightarrow \cdots,$$

Assumption 1 *We assume that $MD \leq MSP$.* □

Finally, we show that our construction method is sound in the sense that it never fails to detect property violations.

---

[3] Such delays are normally cause by interrupts, pipeline stalls, I/O operations, etc.

[4] We note that our work is not concerned with evaluating properties when a sample is taken and, thus, this issue is not addressed.

Theorem 1. *Let P be a program and $w = (a_0, t_0), (a_1, t_1) \cdots$ be a timed word of $\mathcal{A}_{CFG_P} || \mathcal{M}_P$. 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.* □

4 Optimizing Sampling Period and its Complexity

Employing the minimum sampling period as identified in Section 3 results in highly frequent involvement of the monitor in the system at runtime. Obviously, increasing the sampling period naively leads to inability of the monitor to reconstruct the state of the program at sampling time. Thus, in order to reduce the sampling points, we use auxiliary memory to build a history of critical 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 $IT(CFG, v)$. Observe that adding the extra instruction does not affect the calculation of the sampling period. 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.

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 delete 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 runtime, 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.

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 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.* □

Obviously, sampling-based monitoring and in particular, increasing the sampling period introduces *detection latencies*. To tackle this problem, one can specify a *tolerable detection delay* for critical variables. This factor can be easily incorporated in our transformation technique and optimization problem.

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 a given source code and a desired sampling period *SP* as input. Our objective is to find the minimum number of vertices that must be removed from V.

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. $\mathbf{a} = \{a_v \mid v \in V\}$: where each a_v is an integer variable which represents the weight of arcs originating from vertex v. Recall that all the outgoing arcs of a vertex have the same weight in *CFG*. This variable is needed to store the new weight of an arc created by merging a sequence of arcs. For example, in Figure 2(b), initially, variable $a_{B_2} = 1$. However, if $x_{B_3} = 1$ (i.e., vertex B_3 is removed), then $a_{B_2} = 3$.
3. $\mathbf{y} = \{y_v, y'_v \mid v \in V\}$, called *choice variables*, where each y_v and y'_v 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:

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

$$a_{v^0} = w(v^0) \tag{2}$$

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:

$$a_u + SP.x_v \geq SP \tag{3}$$

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 $a_v = w(v)$.
- **Case 2:** If $x_v = 1$, then $a_v = w(v) + w(u)$, where $(u, v) \in A$. Note that in this case, although vertex v is removed, for simplicity, we use variable a_v as the weight of the newly created arc. 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 and y'_v are such that one of them is zero and the other is a_u. This property enforces mutual exclusiveness of the above cases.
- **Prop. 2:** If $x_v = 1$, then $y_v = a_u$ and $y'_v = 0$. On the contrary, if $x_v = 0$, then $y_v = 0$ and $y'_v = a_u$.

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:

$$y_v + y'_v = a_u \tag{4}$$

$$sos_1(y_v, y'_v) \tag{5}$$

$$1 \leq x_v + y'_v \leq a_u \tag{6}$$

The following constraints implement Case 1 and 2, respectively:

$$w(v) + a_u - y'_v = a_v \tag{7}$$

$$y_v + w(v) = a_v \tag{8}$$

For example, if v is deleted (i.e., $x_v = 1$), then we have $y_v = 0$ and $y'_v = a_u$ by Constraints 4-6. Moreover, when v is deleted, the weight of the newly created arc a_v will be $a_u + w(v)$. This is ensured by Constraints 7 and 8.

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 with variables $a_{u_1}, a_{u_2} \cdots a_{u_n}$ and $a_v^{u_1}, a_v^{u_2} \cdots a_v^{u_n}$. 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*:

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

6 Experimental Results

In this section, we present the results of our experiments using the following tool chain. First, we generate the control-flow graph of a given C program using the tool CIL [24]. Next, we generate the critical control-flow graph and transform it into an ILP model. The model is given to the tool lp_solve [22] to obtain the optimal sampling period and the size of auxiliary memory. We use the breakpoint mechanism of gdb [9] to implement monitors. Finally, a Python script controls gdb and handles possible exceptions.

Our case studies are from the MiBench [11] benchmark suite. For reasons of space, we only present experimental results for two case studies:

1. Blowfish: This benchmark has 745 lines of code, which results in a CFG of 169 vertices and 213 arcs. We take 20 variables for monitoring.
2. Dijkstra: This benchmark has 171 lines of code, which results in a CFG of 65 vertices and 78 arcs. We take 8 variables for monitoring.

Our observation is that other benchmarks behave similarly. All experiments in this section are conducted on a Mac Book Pro with 2.26GHz Intel Core 2 Duo and 2GB main memory.

We consider the following different settings for our experiments:

– **Event-based:** gdb extracts the new value of variables of interest whenever they get changed throughout the program execution.
– **Sampling-based with no history:** gdb is invoked every *MSP* time units (see Subsection 3.1) to extract the value of all the variables of interest.

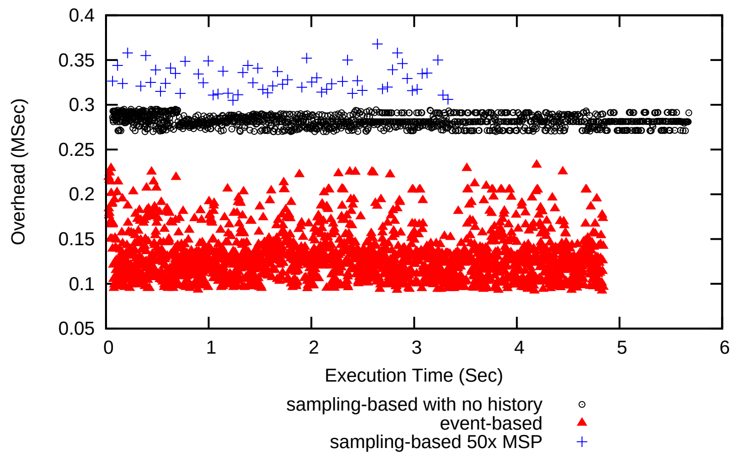


Fig. 4. Experimental results for Dijktra ($50 * MSP$ sampling period).

– **Sampling-based with history:** This setting incorporates our ILP optimization. Thus, whenever gdb is invoked, it extracts the value of variables of interest as well as the history.

In the event-based setting (see Figures 4 and 5), since the monitor interrupts the program execution irregularly, unequal bursts in the overhead can be seen. Moreover, the overhead caused by each data extraction is proportional to the data type. Hence, the data extraction overhead varies considerably from one interruption to another. Thus, the monitor introduces probe-effects, which in

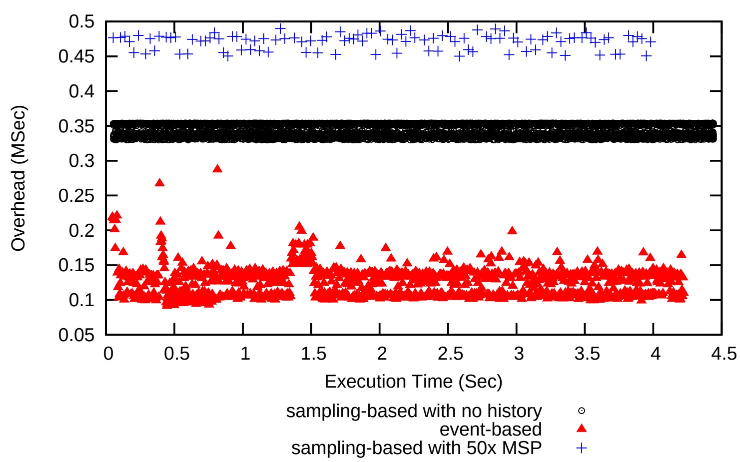


Fig. 5. Experimental results for Blowfish ($50 * MSP$ sampling period).

turn may create unpredictable and even incorrect behaviour. This anomaly is, in particular, unacceptable for real-time embedded and mission-critical systems.

On the contrary, since the sampling-based monitor interrupts the program execution on a regular basis, the overhead introduced by data extraction is not subject to any bursts and, hence, remains consistent and bounded (see Figures 4 and 5). Consequently, the monitored program exhibits a predictable behaviour. Obviously, the sampling-based monitor may potentially increase the overhead, which extends the overall execution time. Nonetheless, in many commonly considered applications, designers prefer predictability at the cost of larger overhead.

Regarding the third setting, recall that we prohibited deletion of self-loops from critical control-flow graphs. Hence, if some variables get updated in loops, the minimum sampling period of loops, can determine the optimal sampling period. For example, in both case studies, since the majority of the variables of interest are updated in loops, we cannot increase the sampling period beyond $4 * MSP$. In such a situation, employing the new sampling period and history does not achieve much. To overcome this problem, we devise a simple heuristic that makes a conservative estimate of the size of a buffer needed to build the history for loops. By incorporating this heuristic, we allow deletion of self-loops. For example, in both case studies, the ILP solver can increase the sampling period up to $100 * MSP$. We note that solving the corresponding ILP problem for all benchmarks take an average of 3 minutes. This clearly shows that we are not even close to the boundaries of ILP solving.

Figures 4 and 5 show the results of our experiments for sampling period of $50 * MSP$. As can be seen, increasing the sampling period results in larger overhead. This is because the monitor needs to read a larger amount of data formed by the history. However, the increase in overhead is considerably small (less than twice the original overhead). Having said that, the other side of the coin is that by increasing the sampling period, the program is subject to less monitoring interrupts. This results in significant decrease in the overall execution time of the programs. This is indeed advantageous for monitoring hard real-time programs. Although adding history causes variability in data extraction overhead, the system behavior is still highly predictable as compared to the event-based setting.

The above observations are valid for the case, where we increase the sampling period by $100 * MSP$ as well (see Figures 6 and 7). Observe that the reduction in execution time of Blowfish is less than Dijkstra, as the overhead of data extraction in Blowfish is proportionally larger than Dijkstra. This is due to the fact that in Blowfish more and larger variables are stored in the history between two samples. On the other hand, overhead variability in Blowfish is less than Dijkstra, as the number of variables stored in the history from one sample to another does not significantly vary in Blowfish.

We now clarify why the sampling-based method naturally incurs more overhead than event-based approaches. While in the event-based method, the monitor only executes whenever a new event occurs, in the sampling-based method, the monitor sometimes takes a sample although no significant event has oc-

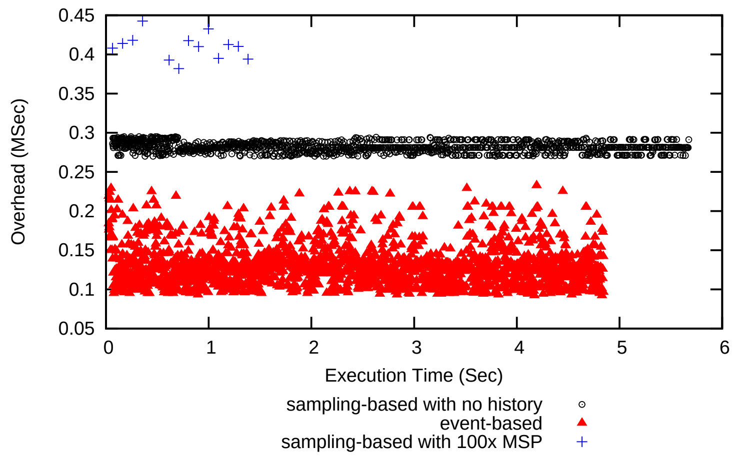


Fig. 6. Experimental results for Dijktra ($100 * MSP$ sampling period).

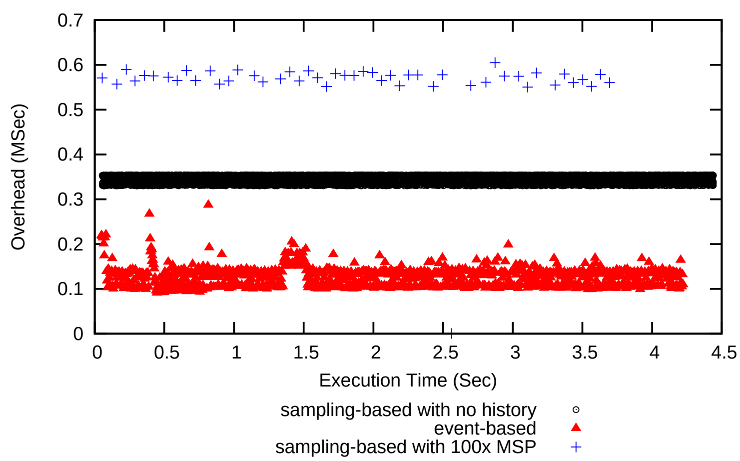


Fig. 7. Experimental results for Blowfish ($100 * MSP$ sampling period).

curred since the last sample. Also, in the sampling-based method, the monitor must collect all relevant variables while in the even-based method the monitor only processes the single new event. Thus, higher overhead of the sampling-based approach is natural and expected.

Finally, we discuss the tradeoff between execution time and the added memory consumption when the sampling period is increased (see Figures 8 and 9). As can be seen, as we increase the sampling period, the system requires negligible extra memory. Also, one can clearly observe the proportion of increase in memory usage versus the reduction in the execution time. In other words,

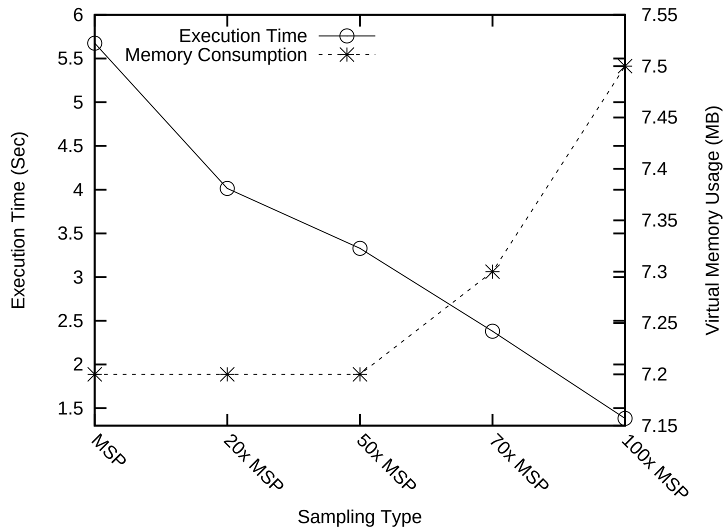


Fig. 8. Memory usage vs. execution time (Dijkstra).

by employing small amount of auxiliary memory, one can achieve considerable speedups.

In summary, our experiments show practical evidence and support our intuition on the benefit of using sampling-based monitors:

> *Sampling-based monitors provide bounded overhead and predictable program behavior. Increasing the sampling period to gain speedups is possible at the cost of adding negligible auxiliary memory.*

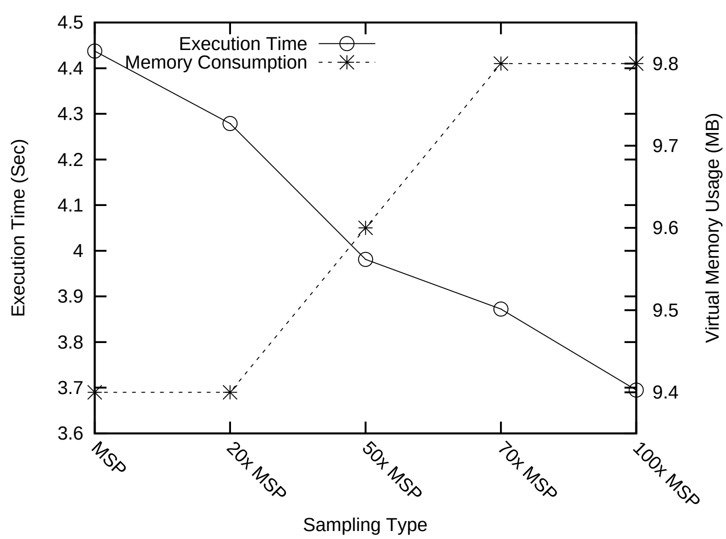


Fig. 9. Memory usage vs. execution time Blowfish.

7 Related Work

In classic runtime verification [26], 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. To the best of our knowledge, in the literature of runtime verification, monitors are event-triggered [21] in the sense that every change in the state of the system invokes the monitor for analysis.

From the logical and language point of view, runtime verification has mostly been studied in the context of Linear Temporal Logic (LTL) properties [2,10,13–15,28] and in particular safety properties [16,27]. Other languages and frameworks have also been developed for facilitating specification of temporal properties [19,20,29]. Runtime verification of ω-languages was considered in [6]. In [7], the authors address runtime verification of safety-progress [4,23] properties.

In [8], the authors introduce a sampling-based program monitoring technique. They propose a framework that allows quantitative reasoning about issues involved in sampling-based techniques. They also discuss how to optimally instrument a program by a set of *markers*, such that different execution paths reachable from the same state are distinguishable. In the same context, in [25], the authors propose the language Capilot 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 designing 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. Finally, in [17], the authors propose a method to control the overhead of software monitoring using control theory for discrete event systems. In this work, overhead control is achieved by temporarily disabling involvement of monitor, thus avoiding the overhead to pass a user-defined threshold.

8 Conclusion

In this paper, we investigated a sampling-based approach for runtime verification. We explored the problem by defining it in formal terms and then showed that the optimization problem for using minimum auxiliary memory to maximize the sampling period is NP-complete. As a practical solution, we encoded our problem in Integer Linear Programming (ILP). Our approach is implemented in a tool chain that 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 sampling-based 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 adapts its sampling period based upon the structure of the input program. Such adaptive sampling will be highly beneficial to overcome loop problems. Another interesting direction is to weave a sampling-based monitor with the program not as a separate process, but as a built-in verification mechanism. Also, one may consider developing hybrid monitors that take advantage of both event-triggered as well as time-triggered techniques.

9 Acknowledgement

This research was supported in part by NSERC DG 357121-2008, ORF RE03-045, ORE RE-04-036, and ISOP IS09-06-037.

References

1. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
2. A. Bauer, M. Leucker, and C. Schallhart. Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2009. in press.
3. A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL Semantics for Runtime Verification. *Journal of Logic and Computation*, 20(3):651–674, 2010.
4. E. Y. Chang, Z. Manna, and A. Pnueli. Characterization of Temporal Property Classes. In *Automata, Languages and Programming (ICALP)*, pages 474–486, 1992.
5. S. Colin and L. Mariani. *Run-Time Verification*, chapter 18. Springer-Verlag LNCS 3472, 2005.
6. M. d'Amorim and G. Rosu. Efficient Monitoring of omega-Languages. In *Computer Aided Verification (CAV)*, pages 364–378, 2005.
7. Y. Falcone, J.-C. Fernandez, and L. Mounier. Runtime Verification of Safety-Progress Properties. In *Runtime Verification (RV)*, pages 40–59, 2009.
8. 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.
9. GNU debugger. `http://www.gnu.org/software/gdb/`.
10. D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Automated Software Engineering (ASE)*, pages 412–416, 2001.
11. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on In Workload Characterization (WWC)*, pages 3–14, 2001.
12. K. Havelund and A. Goldberg. Verify your Runs. pages 374–383, 2008.
13. K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. *Electronic Notes in Theoretical. Computer Science*, 55(2), 2001.
14. K. Havelund and G. Rosu. Monitoring Programs Using Rewriting. In *Automated Software Engineering (ASE)*, pages 135–143, 2001.

15. 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.
16. K. Havelund and G. Rosu. Efficient Monitoring of Safety Properties. *Software Tools and Technology Transfer (STTT)*, 6(2):158–173, 2004.
17. 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)*, 2011. To appear.
18. R. M. Karp. Reducibility Among Combinatorial Problems. In *Symposium on Complexity of Computer Computations*, pages 85–103, 1972.
19. 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.
20. 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.
21. O. Kupferman and M. Y. Vardi. Model Checking of Safety Properties. In *Computer Aided Verification (CAV)*, pages 172–183, 1999.
22. ILP solver lp_solve. `http://lpsolve.sourceforge.net/5.5/`.
23. Z. Manna and A. Pnueli. A Hierarchy of Temporal Properties. In *Principles of Distributed Computing (PODC)*, pages 377–410, 1990.
24. G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. Proceedings of Conference on Compilier Construction, 2002.
25. L. Pike, A. Goodloe, R. Morisset, and S. Niller. Copilot: A Hard Real-Time Runtime Monitor. In *Runtime Verification (RV)*, 2010. 345-359.
26. A. Pnueli and A. Zaks. PSL Model Checking and Run-Time Verification via Testers. In *Symposium on Formal Methods (FM)*, pages 573–586, 2006.
27. 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.
28. V. Stolz and E. Bodden. Temporal Assertions using Aspectj. *Electronic Notes in Theoretical Computer Science*, 144(4), 2006.
29. W. Zhou, O. Sokolsky, B. T. Loo, and I. Lee. *MaC*: Distributed Monitoring and Checking. In *Runtime Verification (RV)*, pages 184–201, 2009.

Appendix

A Summary of Notations

CFG	control-flow graph
V	set of vertices
A	set of arcs
v^0	initial basic block (vertex)
$w(u,v)$	weight of arc (u,v)
$w(v)$	weight of outgoing arcs from v in a control-flow graph
$inst_v$	set of instructions in basic block v
$\langle v^1 \cdots v^n \rangle$	set of n instructions in basic block v
$\mathcal{A}$	timed automata
L	set of locations
L^0	set of initial locations
E	set of switches
Σ	alphabet
$I(l)$	delay invariant of location l
$\Phi(X)$	set of all clock constraints on clocks X
Π	logical property
$\mathcal{V}_\Pi$	set of variables that participate in property Π
$T(\mathit{CFG}, v)$	graph transformation on vertex v for control-flow graph CFG
MSP	minimum sampling period
$IT(\mathit{CFG}, v)$	instrumenting graph transformation on vertex v for control-flow graph CFG
$\mathcal{M}_P$	sampling-based monitor for program P
MD	maximum delay

B Proofs

Theorem 1. *Let P be a program and $w = (a_0, t_0), (a_1, t_1) \cdots$ be a timed word of $\mathcal{A}_{CFG_P} || \mathcal{M}_P$. 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 theorem holds by construction of $\mathcal{M}_P$, as it enforces sampling period *MSP*. 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 *MSP*) 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 M_1 in $\mathcal{M}_P$ causes no incorrectness.
- Finally, removing self-loops from uncritical vertices also create no problems, since those loops contain no critical instructions. Thus, no matter how many times such loops iterate, the minimum sampling period guarantees correctness. □

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 reduction the *Minimum Vertex Cover Problem* (VC) [18] 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 $w(v, u) = 1$, where $v \neq v^0$.

  , such that and .
- 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:

- (⇒) 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.

- (⇐) 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. □