A Case Study on Runtime Monitoring of an Autonomous Research Vehicle (ARV) System

Aaron Kane[1], Omar Chowdhury[2], Anupam Datta[1], and Philip Koopman[1]

[1] Carnegie Mellon University, Pittsburgh, PA
[2] Purdue University, West Lafayette, IN
akane@alumni.cmu.edu, ochowdhu@purdue.edu, danupam@cmu.edu, koopman@cmu.edu

Abstract. Runtime monitoring is a versatile technique for detecting property violations in safety-critical (SC) systems. Although instrumentation of the system under monitoring is a common approach for obtaining the events relevant for checking the desired properties, the current trend of using black-box commercial-off-the-shelf components in SC system development makes these systems unamenable to instrumentation. In this paper we develop an online runtime monitoring approach targeting an autonomous research vehicle (ARV) system and recount our experience with it. To avoid instrumentation we passively monitor the target system by generating atomic propositions from the observed network state. We then develop an efficient runtime monitoring algorithm, `EgMon`, that *eagerly* checks for violations of desired properties written in future-bounded, propositional metric temporal logic. We show the efficacy of `EgMon` by implementing and empirically evaluating it against logs obtained from the testing of an ARV system. `EgMon` was able to detect violations of several safety requirements.

1 Introduction

Runtime verification (RV) is a promising alternative to its static counterparts (*e.g.*, model checking [9] and theorem proving [6]) for checking safety and correctness properties of safety-critical embedded systems. In RV, a runtime monitor observes the concrete execution of the system in question and checks for violations of some stipulated properties. When the monitor detects a violation of a property, it notifies a command module which then attempts to recover from the violation. *In this paper, we develop a runtime monitor that monitors an autonomous research vehicle (ARV) and describe our experience with it.*

The ARV is an autonomous heavy truck which is being designed for use in vehicle platoons. It is representative of common modern ground vehicle designs. These systems are generally built by system integrators who utilize commercial-off-the-shelf components developed by multiple vendors, some of which may be provided as black-box systems. These systems are also often hard real-time systems which leads to additional constraints on system monitoring [13]. This type of system architecture is incompatible with many existing runtime monitoring techniques, which often require program or system instrumentation [4, 7, 15, 19] to obtain the relevant events or system properties (*e.g.*, propositions) necessary to check for violations. Without access to component source code instrumenting systems is more difficult, and even when the source is available there are risks of affecting the timing and correctness of the target system when instrumented.

Obtaining relevant system state. To avoid instrumentation, we obtain the relevant information for monitoring the ARV system through passive observation of its broadcast buses. Controller area network (CAN) is a standard broadcast bus for ground vehicles which is the primary system bus in the ARV. We can obtain useful amounts of system-state relevant information for monitoring the system safety specification by observing the data within the CAN messages that are broadcasted between system components. Before we can start monitoring the ARV system, we need a component, which we call the StP (in short, *state to proposition map*), that observes messages transmitted on the bus and interprets them into propositions relevant to monitoring which are then fed into the monitor. We want to emphasize that the limits of external observability can cause significant challenges in designing the StP when considering the state available from the system messages and the necessary atomic propositions [17].

Specification logic. To obtain the relevant safety requirements and invariants for monitoring the ARV system we consulted the safety requirements of the ARV system. Many desired properties for these types of systems are timing related, so using an explicit-time based specification language for expressing these properties is helpful. System requirements of the form "*the system must perform action a within t seconds of event e*" are common, for instance, "*Cruise control shall disengage for 250ms within 500ms of the brake pedal being depressed*". For efficient monitoring, we use a fragment of propositional, discrete time, future-bounded metric temporal logic (MTL) [20].

Monitoring algorithm. We have developed a runtime monitoring algorithm, which we call `EgMon`, that incrementally takes as input a system state (*i.e.*, a state maps propositions to either true/false) and a MTL formula and eagerly checks the state trace for violations. Some existing monitoring algorithms that support bounded future formulas wait for the full-time of the bound before evaluating the formula (*e.g.*, [2]). `EgMon` uses a dynamic programming based iterative algorithm that tries to reduce the input formula as soon as possible using history summarizing structures and formula-rewriting (leaving a partially reduced formula when future input is required). This eager nature of the algorithm can detect a violation earlier, leaving the system more time to attempt a graceful recovery. We have also proved the correctness of our algorithm. As the target systems we envision to monitor have strict time restriction, it is possible that the eager checks performed by `EgMon` are not finished before the next trace state arrives, possibly leaving trace properties unchecked. To overcome this, we have developed a hybrid monitoring algorithm, HMon, that first performs conservative checking like traditional runtime monitoring algorithms for MTL and performs as many eager checks as the remaining time permits.

Empirical evaluation. We have implemented both `EgMon` and HMon on an inexpensive embedded platform and empirically evaluated it against logs obtained from the testing of an ARV system using properties derived from its safety requirements. `EgMon` (resp., HMon) has moderate monitoring overhead and detected several safety violations.

2 Background and Existing Work

In this section we briefly introduce the background concepts and discuss relevant existing work that will put the current work in perspective.

Monitoring architecture. Goodloe and Pike present a thorough survey of monitoring distributed real-time systems [13]. Notably, they present a set of monitor architecture constraints and propose three abstract monitor architectures in the context of monitoring these types of systems. One of the proposed distributed real-time system monitor architectures is the bus-monitor architecture. This architecture contains an external monitor which receives network messages over an existing system bus, acting as another system component. The monitor can be configured in a silent or receive only mode to ensure it does not perturb the system. This is a simple architecture which requires minor changes to the target system. We utilize this architecture for our monitoring framework.

Controller area network. Controller Area Network is a widely used automotive network developed by Bosch in the 1980s [5]. In this work we primarily focus on CAN as it is a common automotive bus which typically conveys enough of the state information so that we can check for interesting safety properties of the system. CAN is an event-based broadcast network with data rates up to 1Mb/s. Messages on CAN are broadcast with an identifier which is used to denote both the message and the intended recipients. The message identifiers are also used as the message priorities for access control. Although CAN is an event-based bus, it is often used with periodic scheduling schemes so the network usage can be statically analyzed. Hence, our monitoring approach is based on a time-triggered, network sampling model which allows it to monitor time-triggered networks as well. We use `EgMon` as a passive external bus-monitor which can only check system properties that are observable by passive observation of the messages transmitted through CAN.

Monitoring algorithm. Our monitoring algorithm is similar to existing dynamic programming and formula-rewriting based algorithms [3, 14, 15, 24, 25]. Our main area of novelty is the combination of eager and conservative specification checking used in a practical setting showing the suitability of our bounded future logic for safety monitoring. Our monitoring algorithm is inspired by the algorithms *reduce* [12] and *prècis* [8], adjusted for propositional logic and eager checking. The structure of our algorithm is based on *reduce*. *reduce*, *prècis*, and `EgMon` can handle future incompleteness but *reduce* additionally considers incompleteness for missing information which we do not consider. The NASA PathExplorer project has led to both a set of dynamic programming-based monitoring algorithms as well as some formula-rewriting based algorithms [15] for past-time LTL. The formula rewriting algorithms utilize the Maude term rewriting engine to efficiently monitor specifications through formula rewriting [24]. Thati and Roşu [25] describe a dynamic programming and rewriting-based algorithm for monitoring MTL formulas. They perform eager runtime monitoring by formula rewriting which resolves past-time formulas into equivalent formulas without unguarded past-time operators and derive new future-time formulas which separate the current state from future state. While they have a tight encoding of their canonical formulas, they still require more memory than some existing algorithms (formulas grow in size as they are rewritten), including `EgMon`.

Heffernan et. al. present a monitor for automotive systems using ISO 26262 as a guide to identify the monitored properties [16]. They monitor past-time linear temporal logic (LTL) formulas and obtain system state from target system buses (CAN in their example). Our StP component is similar to their "filters" used to translate system state to the atomic propositions used in the policy. Their motivation and goals are similar to ours, but they use system-on-a-chip based monitors which utilize instrumentation to obtain system state, which is not suitable for monitoring black-box commercial-off-the-shelf (COTS) systems. Reinbacher et. al. present an embedded past-time MTL monitor in [23] which generates FPGA-based non-invasive monitors. The actual implementation they describe does however presume system memory access to obtain system state (rather than using state from the target network). Pellizzoni et. al. describe a monitor for COTS peripherals in [22]. They generate FPGA monitors that passively observe PCI-E buses to verify system properties, but they only check past-time LTL and regular expressions which cannot capture timing properties. Basin et. al. compare runtime monitoring algorithms for MTL properties [3]. `EgMon` works similarly to their point-based monitoring algorithm but `EgMon` checks future temporal operators more aggressively. Donzé *et al.* [11] developed a robustness monitor for Signal Temporal Logic which supports continuous signals. Nickovic and Maler [21] developed the AWT tool which monitors analog systems. We only consider discrete events. Dokhanchi *et al.* [10] developed an online runtime monitoring algorithm for checking the robustness of formulas written in a future-bounded MTL fragment. We consider satisfaction of the formula instead of robustness.

3 Monitoring Algorithm

For checking whether the given ARV system adheres to its specification, we need an algorithm which incrementally checks explicit time specifications (*i.e.*, propositional metric-time temporal logic [20]) over finite, timed system traces. This has led to our algorithm `EgMon` which is an iterative monitoring algorithm based on formula rewriting and summarizing the relevant history of the trace in *history-structures*. To detect violations early, `EgMon` eagerly checks whether it can reduce subformulas of the original formula to a truth value by checking the (potentially incomplete) trace history and using formula simplifications (*e.g.*,

$a \wedge false \equiv false$). Many of the existing algorithms for evaluating formulas such as $\Diamond_{[l,h]} a \vee b$ (*i.e.*, either b is true or sometimes in the future a is true such that the time difference between the evaluation state and the future state in which a is true, t_d, is within the bound $[l, h]$) wait enough time so that $\Diamond_{[l,h]} a$ can be fully evaluated. `EgMon` however tries to eagerly evaluate both $\Diamond_{[l,h]} a$ and b immediately and see whether it can reduce the whole formula to a truth value. For another eagerness checking example, let us assume we are checking the property $a \mathcal{U}_{[l,h]} b$ (read, the formula is true at trace position i if there is a trace position j in which b holds such that $j \geq i$ and the time difference between position i and j is in the range $[l, h]$ and for all trace positions k such that $i \leq k < j$ the formula a holds) at trace position i. While monitoring if we can find a trace position $k > i$ for which a is false and no previous b's (*e.g.*, at position $l, i \leq l < k$) are true then we can evaluate the formula to be false without waiting for a trace position in which b is true. We want to emphasize that `EgMon` optimistically checks for violations and hence we could have a trace in which each formula can only be evaluated at the last possible trace position which causes our algorithm will behave in the exact same way as the non-eager algorithms modulo the extra computation for eager checking.

3.1 Specification Logic

Our safety specification language for the ARV system, which we call $\alpha\mathcal{VSL}$, is a future-bounded, discrete time, propositional metric temporal logic (MTL [20]). The syntax of $\alpha\mathcal{VSL}$ is as follows:

$$\varphi ::= \mathtt{t} \mid \mathtt{p} \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathcal{S}_{\mathbb{I}} \varphi_2 \mid \varphi_1 \mathcal{U}_{\mathbb{I}} \varphi_2 \mid \circleddash_{\mathbb{I}} \varphi \mid \bigcirc_{\mathbb{I}} \varphi$$

Syntax. $\alpha\mathcal{VSL}$ has logical true (*i.e.*, $\mathtt{t}$), propositions $\mathtt{p}$, logical connectives (*i.e.*, $\neg, \vee$), past temporal operators *since* and *yesterday* ($\mathcal{S}, \circleddash$), and future temporal operators *until* and *next* ($\mathcal{U}, \bigcirc$). Other temporal operators (*i.e.*, $\diamondminus$,$\boxminus$,$\Diamond$,$\Box$) can be easily derived from the ones above. There is a bound $\mathbb{I}$ of form $[l, h]$ ($l \leq h$ and $l, h \in \mathbb{N} \cup \infty$) associated with each temporal operator. Note that the bound $[l, h]$ associated with the future temporal operators must be finite. Specification propositions $\mathtt{p}$ come from a finite set of propositions provided in the system trace by the StP. These propositions are derived from the observable system state and represent specific system properties, for instance, proposition $\mathsf{speedLT40mph}$ could describe whether the vehicle speed is less than 40mph. We use φ, ϕ, α, and β (possibly with subscripts) to denote valid $\alpha\mathcal{VSL}$ formulas.

Semantics. $\alpha\mathcal{VSL}$ formulas are interpreted over time-stamped *traces*. A trace σ is a sequence of states, each of which maps all propositions in StP, to either $\mathtt{t}$ or $\mathtt{f}$. We denote the i^{th} position of the trace with σ_i where $i \in \mathbb{N}$. Moreover, each σ_i has an associated time stamp denoted by τ_i where $\tau_i \in \mathbb{N}$. We denote the sequence of time stamps with τ. For all $i, j \in \mathbb{N}$ such that $i < j$, we require $\tau_i < \tau_j$. For a given trace σ and time stamp sequence τ, we write $\sigma, \tau, i \models \varphi$ to denote that the formula φ is true with respect to the i^{th} position of σ and τ. The semantics of $\alpha\mathcal{VSL}$ future bounded MTL is standard, see for instance, [2]. Each property φ has an implicit unbounded $\Box$ future operator ($\Box\varphi$ signifies that φ

is true in all future trace positions including the current trace position) at the top-level which is handled by checking whether φ is true in each trace position.

$$\texttt{tempSub}(\varphi) = \begin{cases} \emptyset & \text{if } \varphi \equiv p \\ \{\alpha\} \cup \{\beta\} \cup \texttt{tempSub}(\alpha) \cup \texttt{tempSub}(\beta) & \text{if } \varphi \equiv \alpha\,\mathcal{U}_{\mathbb{I}}\beta | \alpha\,\mathcal{S}_{\mathbb{I}}\beta \\ \texttt{tempSub}(\alpha) \cup \texttt{tempSub}(\beta) & \text{if } \varphi \equiv \alpha \vee \beta \\ \texttt{tempSub}(\alpha) & \text{if } \varphi \equiv \neg\alpha \end{cases}$$

We now introduce the readers with some auxiliary notions which will be necessary to understand our algorithm `EgMon`. We first define "*residual formulas*" or, just "*residues*". Given a formula φ, we call another formula ϕ as φ's residual, if we obtain ϕ after evaluating φ with respect to the current information of the trace. Note that a formula residue might not be a truth value if the formula could not conclusively be reduced given the current trace state (e.g, if future state is required to determine the truth value). A residue r^j_φ is a tagged pair $\langle j, \phi \rangle_\varphi$ where j is a position in the trace in which we intend to evaluate φ (the original formula) and ϕ is the current residual formula. We use these residues to efficiently hold trace history for evaluating temporal formulas. The next notion we introduce is of "*wait delay*". It is a function Δ^w that takes as input a formula φ and $\Delta^w(\varphi)$ returns an upper bound on the time one has to wait before they can evaluate φ with certainty. For past- and present-time formulas ϕ, $\Delta^w(\phi) = 0$. Future time formulas have a delay based on the interval of the future operator (*e.g.*, $\Delta^w(\Diamond_{[0,3]}\mathsf{p}) = 3$). The length of a formula φ, denoted $|\varphi|$, returns the total number of subformulas of φ.

3.2 `EgMon` Algorithm

Our runtime monitoring algorithm `EgMon` takes as input an $\alpha\mathcal{VSL}$ formula φ and monitors a growing trace, building history structures and reporting the specification violations as soon as they are detected. We summarize the relevant algorithm functions below:

`EgMon`(φ) is the top-level function.
`reduce`$(\sigma_i, \tau_i, \mathbb{S}^i_\varphi, \langle i, \varphi \rangle_\varphi)$ reduces the given residue based on the current state (σ_i, τ_i) and the history $\mathbb{S}^i_\varphi$.
`tempSub`(φ) identifies the subformulas which require a history structure to evaluate the formula φ.
`incrS`$(S^{i-1}_\varphi, \mathbb{S}^i_\varphi, \sigma_i, \tau_i, i)$ updates the history structure S^{i-1}_φ to step i given the current trace and history state.

Top-level monitoring algorithm. The top-level monitoring algorithm `EgMon` is a sampling-based periodic monitor which uses history structures to store trace state for evaluating temporal subformulas. *History structures* are lists of residues along with past-time markers for evaluating infinite past-time formulas. The algorithm checks the given formula φ periodically at every trace sample step. When

```
1: For all recognized formulas $\phi \in \texttt{tempSub}(\varphi)$: $S_\phi^{-1} \leftarrow \emptyset$
2: $i \leftarrow 0$
3: loop
4:     Obtain next trace step $(\sigma_i, \tau_i)$
5:     for every $\phi \in \texttt{tempSub}(\varphi)$ in increasing size do
6:         $S_\phi^i \leftarrow \texttt{incrS}(S_\phi^{i-1}, \mathbb{S}_\phi^i, \sigma_i, \tau_i, i)$
7:     end for
8:     $S_\varphi^i \leftarrow \texttt{incrS}(S_\varphi^{i-1}, \mathbb{S}_\varphi^i, \sigma_i, \tau_i, i)$
9:     for all $\langle j, \texttt{f} \rangle \in S_\varphi^i$ do
10:        Report violation on $\sigma$ at position $j$
11:    end for
12:    $i \leftarrow i + 1$
13: end loop
```

Fig. 1. `EgMon` Algorithm

the formula cannot be decided at a given step (*e.g.*, it requires future state to evaluate), the remaining formula residue is saved in a history structure for evaluation in future steps when the state will be available. The history structure for formula ϕ at trace step i is denoted S_ϕ^i. We use $\mathbb{S}_\varphi^i$ to denote the set of history structures for all temporal subformula of φ, *i.e.*, $\mathbb{S}_\varphi^i = \bigcup_{\phi \in \texttt{tempSub}(\varphi)} S_\phi^i$.

The high level algorithm `EgMon` is shown in Figure 1. First, all the necessary history structures S_ϕ are identified using $\texttt{tempSub}(\varphi)$ and initialized. Once these structures are identified, the monitoring loop begins. In each step, all the history structures are updated with the new trace step. This is done in increasing formula size since larger formula can depend on the history of smaller formula (which may be their subformula). Each structure is updated using $\texttt{incrS}(S_\phi^{i-1}, \mathbb{S}_\phi^i, \sigma_i, \tau_i, i)$ which adds a residue for the current trace step to the structure and reduces all the contained residues with the new step state. Then, the same procedure is performed for the top level formula that is being monitored – the formula's structure is updated with $\texttt{incrS}(S_\varphi^{i-1}, \mathbb{S}_\varphi^i, \sigma_i, \tau_i, i)$. Once updated, this structure contains the evaluation of the top-level formula. The algorithm reports any identified formula violations (*i.e.*, any `f` residues) before continuing to the next trace step. We note that due to the recursive nature of the monitoring algorithm, the top-level formula is treated exactly the same as any temporal subformula would be (which follows from the fact that the top-level formula contains an implicit *always* $\Box$). The history structure updates for the top-level formula are separated in the algorithm description for clarity only. The only difference between the top-level formula and other temporal subformula is that violations are reported for the top-level formula.

Reducing Residues. `EgMon` works primarily by reducing formula residues down to truth values. Residues are reduced by the $\texttt{reduce}(\sigma_i, \tau_i, \mathbb{S}_\varphi^i, \langle j, \phi \rangle_\varphi)$ function, which uses the current state (σ_i, τ_i) and the stored history in $\mathbb{S}_\varphi^i$ to rewrite the formula ϕ to a reduced form, either a truth value or a new formula which will evaluate to the same truth value as the original. For past or present-time formulas, `reduce`() is able to return a truth value residue since all the necessary information to decide the formula is available in the history and current state. In a given state, if the input formula $\varphi \equiv p$, `reduce` returns true only if

p is true in the state and returns false otherwise. For input formula of form $\varphi \equiv \varphi_1 \vee \varphi_2$, `reduce` is recursively called for φ_1 and φ_2, respectively, and the formula $\varphi_1 \vee \varphi_2$ is reduced to $\varphi_a \vee \varphi_b$ (simplified if necessary) where φ_a and φ_b are reduced form of φ_1 and φ_2, respectively. Negation is handled similarly. Future-time policies may be fully-reducible if enough state information is available. If a future-time formula cannot be reduced to a truth value, it is returned as a reduced (potentially unchanged) residue. For residues whose formula is an *until* formula $\alpha \mathcal{U}_{[l,h]} \beta$, the history structures S^i_α and S^i_β are used to reduce the formula. If the formula can be evaluated conclusively then the truth value is returned, otherwise the residue is returned unchanged. The reduction algorithm for *until* temporal formula is shown below. Reducing *since* formulas is essentially the same except with reversed minimum/maximums and past time bounds.

```
$\mathbf{reduce}(\sigma_i, \tau_i, i, \mathbb{S}^i_{\alpha \mathcal{U}_{[l,h]} \beta}, \langle j, \alpha \mathcal{U}_{[l,h]} \beta \rangle) ::=$
$\text{let } a_a \leftarrow min(\{k | \tau_j \le \tau_k \le \tau_j + h \wedge \langle k, \bot \rangle \in S^i_\alpha\}, i)$
$a_u \leftarrow max(\{k | \tau_k \in [\tau_j, \tau_j + h] \wedge \forall k' \in [j, k-1].(\langle k', \alpha' \rangle \in S^i_\alpha \wedge \alpha' \equiv \top\}, i)$
$b_a \leftarrow min(\{k | \tau_j + l \le \tau_k \le \tau_j + h \wedge \langle k, \beta' \rangle \in S^i_\beta \wedge \beta' \ne \bot\})$
$b_t \leftarrow min(\{k | \tau_j + l \le \tau_k \le \tau_j + h \wedge \langle k, \top \rangle \in S^i_\beta\})$
$b_n \leftarrow \top \text{ if } (\tau_i - \tau_j \ge \Delta^w(\psi)) \wedge \forall k.(\tau_j + l \le \tau_k \le \tau_j + h).\langle k, \bot \rangle \in S^i_\beta$
$\text{if } b_t \ne \emptyset \wedge a_u \ge b_t \quad \mathbf{return}\langle j, \top \rangle$
$\text{else if } (b_a \ne \emptyset \wedge a_a < b_a) \text{ or } b_n = \top \quad \mathbf{return}\langle j, \bot \rangle$
$\text{else} \quad \mathbf{return}\langle j, \alpha \mathcal{U}_{[l,h]} \beta \rangle$
```

The `reduce` function for *until* formulas uses marker values to evaluate the semantics of *until*. `reduce` calculates five marker values: a_a is the earliest step within the time interval where α is known false. a_u is the latest step within the interval that $\alpha \mathcal{U}_{[l,h]} \beta$ would be true if β were true at that step. b_a is the earliest step within the interval at which β is not conclusively false, and b_t is the earliest step within the interval at which β is conclusively true. b_n holds whether the current step i is later than the wait delay and all β values within the interval are false. With these marker variables, reduce can directly check the semantics of *until*, and either return the correct value or the unchanged residue if the semantics are not conclusive with the current history. Reducing *since* formulas works in the same way (using the same marker values) adjusted to past time intervals and utilizing the unbounded past time history values.

Incrementing History Structures. To evaluate past and future-time policies, we must correctly store trace history which can be looked up during a residue reduction. We store the trace history of a formula ϕ in a history structure S_ϕ. This history structure contains a list of residues for the number of steps required to evaluate the top-level formula. History structures are incremented by the function $\texttt{incrS}(S^{i-1}_\phi, \mathbb{S}^i_\phi, \sigma_i, \tau_i, i) = (\bigcup_{r \in S^{i-1}_\phi} \texttt{reduce}(\sigma_i, \tau_i, S^i_\phi, r)) \cup$ $\texttt{reduce}(\sigma_i, \tau_i, S^i_\phi, \langle i, \phi \rangle)$. This function takes the previous step's history structure S^{i-1}_ϕ and the current state σ_i and performs two actions: 1) Adds a residue for the current step i to S^i_ϕ and 2) Reduces all residues contained in S^{i-1}_ϕ using σ_i.

3.3 Algorithm Properties

There are two important properties of `EgMon` which need to be shown. First, *correctness* states that the algorithm's results are correct. That is, that if `EgMon` reports a property violation, the trace really did violate the property. Second, *promptness* requires that the algorithm provide a decision for the given property in a timely fashion. Promptness precisely requires that the algorithm decides satisfaction of the given property at trace position i as soon as there is another trace position j available such that $j \geq i$ and $\tau_j - \tau_i \geq \Delta^w(\varphi)$. The following theorem states that `EgMon` is correct and prompt. It requires the history structures $\mathbb{S}^i_\varphi$ to be consistent at i analogous to the trace σ, τ, that is, the history structures contain correct history of σ till step i.

Theorem 1 (Correctness and Promptness of `EgMon`). *For all $i \in \mathbb{N}$, all formula φ, all time stamp sequences τ and all traces σ it is the case that (1) if $\langle j, \mathsf{f} \rangle \in S^i_\varphi$ then $\sigma, \tau, j \not\models \varphi$ and if $\langle j, \mathsf{t} \rangle \in S^i_\varphi$ then $\sigma, \tau, j \models \varphi$ (Correctness) and (2) if $\tau_i - \tau_j \geq \Delta^w(\varphi)$ then if $\sigma, \tau, j \not\models \varphi$ then $\langle j, \mathsf{f} \rangle \in S^i_\varphi$ and if $\sigma, \tau, j \models \varphi$ then $\langle j, \mathsf{t} \rangle \in S^i_\varphi$ (Promptness) .*

Proof. By induction on the formula φ and time step i. See [18]

We now discuss the runtime complexity of `EgMon` while checking satisfaction of a property φ. For any evaluation position of the trace σ, let us assume we have maximum L positions in σ for which φ has not yet been reduced to a boolean value. Note that the maximum number of positions in that are not yet reduced must be $\frac{\Delta^w(\varphi)}{P}$ where P is the monitor's period. Additionally, for each temporal subformula $\phi_1 \mathcal{U} \phi_2$ of φ, we build history structures that keep track of segments of positions in σ for which ϕ_1 is true. Let us assume we have a maximum of M such segments that are relevant for φ evaluation. Hence, the complexity of `EgMon` is $\mathcal{O}(LM|\varphi|)$.

4 Monitor Implementation and Evaluation

To evaluate the feasibility of our monitoring algorithm for safety-critical real-time systems we have built a real-time CAN monitor on an ARM Cortex-M4 development board. This allowed us to explore the necessary optimizations and features required to perform real-time checking of realistic safety policies.

Challenges. Software for safety-critical embedded systems typically contains more strict design and programming model constraints than less critical software. Two important and common constraints for these systems are avoiding recursion and dynamic memory allocation. As $\alpha\mathcal{VSL}$ is future bounded, we avoid dynamic allocation in our `EgMon` implementation by statically allocating space for the maximum number of entries for our history structures and other temporary data structures. Although `EgMon` is defined recursively, we implement `EgMon` using a traditional iterative traversal of the specification formulas instead.

Discussion. Passive monitoring of running systems requires attention to timing issues with regards to system state sampling. The monitor possesses a copy of the target system state which is updated when a new CAN message is observed. The monitor periodically takes a snapshot of this constantly updated state and uses that snapshot as the trace state which is monitored. Thus, the actual monitored state (*i.e.*, the trace) is a discrete sampling of the monitor's constantly updated system state. The monitor's period must be fast enough that any changes in system state which are relevant to the specification are seen in the sampled trace. To ensure this, the monitor's period should be twice as fast as the shortest CAN message period. If the monitor's period is not fast enough, multiple CAN messages announcing the same system value may end up in the same trace state causing those values to not be seen in the trace. For example, if the monitor is sampling at $2ms$, and three messages announcing the value of property X are received at times $0ms$, $1ms$, and $1.5ms$, only the value announced at $1.5ms$ will be seen in the trace. To avoid this, the monitor would need to run faster than the messages inter-arrival rate (at least $0.5ms$ in this case).

Along with requiring the monitor to sample its trace state fast enough to see all the relevant state changes, the specification time bounds must be a multiple of the monitoring period. This ensures that each time step in the formula is evaluated on the updated information. A simple way of understanding this is to use monitor steps as the temporal bounds instead of explicit time bounds. For example, if the monitor is running at 2ms intervals, we can use $\square_{[0,50]}p$ as an equivalent to $\square_{[0,100ms]}\mathsf{p}$. If a bound is not a multiple of the monitor's interval, then formulas with different bounds can look indistinguishable, *e.g.*, $\square_{[0,5ms]}\mathsf{p}$ is equivalent to $\square_{[1ms,4ms]}\mathsf{p}$ for a $2ms$ monitor. To summarize, using a monitoring period at least twice as fast as the shortest CAN message period (*i.e.*, shortest time between a CAN message retransmission) and only using temporal bound values that are multiples of this period provides intuitive monitoring results.

Hybrid Algorithm (HMon). The eager monitoring algorithm attempts to evaluate specification rules as soon as possible which requires checking properties which may not be fully reducible given the current trace. Continuously attempting to check partially reduced residues which need future information to evaluate requires extra computation. In some cases, the worst-case execution time to eagerly check a property (or set of properties) over a trace may be longer than the monitor's period. In this case the eager algorithm cannot guarantee prompt monitoring since some residues will be left unchecked. This is unacceptable for safety-critical system monitoring, as without a promptness guarantee the results cannot be fully trusted. To enable the benefits of eager checking while avoiding the risks of losing real-time correctness, we can use a hybrid approach (called HMon) which first performs non-eager (conservative) checking and then uses any spare time left in the monitoring period to perform as much eager checking as possible. HMon preserves the monitor's promptness guarantee (all violations will be detected within their delay $\varDelta^w$) even if the `EgMon` cannot finish eager checking the specification. HMon preserves promptness while providing a chance to eagerly detect specification violations. HMon is functionally equivalent to `EgMon`

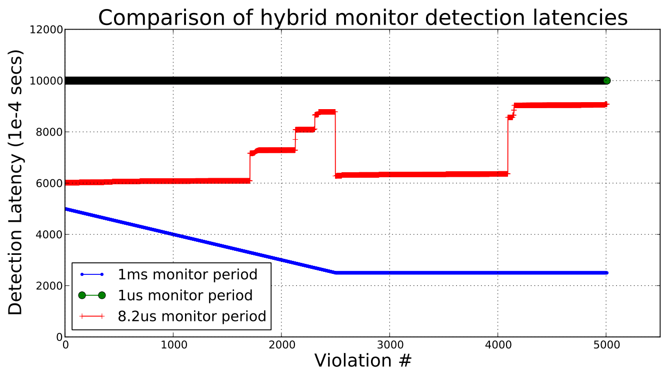

Fig. 2. Detection latencies with different hybrid monitor periods

when it finishes execution before the monitoring period whereas HMon is equivalent to the conservative algorithm when there is no spare execution time to perform eager checks.

Figure 2 shows the violation detection latency for HMon while checking the property $(\mathsf{CruiseActive} \wedge \mathsf{brakePressed}) \rightarrow \square_{[250ms,1s]}(\neg\mathsf{CruiseActive})$ over a synthetic trace using different monitoring periods. With a $1ms$ monitoring period HMon is able to eagerly detect every violation early. For $8.2us$ monitoring period, HMon is able to detect some violations early, but it does not detect all of them right away. For $1us$ monitoring period, HMon only evaluates conservatively due to lack of left-over time, identifying the violations at the promptness limit of $1s$.

We have also implemented HMon in our embedded board. HMon updates the history structures (shared between the conservative and eager checking) and performs a conservative check once every monitoring period. Eager checking is then performed until the next monitor period interrupts it.

4.1 Case Study

This section reports our case study performing real-time monitoring of a CAN network for realistic safety properties. For this case study we have obtained CAN network logs from a series of robustness tests on the ARV which we have replayed on a test CAN bus. This setup helps us show the feasibility of performing external bus monitoring on this class of system with a realistic safety specification.

The logs contain both normal system operation as well as some operation under network-based robustness testing. During robustness testing, the testing framework can intercept targeted network messages on the bus and inject its own testing values. A PC was connected to a PCAN-USB Pro [1] device which provides a USB interface to two CAN connections. One CAN channel was used to replay the logs, while the other was used as a bus logger for analysis purposes.

Requirements documentation for this system was available, so we were able to build a monitoring specification based on actual system requirements. The specification evaluated in the embedded monitor on the test logs are shown in Table 1. This specification was derived from the system requirements based on

Rule #	Informal Rule MTL
0	A feature heartbeat will be sent within every 500ms $\mathsf{HeartbeatOn} \rightarrow \Diamond_{[0,500ms]}\mathsf{HeartBeat}$
1	The interface component heartbeat counter is correct $\mathsf{HeartbeatOn} \rightarrow \mathsf{HeartbeatCounterOk}$
2	The vehicle shall not transition from manual mode to autonomous mode $\neg((\circleddash_{[0,25ms]}\mathsf{IntManualState}) \wedge \mathsf{IntAutoStat})$
3	The vehicle controller shall not command a transition from manual mode to autonomous mode $\neg((\circleddash_{[0,25ms]}\mathsf{VehManualModeCmd}) \wedge \mathsf{VehAutoModeCmd})$
4	The vehicle shall not transition from system off mode to autonomous mode $\neg((\circleddash_{[0,25ms]}\mathsf{IntSDState}) \wedge \mathsf{IntAutoStat})$
5	The vehicle controller shall not command a transition from system off mode to autonomous mode $\neg((\circleddash_{[0,25ms]}\mathsf{VehSDModeCmd}) \wedge \mathsf{VehAutoModeCmd})$

Table 1. Case study monitoring specification

the observable system state available in the testing logs. We note that the safety specification is simple and does not fully exercise the monitor's constraints. Since this specification is small, does not have long duration future-time formulas, and the monitor period is relatively slow ($25ms$) HMon and EgMon function equivalently for the case study – EgMon can always finish within the monitoring period in the worst case. We use the HMon for the study since it acts the same as EgMon when there is excess monitoring time. Using HMon allows us to avoid worrying about the execution time and has little downside.

Table 1 shows the monitored specification. HeartBeatOn is a guard used to avoid false-positive violations during system startup. The system's heartbeat message contains a single heartbeat status bit which we checked directly in Rule #0 to ensure the component is still running (essentially a watchdog message). The heartbeat message also has a rolling counter field. We use the StP to ensure that this counter is incrementing correctly and output this check as the HeartbeakOk proposition which is monitored in Rule #1. We also checked for illegal state transitions. Rules #2 through #5 check both for illegal transition commands from the vehicle controller and actual illegal state transitions in the interface component.

4.2 Monitoring Results

Monitoring the test logs with the above specification resulted in identifying two real violations as well as some false positive violation detections caused by the testing infrastructure. Three different types of heartbeat violations were identified after inspecting the monitor results, with one being a false positive. We also identified infrastructure-caused false-positive violations of the transition rules.

Specification violations. The first violation is a late heartbeat message. In one of the robustness testing logs the heartbeat message was not sent on time, which

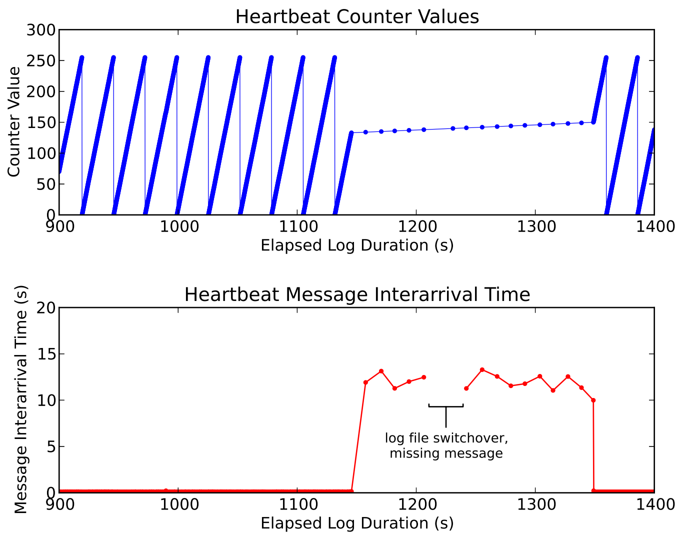

Fig. 3. Heartbeat counter values over time

is clearly a heartbeat violation. Figure 3 shows the heartbeat counter values and the inter-arrival time of the heartbeat messages over time for this violation. We can see here that the heartbeat counter did in fact increment in a valid way, just too slowly. The second violation is on-time heartbeat status message but the heartbeat status field is 0. We do not know from the available documentation whether a bad status in an on-time message with a good counter is valid or not. So without more information we cannot tell whether these violations are false positives or not. This is worthy of further investigation.

False-positive violations. The last type of heartbeat violation is a bad counter. A good rolling counter should increment by one every message up to its maximum (255 in this case) before wrapping back to zero. Every consecutive heartbeat status message must have an incremented heartbeat counter or a violation will be triggered. Figure 4 shows the counter value history for one of the traces with a heartbeat violation caused by a bad counter value. Further inspection of this violation showed that the bad counter values were sent by the testing framework rather than the actual system. In this case, the network traffic the monitor is seeing is not real system state instead are observing messages that are being injected by the testing framework. This is a false positive violation since the violating state is not actual system state.

The monitor also reported violations of the legal transition rules, but these, similar to the heartbeat counter violation, also turned out to be false positives triggered by message injections by the robustness testing harness. Since the monitor checks network state, if we perform testing that directly affects the values seen on the network (such as injection/interception of network messages) we may detect violations which are created by the testing framework rather than

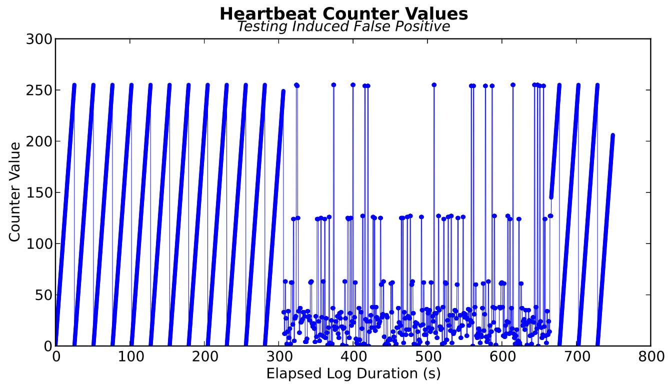

Fig. 4. Bad heartbeat counter values

the system. Information about the test configurations can be used to filter out these types of false positives which arise from test-controlled state. This type of filtering can be automated if the test information can be input to the monitor, either directly on the network (e.g., adding a message value to injected messages) or through a side-channel (i.e., building a testing-aware monitor).

5 Conclusion and Future Work

We have developed a runtime monitoring approach for an ARV system. Instead of instrumentation, we passively monitor the system, generating the system trace from the observed network state. We have developed an efficient runtime monitoring algorithm, `EgMon`, that eagerly checks for violations of properties written in future-bounded propositional MTL. We have shown the efficiency of `EgMon` by implementing it and evaluating it against logs obtained from system testing of the ARV. `EgMon` was able to detect violations of several safety requirements in real-time. We want to further explore runtime monitors executing in multi-core environment to provide increased monitoring power as well as further formalizing the StP (in a domain specific language). Currently, we do not investigate the energy consumption of the runtime monitors. It could be possible that the extra checks required for eager checking might not feasible due to energy consumption restrictions in which case one has to investigate energy-efficient alternatives.

References

1. Pcan-usb pro: Peak system. `http://www.peak-system.com/PCAN-USB-Pro.200.0.html?&L=1`
2. Basin, D., Klaedtke, F., Mller, S., Pfitzmann, B.: Runtime monitoring of metric first-order temporal properties. In: FSTTCS. vol. 8, pp. 49–60 (2008)
3. Basin, D., Klaedtke, F., Zalinescu, E.: Algorithms for monitoring real-time properties. In: Runtime Verification, LNCS, vol. 7186, pp. 260–275 (2012)
4. Bonakdarpour, B., Fischmeister, S.: Runtime monitoring of time-sensitive systems. In: Runtime Verification, LNCS, vol. 7186, pp. 19–33 (2012)

5. Bosch, R.: CAN specification version 2.0 (Sep 1991)
6. Chang, C.L., Lee, R.C.T.: Symbolic Logic and Mechanical Theorem Proving. Academic Press, Inc., Orlando, FL, USA, 1st edn. (1997)
7. Chen, F., Rosu, G.: Towards monitoring-oriented programming: A paradigm combining specification and implementation. Electronic Notes in Theoretical Computer Science 89(2), 108 – 127 (2003)
8. Chowdhury, O., Jia, L., Garg, D., Datta, A.: Temporal mode-checking for runtime monitoring of privacy policies. In: CAV (2014)
9. Clarke, E.M., Wing, J.M.: Formal methods: state of the art and future directions. ACM Comput. Surv. 28, 626–643 (December 1996)
10. Dokhanchi, A., Hoxha, B., Fainekos, G.: On-line monitoring for temporal logic robustness. In: Runtime Verification (2014)
11. Donzé, A., Ferrére, T., Maler, O.: Efficient robust monitoring for stl. In: CAV (2013)
12. Garg, D., Jia, L., Datta, A.: Policy auditing over incomplete logs: Theory, implementation and applications. In: Proceedings of the 18th ACM Conference on Computer and Communications Security. pp. 151–162. ACM (2011)
13. Goodloe, A., Pike, L.: Monitoring distributed real-time systems: a survey and future directions (NASA/CR-2010-216724) (July 2010)
14. Havelund, K., Rosu, G.: Synthesizing monitors for safety properties. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 342–356. TACAS '02, Springer-Verlag, London, UK, UK (2002)
15. Havelund, K., Rosu, G.: Efficient monitoring of safety properties. Int. J. Softw. Tools Technol. Transf. 6(2), 158–173 (Aug 2004)
16. Heffernan, D., MacNamee, C., Fogarty, P.: Runtime verification monitoring for automotive embedded systems using the iso 26262 functional safety standard as a guide for the definition of the monitored properties. Software, IET 8(5), 193–203 (2014)
17. Kane, A., Fuhrman, T., Koopman, P.: Monitor based oracles for cyber-physical system testing: Practical experience report. In: Dependable Systems and Networks (DSN). pp. 148–155 (2014)
18. Kane, A., Chowdhury, O., Koopman, P., Datta, A.: A case study on runtime monitoring of an autonomous research vehicle (arv) system. Tech. rep., CMU (2015)
19. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-mac: A runtime assurance approach for java programs. Formal methods in system design 24(2), 129–155 (2004)
20. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. 2, 255–299 (October 1990)
21. Nickovic, D., Maler, O.: Amt: A property-based monitoring tool for analog systems. In: Formal Modeling and Analysis of Timed Systems (2007)
22. Pellizzoni, R., Meredith, P., Caccamo, M., Rosu, G.: Hardware Runtime Monitoring for Dependable COTS-Based Real-Time Embedded Systems. 2008 Real-Time Systems Symposium pp. 481–491 (Nov 2008)
23. Reinbacher, T., Fgger, M., Brauer, J.: Runtime verification of embedded real-time systems. Formal Methods in System Design pp. 1–37 (2013)
24. Rosu, G., Havelund, K.: Rewriting-based techniques for runtime verification. Automated Software Engineering 12(2), 151–197 (2005)
25. Thati, P., Roşu, G.: Monitoring algorithms for metric temporal logic specifications. Electron. Notes Theor. Comput. Sci. 113, 145–162 (Jan 2005)