Preprint: 15th International Conference on Runtime Verification (RV 2015)

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, IN
akane@cmu.edu, ochowdhu@purdue.edu, danupam@cmu.edu, koopman@cmu.edu

Abstract. Although runtime monitoring is a promising technique to improve the verification of complex safety-critical systems, the general design trend towards utilizing black-box commercial-off-the-shelf (COTS) components means that these systems are not always amenable to instrumentation which is commonly used to produce the relevant events necessary for checking the desired properties. In this paper, we develop an online, real-time monitoring approach that targets an autonomous research vehicle (ARV) system and recount our experience with it. To avoid instrumentation we passively monitor the target system by generating high-level property constructs (*i.e.*, 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 a future-bounded, propositional metric temporal logic. We show the efficacy of `EgMon` by implementing it 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 system safety and correctness properties of safety-critical embedded systems in the face of increasing design complexity. In RV, a runtime monitor observes the execution of the system in question and checks for violations of some well-defined properties. When the monitor detects a violation, it can notify 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 (COTS) components, some of which may be provided as black-box systems, from multiple vendors. These systems are also often hard real-time systems which leads to additional constraints on system monitoring [11]. This type of system architecture is incompatible with many existing runtime monitoring

techniques, which often require program or system instrumentation [4,7,13,17] to obtain the relevant events or policy-constructs (*e.g.*, propositions) necessary to check for violations. Instrumenting systems without access to component source code 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 policy constructs. Instead of instrumentation, we obtain the relevant information for monitoring the ARV system through passive observation of the system's broadcast buses. Controller area network (CAN) is a common and standard broadcast bus for ground vehicles which is the primary system bus in the ARV. We can obtain useful amounts of system state relevant to monitoring the system safety specification by observing the data within the CAN messages being broadcast between system components. However, before we can start monitoring the ARV system, we need a component, which we call the SF Map, that observes messages transmitted on the bus, decoding them into propositions relevant to monitoring which are fed into the monitor. This acts similarly to the low-level specification and filter/event recognizers from MaC [17]. We want to emphasize that the limits of external observability can cause significant challenges in designing the SF Map when considering the state available from the system messages and the necessary atomic policy-constructs [15].

Specification logic. To obtain the relevant safety requirements and invariants for monitoring the ARV system we consulted the safety requirements of the ARV system. We observed that 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 such as "*the system must perform action a within t seconds of event e*" are common, for example: *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 metric temporal logic (MTL) [18] in which the bound associated with the future temporal operators must be finite.

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 relevant propositions to either true or false) and a MTL formula and eagerly checks the state trace for violations. Some of the 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 straightforward formula-rewriting based simplifications when possible (leaving a partially reduced formula when future input is required). This eager nature of the algorithm is helpful because detecting a violation earlier provides the system more time to attempt a recovery. We have also proved the correctness of our algorithm.

Empirical evaluation. We have implemented `EgMon` 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`

has moderate monitoring overhead and detected several safety violations in our experimental evaluation.

2 Background and Existing Work

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

Monitoring safety-critical embedded systems. Goodloe and Pike present a thorough survey of monitoring distributed real-time systems in [11]. 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 Goodloe and Pike's 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 few (essentially no) changes to the target system architecture. We utilize this architecture for our monitoring framework.

Monitors. Our monitoring algorithm is similar to existing dynamic programming and formula-rewriting based algorithms. 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.

Dynamic programming monitors. Our monitoring algorithm is inspired by the algorithms *reduce* [10] and *prècis* [8], adjusted for propositional logic and eager checking. The structure of our algorithm is based on *reduce*. We utilize an iterative, formula-rewriting based algorithm targeted at both offline log analysis as well as runtime monitoring. Both *reduce* and *prècis* can handle future incompleteness but *reduce* also considers incompleteness for missing information which we do not consider. *prècis* and `EgMon` both require the input trace to contain complete information.

The NASA PathExplorer project has led to both a set of dynamic programming-based monitoring algorithms as well as some formula-rewriting based algorithms [13] for past-time LTL. These dynamic programming algorithms require checking the trace in reverse (from the end to the beginning) which makes them somewhat unsuitable for online monitoring [12]. The formula rewriting algorithms utilize the Maude term rewriting engine to efficiently monitor specifications through formula rewriting [21]. Thati and Roşu [23] describe an dynamic programming algorithm for monitoring MTL which is based on resolving the past and deriving the future. They perform 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 state to be stored than some other algorithms (because formulas grow in size as they are rewritten), including this work.

Embedded Monitors. Heffernan et. al. present a monitor for automotive systems using ISO 26262 as a guide to identify the monitored properties in [14]. They monitor past-time linear temporal logic (LTL) formulas and obtain system state from target system buses (CAN in their example). Our semi-formal interface is similar to their "filters" used to translate system state to the atomic propositions that are monitored. Their motivation and goals are similar to ours, but they use on-chip system-on-a-chip based monitors which utilize instrumentation to obtain system state, which is not suitable for monitoring black-box systems. Reinbacher et. al. present an embedded past-time MTL monitor in [20] 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 [19]. They generate FPGA monitors that passively observe PCI-E buses to verify system properties. This is a similar architecture to ours, but they only check past-time LTL and regular expressions so they cannot perform eager checking.

Basin et. al. compare algorithms for monitoring real-time MTL properties in [3]. Our monitoring algorithm works similarly to their point-based monitoring algorithm, iteratively calculating truth values over the formula structure using history lists. Though they discuss the use of delay queues to monitor future-time properties (and thus do not eagerly check future-time formulas), we could integrate their algorithm into our eager monitoring framework.

3 Monitoring Architecture

Many modern safety-critical embedded systems are designed as distributed systems to provide the ability to meet the necessary reliability, fault-tolerance, and redundancy. The individual system components are typically connected via a network bus, of which there are many different common types [22].

Controller Area Network. Controller Area Network is a widely used automotive network developed by Bosch in the 1980s [5]. In this work we focus primarily on monitoring CAN because it is a common automotive bus which typically conveys enough of the state we wish to observe without instrumentation. CAN is an event-based broadcast network with data rates up to 1Mb/s (although usually used at 125-500kbps). 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. Because of this our monitoring scheme is based on a time-triggered, network sampling model which allows it to monitor time-triggered networks as well.

As mentioned previously, existing runtime monitoring techniques which rely on code instrumentation are not directly applicable to systems with black-box components. Instead of instrumentation, we propose a passive external bus-monitor which only checks system properties that are observable by passively

observing system state from an existing broadcast bus. We focus on ground vehicles and CAN buses specifically in this work, but other similar systems and broadcast networks can also be monitored using this approach. For example, safety-critical buses in star configurations which don't have a single bus line that can observe all traffic can be monitored by placing the monitor in the network's hub or by connecting multiple buses lines directly to the monitor.

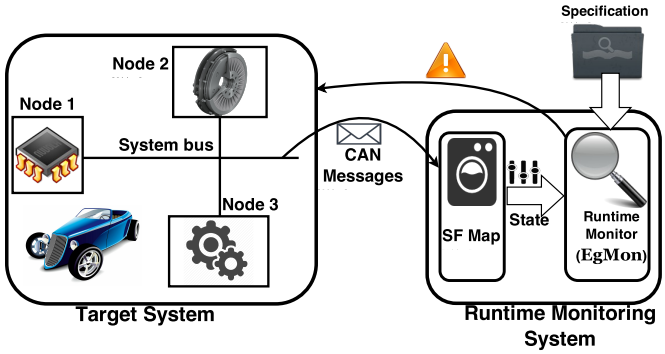

Fig. 1. External monitor architecture outline

An outline of our monitor architecture is shown in Figure 1. The monitor is connected to the target system as an additional passive node on its broadcast bus. Different systems have varying specification needs which can not always be easily met within a formal specification language. In order to provide flexibility to map system state onto the formal specification language (in our case, in propositions) we provide a SF Map interface which defines the mapping between the observed system state and the monitored specification. This type of interface is common in monitors for real systems, including MaC's filters [17] and the AP evaluation filter from [14]. The monitor's SF Map generates the system trace by building the necessary propositions based on the observed bus traffic. This generated trace is provided to the monitoring algorithm which checks it against the given system specification, outputting whether the trace violated or satisfied the specification at each trace step (subject to delays waiting on future state).

This architecture separates the lower-level system dependent configuration from the high-level system specification in a way similar to the architecture used in MaC [17]. This allows us to utilize a core formal monitoring algorithm and framework with any system where an SF Map can be used to create a system trace. Separating the system dependent and system-independent aspects of the monitor allows the high level system requirements be somewhat abstracted away from the implementation. This helps keep the monitoring specification matched closer to the system requirements documentation which usually does not include low-level implementation details. This architecture also makes the monitor more robust to changes in the target system. If the system changes in a way that affects the monitoring-relevant messages, only the SF Map configuration needs to be updated, rather than the monitor itself.

4 Monitoring Algorithm

For monitoring our desired ARV system so that it adheres to its specification, we need an algorithm which incrementally checks explicit time specifications (*i.e.*, propositional metric-time temporal logic [18]) over finite 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 boolean value using formula simplifications (*e.g.*, $a \wedge \mathit{false} \equiv \mathit{false}$). Many of the existing algorithms for evaluating formulas like $\Diamond_{[l,h]} a \vee b$ (read, 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 and see whether it can reduce the whole formula to boolean.

4.1 Specifications

Many safety specification rules for ARV-like systems require explicit time bounds to ensure timely behavior, so a specification language with explicit time bounds is important. For instance, "*cruise control shall disengage for 250ms within 500ms of the brake pedal being depressed*" ($\mathsf{brakeDepress} \rightarrow \Diamond_{[0,500]} \Box_{[0,250]} \neg \mathsf{CruiseEng}$). 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 [18]). The syntax of $\alpha\mathcal{VSL}$ is as follows:

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

Syntax. $\alpha\mathcal{VSL}$ has boolean true (*i.e.*, `t`), atomic propositions `p`, logical connectives (*i.e.*, $\neg, \vee$), past temporal operators *since* and *previously* ($\mathcal{S}, \ominus$), and future temporal operators *until* and *next* ($\mathcal{U}, \bigcirc$). The temporal bound $\mathbb{I}$ of form $[l, h]$ ($l \leq h$ and $l, h \in \mathbb{N}$) associated with the future temporal operators must be finite. Specification propositions `p` come from a finite set of atomic propositions provided in the system trace by the SF Map. These propositions are derived from the observable system state and represent specific system properties, for instance, proposition speedLT40mph could describe whether the vehicle speed is less than 40mph.

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 SF Map, to either `t` or `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 $\alpha\mathcal{VSL}$ are standard for MTL, *e.g.*, [2].

Auxiliary notions. Before we go any further, we 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, residual of a formula might not be boolean because of future states not yet being available. 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 policy summary for future time formulas which cannot be evaluated due to incomplete information.

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.*, $\Delta^w(\Diamond_{[0,3]}\mathtt{p}) = 3$). The length of a formula φ, denoted $|\varphi|$, returns the total number of subformulas of φ. The function `tempSub` takes as input a formula φ, and returns all the temporal subformulas ϕ of φ and strict subformulas of ϕ.

4.2 `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 policy 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 policy φ periodically at every trace sample step. When the policy cannot be decided at a given step (*e.g.*, it requires future state to evaluate), the remaining policy 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^i_ϕ. We use $\mathbb{S}^i_\varphi$ to denote the set of history structures for all temporal subformula of φ, *i.e.*, $\mathbb{S}^i_\varphi = \bigcup_{\phi \in \mathtt{tempSub}(\varphi)} S^i_\phi$.

The high level algorithm `EgMon` is shown in Figure 2. First, all the necessary history structures S_ϕ are identified using `tempSub`(φ) 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

1: For all recognized formulas $\phi \in \texttt{tempSub}(\varphi)$: $S^{-1}_{\varphi_1} \leftarrow \emptyset$
2: $i \leftarrow 0$
3: **loop**
4: Obtain next trace step (σ_i, τ_i)
5: **for** every $\phi \in \texttt{tempSub}(\varphi)$ in increasing size **do**
6: $S^i_\phi \leftarrow \texttt{incrS}(S^{i-1}_\phi, \mathbb{S}^i_\phi, \sigma_i, \tau_i, i)$
7: **end for**
8: $S^i_\varphi \leftarrow \texttt{incrS}(S^{i-1}_\varphi, \mathbb{S}^i_\varphi, \sigma_i, \tau_i, i)$
9: **for** all $\langle j, \texttt{f} \rangle \in S^i_\varphi$ **do**
10: `Report violation on` σ `at position` j
11: **end for**
12: $i \leftarrow i + 1$
13: **end loop**

Fig. 2. `EgMon` Algorithm

be their subformula). Each structure is updated using $\texttt{incrS}(S^{i-1}_\phi, \mathbb{S}^i_\phi, \sigma_i, \tau_i, i)$ 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 policy that is being monitored – the policy's structure is updated with $\texttt{incrS}(S^{i-1}_\varphi, \mathbb{S}^i_\varphi, \sigma_i, \tau_i, i)$. Once updated, this structure contains the evaluation of the top-level policy. The algorithm reports any identified policy 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 policy is treated exactly the same as any temporal subformula would be (which follows from the fact that the top-level policy contains an implicit *always* $\Box$). The history structure updates for the top-level policy are separated in the algorithm description for clarity only. The only difference between the top-level policy and other temporal subformula is that violations are reported for the top-level policy.

Reducing Residues. `EgMon` works primarily by reducing policy residues down to truth values. Residues are reduced by the $\texttt{reduce}(\sigma_i, \tau_i, \mathbb{S}^i_\varphi, \langle j, \phi \rangle_\varphi)$ function, which uses the current state (σ_i, τ_i) and the stored history in $\mathbb{S}^i_\varphi$ to rewrite the policy ϕ to a reduced form, either a truth value or a new policy 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 policy is available in the history and current state. Future-time policies may be fully-reducible if enough state information is available. If a future-time policy 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. Figure 3 shows the reduction algorithm for *until* temporal formula. Reducing *since* formulas is essentially the same except with reversed minimum/-maximums and past time bounds.

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

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

Fig. 3. Definition of `reduce` for *until* formulas

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 policy 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 policy. 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 and performs two actions: 1) Adds a residue for the current step i to S^{i-1}_ϕ and 2) Reduces all residues contained in S^{i-1}_ϕ with the current state.

4.3 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 policy violation, the trace really did violate the policy. Second, *prompt-*

ness requires that the algorithm provide a decision for the given policy in a timely fashion (*i.e.*, with t such that $t \leq \Delta^w(\varphi)$). Promptness requires that the algorithm decide satisfaction as soon as it is guaranteed to be possible.

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. This means that the history structures contain correct history of the trace 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 $\langle j, \mathtt{f}\rangle \in S^i_\varphi$ then $\sigma, \tau, j \not\models \varphi$ and if $\langle j, \mathtt{t}\rangle \in S^i_\varphi$ 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 $\langle j, \mathtt{f}\rangle \in S^i_\varphi$ and if $\sigma, \tau, j \models \varphi$ then $\langle j, \mathtt{t}\rangle \in S^i_\varphi$ (Promptness) .*

Proof. By induction on the policy formula φ and time step i. See [16]

5 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.

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 not using dynamic memory allocations. Common safety-critical coding guidelines discourage or prohibit dynamic memory allocation to avoid memory leaks. Because our specification language is bounded, we can 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. Recursion is also usually prohibited because it can be difficult to guarantee a maximum stack depth when using recursion. Although `EgMon` utilizes recursion extensively, we can implement `EgMon` using a traditional iterative traversal of the specification formulas instead.

Hybrid Algorithm. Our eager monitoring algorithm attempts to evaluate specification rules as soon as possible, but this requires checking trace properties which may not be fully reducible given the current trace. These unfinished formula reductions require extra computation time, and in practice the majority of the policy reductions performed by `EgMon` will be these eager reductions which may not fully reduce.

While early detection of violations can be useful, there are situations where eagerly checking an entire target specification may require more computation than is available from the monitor in a given period.

To enable the benefits of eager checking while avoiding the risks of losing real-time correctness, we have implemented a hybrid eager monitoring algorithm which performs non-eager (conservative) checking first and uses any spare time

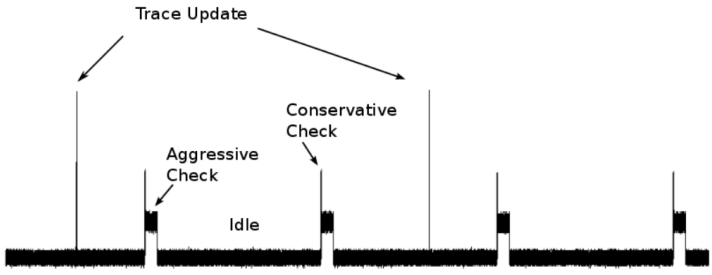

Fig. 4. Oscilloscope capture of embedded monitor task execution

to eagerly check the remaining monitor residues. Conservative `EgMon` monitoring is performed by only checking residues which are older than their formula delay, which guarantees that these residue will be reduced at their first evaluation. Under our periodic sampling design, each step of conservative monitor only requires updating the history structures and checking a single residue (the oldest remaining one) for each specification policy. This conservative check can be done quickly at each period, leaving any extra time until the next period for eager checking. This provides a conservative monitoring guarantee (*i.e.*, the specification is checked within a known promptness delay) while also allowing the monitor to eagerly check as much of the specification as possible.

We have implemented the hybrid monitoring algorithm in our embedded monitor. The monitor updates the history structures (shared between the conservative and eager checking) and performs a conservative check once every monitoring period. It then uses the idle time between periods to perform eager checking of any remaining unchecked specification properties. Figure 4 shows the execution of the embedded monitor instrumented to output the currently executing task to an oscilloscope. The residue checks run twice per trace update due to the monitor configuration used during the test, but this is not required for correct monitoring. This task output was captured while monitoring the specification used in the case study (see Section 5.1) plus another 200 time-step *eventually* rule which was never satisfied. The rule never being satisfied means that the monitor performed an eager check of all 200 residues for this rule at every step (i.e., since they were never satisfied, they could never be reduced early). Even with this excess computation there was still a large portion of extra idle time – 23ms of the 25ms monitoring loop was spent idle. This shows the eager checking finished reasonably quickly and the monitor could handle much longer formula durations or more complex formulas before the execution time becomes bad enough to require the hybrid algorithm for correctness guarantees.

5.1 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 for the monitor to check. This setup helps us show

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 $\neg((\ominus_{[0,25ms]} \mathsf{IntManualState}) \wedge \mathsf{IntAutoStat})$
3	The vehicle controller shall not command a transition from manual mode to autonomous mode $\neg((\ominus_{[0,25ms]} \mathsf{VehManualModeCmd}) \wedge \mathsf{VehAutoModeCmd})$
4	The vehicle shall not transition from system off mode to autonomous mode $\neg((\ominus_{[0,25ms]} \mathsf{IntSDState}) \wedge \mathsf{IntAutoStat})$
5	The vehicle controller shall not command a transition from system off mode to autonomous mode $\neg((\ominus_{[0,25ms]} \mathsf{VehSDModeCmd}) \wedge \mathsf{VehAutoModeCmd})$

Table 1. Case study monitoring specification

the feasibility of performing external bus monitoring on this class of system with real safety specifications.

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 the observable system state available in the testing logs.

Rule #0 is a heartbeat detection which ensures that the interface component is still running (essentially a watchdog message). Rule #1 is a second component of this check. The system's heartbeat message contains a single heartbeat status bit which we checked directly in Rule #0, but the message also has a rolling counter field. We use the SF Map to ensure that this counter is incrementing correctly and output this check as the HeartbeakOk predicate which is checked 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.

5.2 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

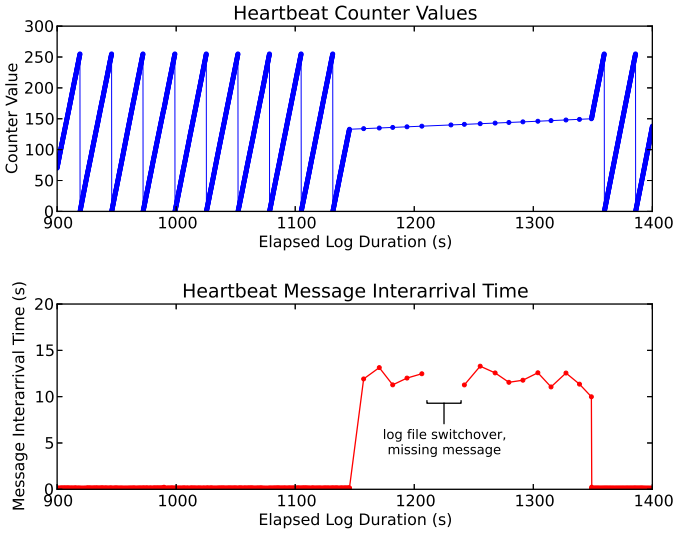

Fig. 5. Heartbeat counter values over time

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 is clearly a heartbeat violation. Figure 5 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 6 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 but actually it is messages being injected by the testing framework. This is not a real violation (since the violating state is not the actual system state), and so we consider this a false positive violation.

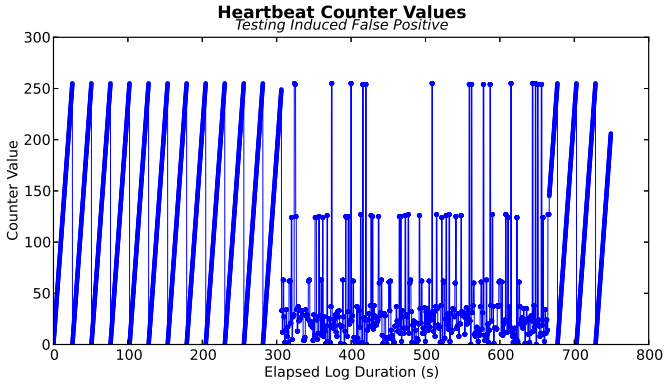

Fig. 6. Bad heartbeat counter values

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 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).

6 Conclusion

We have developed a runtime monitoring approach for an autonomous research vehicle. Rather than instrumenting the target system, 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 our 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.

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, Lecture Notes in Computer Science, vol. 7186, pp. 260–275. Springer Berlin Heidelberg (2012)

4. Bonakdarpour, B., Fischmeister, S.: Runtime monitoring of time-sensitive systems. In: Runtime Verification, Lecture Notes in Computer Science, vol. 7186, pp. 19–33. Springer Berlin / Heidelberg (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: Computer Aided Verification. Springer International Publishing (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. 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)
11. Goodloe, A., Pike, L.: Monitoring distributed real-time systems: a survey and future directions (NASA/CR-2010-216724) (July 2010)
12. 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)
13. Havelund, K., Rosu, G.: Efficient monitoring of safety properties. Int. J. Softw. Tools Technol. Transf. 6(2), 158–173 (Aug 2004)
14. 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)
15. Kane, A., Fuhrman, T., Koopman, P.: Monitor based oracles for cyber-physical system testing: Practical experience report. In: Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on. pp. 148–155 (2014)
16. 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), see `http://users.ece.cmu.edu/~koopman/thesis/kane.pdf` temporarily for proofs
17. 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)
18. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. 2, 255–299 (October 1990)
19. 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)
20. Reinbacher, T., Fgger, M., Brauer, J.: Runtime verification of embedded real-time systems. Formal Methods in System Design pp. 1–37 (2013)
21. Rosu, G., Havelund, K.: Rewriting-based techniques for runtime verification. Automated Software Engineering 12(2), 151–197 (2005)
22. Rushby, J.M.: Bus architectures for safety-critical embedded systems. In: Embedded Software. pp. 306–323. EMSOFT '01, Springer-Verlag, London, UK, UK (2001)
23. Thati, P., Roşu, G.: Monitoring algorithms for metric temporal logic specifications. Electron. Notes Theor. Comput. Sci. 113, 145–162 (Jan 2005)