

# Differentiated Monitor Generation for Real-Time Systems

Behnaz Rezvani<sup>a</sup> and Cameron Patterson<sup>b</sup>

*Bradley Dept. of Electrical and Computer Engineering  
Virginia Tech, Blacksburg, VA, U.S.A.*

**Keywords:** Runtime Verification, Monitor, Timing Constraints, Real-Time Systems, Formal Analysis.

**Abstract:** Safety-critical real-time systems require correctness to be validated beyond the design phase. In these systems, response time is as critical as correct functionality. Runtime verification is a promising approach for validating the correctness of system behaviors during runtime using monitors derived from formal system specifications. However, practitioners often lack formal method backgrounds, and no standard notation exists to capture system properties that serve their needs. To encourage the adoption of formal methods in industry, we present GROOT, a runtime monitoring tool for real-time systems that automatically generates efficient monitors from structured English statements. GROOT is designed with two branches, one for functional requirements and one for specifications with metric time constraints, which use appropriate formalisms to synthesize monitors. This paper introduces TIMESPEC, a structured English dialect for specifying timing requirements. Our tool also automates formal analysis to certify the C monitors' construction. We apply GROOT to timing specifications from an industrial component and a simulated autonomous system in Simulink.

## 1 INTRODUCTION


Runtime verification (RV) is a dynamic verification technique that uses monitors to observe, validate and sometimes correct system behaviors during execution (Bartocci et al., 2018). Monitors are typically derived from system requirements expressed in a formal language such as linear temporal logic (LTL) (Pnueli, 1977). RV can be used for debugging during development or for high-assurance applications post-deployment. Although plenty of RV frameworks exist that target different languages and applications (Falcone et al., 2021), only a few have an explicit notion of time and support real-time systems monitoring.


Ensuring the correct behavior of real-time systems is challenging, as they must satisfy both functional and non-functional requirements containing timing constraints. However, the use of formal methods to support these requirements is often daunting for practitioners. It is a significant issue as it requires verification engineers to have a background in formal methods and to learn the syntax of each monitoring tool, which is often domain-specific. It is noteworthy that timing requirements are inherently different from functional requirements and are repetitive, allowing

the reuse of the same monitor. Accordingly, using distinct formalisms for capturing functional and timing specifications may minimize monitors' overhead.

To address the above issues, we propose a dual monitoring architecture for real-time embedded systems named GROOT (Generalized Runtime mOni-toring Tool) that automatically generates stand-alone non-intrusive formally verified monitors from structured English statements. Our framework uses distinct front ends, intermediate forms and formal verification techniques to translate pseudo-English functional and timing specifications into monitors written in C. To the best of our knowledge, GROOT is the first RV tool that employs different formalisms for functional and timing requirements of real-time systems.

At the front end of the functional requirement flow, the NASA FRET tool (Giannakopoulou et al., 2020) captures structured English properties and generates the equivalent LTL formulas. For timing constraints, we propose a structured English language called TIMESPEC to express time limits, which are then translated into timed automata (Alur and Dill, 1994). GROOT monitors are executed outside the software/hardware system, and the application is considered a black box. The monitors are automatically created and confirmed correct by automated formal analysis. Processing the monitor's inputs and manag-

<sup>a</sup>  <https://orcid.org/0000-0002-1947-1764>

<sup>b</sup>  <https://orcid.org/0000-0003-2482-5261>

ing the violations are handled by separate modules to keep the monitor structure simple.

Timing requirements are essential for both software and hardware applications, as they refer to the specific time intervals or deadlines that must be met for the application to function correctly. In software, specific tasks must be completed within a certain time frame to ensure safety and reliability, while in hardware, inputs and outputs must be precisely timed to ensure correct operation. To demonstrate the usefulness of GROOT generated monitors, examples from both software and hardware are presented.

For the sake of brevity, this paper focuses exclusively on the TIMESPEC flow of GROOT, and the main contributions are:

- *TIMESPEC templates*: This paper presents several structured English statements to collect various timing bounds, providing practitioners with a practical way to capture timing requirements without needing a formal methods background.
- *Adding metric time constraints*: GROOT’s framework fully automates the monitor generation procedure from TIMESPEC requirements, reducing the need for manual intervention.
- *Formally verified monitors*: GROOT also provides an automated process for formally verifying the correctness of each monitor implementation.

## 2 TIMESPEC OVERVIEW

To make RV more accessible to practitioners, this paper proposes a structured English dialect called TIMESPEC to capture timing specifications. This vocabulary supports properties containing metric time bounds by defining a set of property patterns for requirements sharing similarities.

### 2.1 Pulse Duration

The timing of clocks and signals in digital systems and interfaces plays a critical role in ensuring the correct behavior of the overall design. In particular, the pulse width of certain signals must fall within a specific range to trigger the desired events. To address this type of requirement, a pulse duration template specifies the pulse width of a signal or period and duty cycle of a clock signal. Table 1 shows the supported values for each field of this template. TYPE indicates the type of pulse duration requirement. SIGNAL specifies the clock signal or the source of events to be monitored. The TIMING\_CONSTRAINT value depends on the specification constraint. Each time

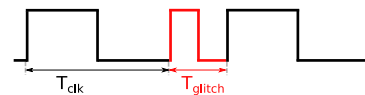


Figure 1: Clock signal including a glitch.

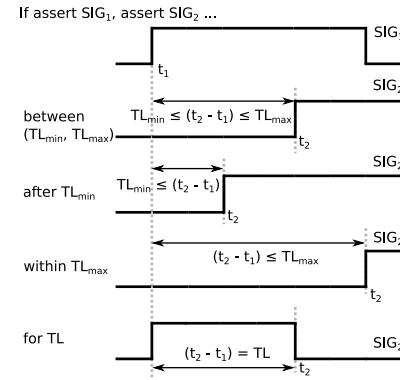


Figure 2: Timing diagram of TIMESPEC causality template for specific ACTIONS and SIGNALS.

limit consists of a numeric value and its corresponding time unit. The duty cycle bounds are typically expressed as a percentage, unlike other TYPES written in an absolute format such as nanoseconds (ns). Figure 1 demonstrates a clock signal (clk) with specific period ( $T_{clk}$ ), say 50 ms. If this clock has any glitches to be detected, a monitor can be created by GROOT using a TIMESPEC requirement similar to “P1: Period of clk should be 50 ms”.

### 2.2 Causality

In communication protocols such as handshaking, events often have causal relationships, where one event may trigger another with a constraint on the elapsed time between the two. To capture this behavior, TIMESPEC uses a causality template described in Table 2. For example, the requirement “P2: If deassert RESET, start CLK within 3 us” indicates the clock signal of a digital system must start within a specific time after the reset signal is deasserted. The ACTION field includes asserting/deasserting signals or starting a clock, with the SIGNAL field indicating the events involved. TIMING\_CONSTRAINT expresses relationships between events based on time bounds. The causality relationship can be visualized with a timing diagram, as shown in Figure 2 to aid practitioners in understanding this template.

## 3 GROOT ARCHITECTURE

Konrad and Cheng proposed a structured English grammar to formally capture requirements (Konrad

Table 1: TIMESPEC pulse duration template.

| Constraint Type                 | Pulse Duration  |
|---------------------------------|---|
| Template                        | TYPE of SIGNAL should be TIMING_CONSTRAINT.                           |
| TYPE                            | active_pulse_width, period, duty_cycle                                |
| SIGNAL                          | Source of events such as: CLK, RESET                                  |
| TIMING_CONSTRAINT               | $TL, \geq TL_{min}$ and $\leq TL_{max}, \geq TL_{min}, \leq TL_{max}$ |
| Time limits (value + time unit) | $TL_{min}$ (lower bound), $TL_{max}$ (upper bound), TL (duration)     |
| Time unit                       | %, ns, us, ms, s, min, h  |

Table 2: TIMESPEC causality template.

| Constraint Type                 | Causality   |
|---------------------------------|---|
| Template                        | If ACTION SIGNAL, ACTION SIGNAL TIMING_CONSTRAINT.                                |
| ACTION                          | assert, deassert, start   |
| SIGNAL                          | Source of events such as: CLK, START  |
| TIMING_CONSTRAINT               | between $TL_{min}$ and $TL_{max}$ , after $TL_{min}$ , within $TL_{max}$ , for TL |
| Time limits (value + time unit) | $TL_{min}$ (lower bound), $TL_{max}$ (upper bound), TL (duration)                 |
| Time unit                       | ns, us, ms, s, min, h   |

and Cheng, 2005). However, temporal logics are declarative while monitors require an operational description such as automata to be implemented effectively. An overview of timed automaton (TA) will be given next, followed by GROOT's monitor automata and its synthesis procedure.

### 3.1 Timed Automaton

A TA is a finite automaton equipped with a finite set of real-valued clocks for modeling timed behaviors of real-time systems (Alur and Dill, 1994). TA can be considered as a directed graph, where the nodes and edges are the states and transitions, respectively. Clock constraints guard states and transitions to restrict the behavior of the automaton.

The semantics of TA is a timed transition system that specifies how the automaton behaves over time in response to events and clock constraints. Figure 3 represents a TA for requirement P2 from Section 2.2. The state  $q_0$  is the initial state while  $q_0$  and  $q_2$  are the accepting states (denoted as double green circles). The events include “ $e_0(e_2)$ : deasserting (asserting) RESET” and “ $e_1$ : starting CLK”. This TA has one clock  $x$ , and the guard  $x \leq 3$  is assigned to the non-accepting state  $q_1$  to limit its duration in  $q_1$ . The TA's clock is assumed to be increased with a pace of 1 us.

When RESET is deasserted, the TA resets the clock ( $x := 0$ ) and moves to  $q_1$ , where it can stay for

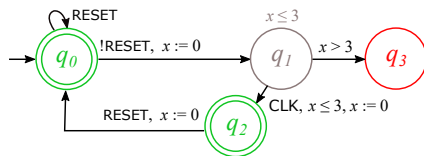


Figure 3: Property “P2: If deassert RESET, start CLK within 3 us” illustrated by a TA.

a maximum of 3 units. If  $e_1$  does not occur before passing 3 units, the TA moves to the non-accepting state  $q_3$  (marked as a red circle), which indicates a violated behavior. If  $e_1$  does occur within 3 units, the TA moves to  $q_2$  and the clock is reset. The accepting state  $q_2$  has no constraint, which allows the TA to remain in this state until  $e_2$  occurs. This TA is deterministic as there is, at most, one transition for any given state and set of clock values. This example demonstrates that a TA is a straightforward and expressive structure for capturing the timing aspects of a system.

### 3.2 Monitor Structure

It is paramount to approach the addition of monitors carefully to avoid compromising system performance. To minimize overhead and ensure isolation, GROOT monitors are performed outside the system and the application is treated as a black box. The monitors are kept simple by using separate external modules to translate inputs into relevant Boolean atomic propositions (APs) and handle any necessary actions in case of violation. This simplicity makes the monitor automata suitable for automated formal analysis.

Figure 4 depicts the monitor code structure. The block Main and its components are automatically generated by GROOT and will be explained later. It is as-

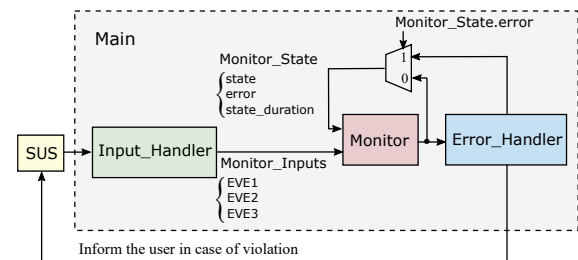


Figure 4: Monitoring process structure.

sumed that Main and the system under scrutiny (SUS) share the same time reference and Monitor is called periodically based on the SUS time signals. If the inputs have a different time base, a separate clock signal can be used for Main to invoke Monitor whenever new data are available. The Input\_Handler module processes the inputs and extracts the necessary information from SUS signals to create events (i.e., APs). These APs are stored in the Monitor\_Inputs structure and updated exclusively by Input\_Handler. The Monitor\_State structure captures the Monitor’s current status, while the counter variable state\_duration stores the number of transitions from the current state to itself. In the event of a violation, Monitor prompts Error\_Handler to notify SUS about the error and reset Monitor\_State for the subsequent observation. This monitoring structure is applied to both functional and timing requirements, with the only difference being the Monitor architecture.

### 3.3 Monitor Synthesis Workflow

GROOT’s separation of functional properties and timing constraints reflects the common practice in hardware engineering where functionality and timing are treated separately in documentation and design. Using a single logic or automata for both functional and timing requirements can result in complications that hinder formal analysis. The TIMESPEC flow utilizes a timing-oriented language, automata and monitor structure to focus specifically on timing constraints.

As shown in Figure 5, GROOT takes a TIMESPEC statement as input and outputs an executable monitor and formal specifications for verifying monitor construction. The flow is composed of three steps.

- *Parsing*: Read a TIMESPEC specification and extract information to build the TA.
- *TA generation*: A particular TA is automatically constructed from the data provided by the previous step. The constructed TA is also visualized.
- *Code generation*: The last step automatically translates the TA into C and generates the formal contracts necessary for formal analysis.

Details related to the GROOT synthesis steps are now described.

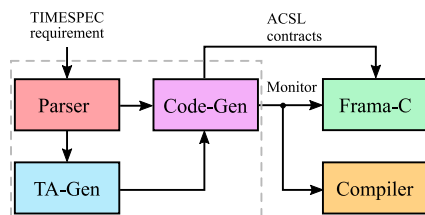


Figure 5: GROOT monitor synthesis in TIMESPEC flow.

### 3.3.1 TIMESPEC Parsing

The TIMESPEC parser extracts specific information from the input requirement and categorizes it based on the values of template fields. The parser accepts Booleans, integers, logical, arithmetic, and comparison operations for the SIGNAL field. The value of SIGNAL could be an expression and is used to create Boolean events, which are defined in Input\_Handler and used by the monitor.

### 3.3.2 TA Generation

A template-based flow reduces number of distinct TA variations needed, so TA-Gen creates the appropriate TA based on the data provided by the parser. Every monitor supports only one requirement and therefore the generated TA has one clock. Due to the wide application of RV in safety-critical systems, the TA must be deterministic to avoid any unpredictable behavior.

Similarities in timing constraints make the construction of the TA easier. For example, TA-Gen uses the structure illustrated in Figure 6 for three values of the TIMING\_CONSTRAINT field: between  $TL_{min}$  and  $TL_{max}$ , after  $TL_{min}$ , within  $TL_{max}$ . Transitions are guarded by Boolean events or restricted by clock values. As indicated earlier, state\_duration (sd) tracks time to detect deadline violations. Specifically, it records the number of transitions spent in the present state for comparison with the time bounds of the requirement every time the monitor is called. The sd variable purpose is twofold: detecting a timing violation and providing useful information such as how long TA spends in a state. TA-Gen also displays the generated TA to help the engineer understand how the automaton behaves.

### 3.3.3 Code Generation

Code-Gen translates the TA created by TA-Gen into a finite state machine (FSM) in C. It generates header

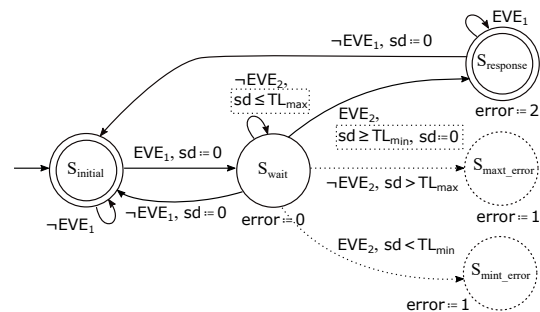


Figure 6: The TA structure used for between, after and within values in the TIMESPEC-flow. Solid lines are shared, while the dotted lines are chosen based on the requirement’s timing constraint.

and source files for the Monitor, Input\_Handler and Error\_Handler modules. The header file enumerates the TA states, defines time limit values, Monitor\_State, Monitor\_Inputs, and function declarations. Boolean events are stored in Monitor\_Inputs to simplify the monitor structure and reduce runtime overhead. The monitor output depends on the value of the error variable: (1) error = 0 means an unknown answer, indicating the monitor has not yet reached a verdict, (2) error = 1 indicates an error that needs action, and (3) error = 2 indicates the expected event is captured, and monitoring for the current trace is no longer needed.

To increase the portability of monitors on different platforms, a variable named monitor\_period is defined in the header file to specify the period of monitor invocation. As a result, the same monitor can be used without resynthesizing for systems with different time references by simply updating the monitor\_period value. The only human intervention required in this step is to assign a name to the monitor function and specify the argument types for Input\_Handler. This process ends with the creation of formal specifications, explained in the next section.

### 3.4 Monitor Formal Analysis

The correctness of the synthesized monitors is crucial as they are used to ensure system meets its specifications. Due to integer arithmetic calculations arising from time measurement, TIMESPEC analysis cannot easily use a model checker because of the state explosion problem (Valmari, 1998). Instead, monitors are analyzed using the Frama-C theorem prover (Cuoq et al., 2012) and formal behavioral contracts written in the ANSI/ISO C specification language (ACSL) (Baudin et al., 2008).

A significant advantage of using templates for real-time requirements is automating formal analysis by defining templated ACSL properties for each TIMESPEC statement. These formal contracts are generated by Code-Gen, verifying the validity of pointers, spatial distinctness and non-negativity of state\_duration and ensuring the input events are updated only by Input\_Handler. Some behaviors are also provided to verify the TA transition system, such as moving from INITIAL to WAIT state when the trigger event is captured. Another ACSL contract certifies that these behaviors are mutually exclusive.

## 4 EVALUATION

GROOT uses a Python implementation to read TIMESPEC statements and create the relevant monitor

modules and ACSL properties used in the Frama-C theorem prover. It also generates a task that is specifically written for embedded systems using a real-time operating system (RTOS). This task is responsible for wrapping the monitor blocks and can be executed by the RTOS scheduler with the lowest priority, and has the following structure:

```
void monitor_task(void *pvParameters) {
    struct Monitor_State monitor_state =
        { INITIAL, 0, 0 };
    struct Monitor_Inputs monitor_inputs =
        { 0 };
    while(1) { input_handler(&monitor_inputs);
              monitor(&monitor_state,
                    &monitor_inputs);
              error_handler(&monitor_state); }
}
```

The first example is hardware-oriented and illustrates monitor generation from TIMESPEC requirements for an industrial sensor. The second example uses a GROOT synthesized monitor to verify the correctness of an autonomous system modelled in Simulink (MathWorks, 2022b).

### 4.1 Slave Configuration Timings

This section selects several timing requirements from an industrial sensor, OPT9221, to illustrate how the TIMESPEC templates work. OPT9221 is a time-of-flight (ToF) controller that measures the depth of data from the digitized sensor data and is often used in robotics and automotive applications (Texas Instruments, 2015). Table 3 and Figure 7 show timing characteristics and timing diagram of the OPT9221 slave configuration signals, respectively. The  $t_{CZ}$  variable implies the pulse width of the active-low signal TIC\_CONFIGZ, which has a minimum duration. A  $t_{CLK}$  parameter refers to TIC\_CLK period with a specific duty cycle. The pulse width of these signals should be within a particular range for proper operation and therefore a duration template is used.

Selecting the appropriate timing constraint is critical for the causality template. For example, asserting TIC\_CONF\_DONE causes TIC\_INIT\_DONE (denoted by  $t_{INIT}$ ) to be asserted. This behavior should happen within a certain time bound, captured by the between value for TIMING\_CONSTRAINT. Another instance is  $t_{DC}$ , where TIC\_STATUSZ deassertion starts the clock

Table 3: Timing characteristics of OPT9221 slave configuration (Texas Instruments, 2015).

|            | PARAMETER   | MIN | MAX | UNIT |
|------------|---|-----|-----|------|
| $t_{CZ}$   | TIC_CONFIG low pulse duration                                     | 500 |     | ns   |
| $t_{CR}$   | Delay from TIC_CONFIG falling edge to TIC_CONF_DONE falling edge  |     | 500 | ns   |
| $t_{SZ}$   | TIC_STATUS low pulse duration                                     | 45  | 230 | us   |
| $t_{DC}$   | TIC_STATUS rising edge to configuration clock's first rising edge |     | 2   | us   |
| $t_{CLK}$  | Configuration clock period  | 15  |     | ns   |
| DC         | Configuration clock duty cycle                                    | 40% | 60% |      |
| $t_{INIT}$ | End of configuration to start of firmware execution               | 300 | 650 | us   |

Table 4: Timing requirements of OPT922 using pulse duration and causality templates.

|                |     |            |   |
|----------------|-----|------------|---|
| Pulse Duration | RP1 | $t_{CZ}$   | Active_pulse_width of TIC_CONFIGZ should be $\geq 500$ ns.                  |
|                | RP2 | $t_{SZ}$   | Active_pulse_width of TIC_STATUSZ should be $\geq 45$ us and $\leq 230$ ns. |
|                | RP3 | $t_{CLK}$  | Period of TIC_CLK should be $\geq 15$ ns.                                   |
|                | RP4 | DC         | Duty_cycle of TIC_CLK should be $\geq 40\%$ and $\leq 60\%$ .               |
| Causality      | RC1 | $t_{CR}$   | If assert TIC_CONFIGZ, deassert TIC_CONF_DONE within 500 ns.                |
|                | RC2 | $t_{DC}$   | If deassert TIC_STATUSZ, start TIC_CLK within 2 us.                         |
|                | RC3 | $t_{INIT}$ | If assert TIC_CONF_DONE, assert TIC_INIT_DONE between 300 us and 650 us.    |

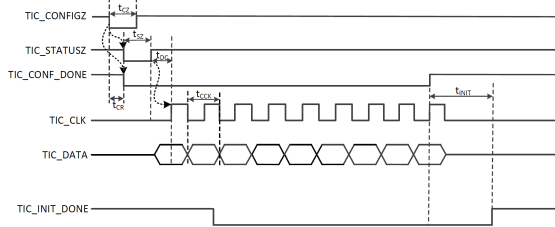


Figure 7: Timing diagram of OPT9221 slave configuration (Texas Instruments, 2015).

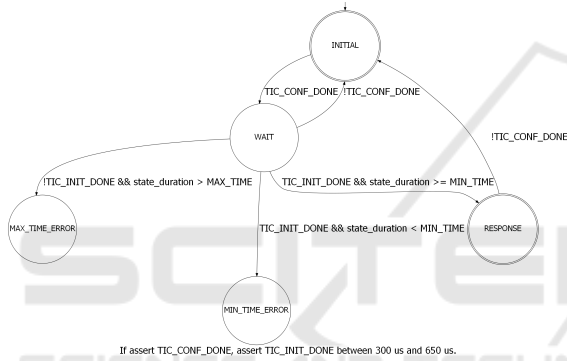


Figure 8: TA constructed by TA-Gen for requirement RC3.

signal TIC\_CLK within a specific time. These timing specifications are listed in Table 4.

GROOT automatically displays a simplified version of the generated TA as shown in Figure 8 for RC3. Assuming the monitor works with a 1 MHz clock ( $monitor\_period = 1$ ), Code\_Gen sets MIN.TIME and MAX.TIME values to 300 and 650, respectively. The Input.Handler block captures the relevant events such as the rising edge of TIC.CONFIG\_DONE and TIC.INIT\_DONE signals. As long as the monitor stays in the WAIT state, state.duration increments by one every time the monitor is invoked. This counter is compared to the MIN.TIME and MAX.TIME constants to ensure the expected event does not arrive sooner than the minimum time (300 us) but not later than the maximum time (650 us) either. If these clock guards fail, the monitor sets the error variable, notifies Error\_Handler, and transitions to x.TIME.ERROR. The monitor remains in this state until the external Error\_Handler resets it.

Figure 9 shows the ACSL properties that Code-Gen creates for RC3. Section 3.4 describes the mean-

```

/*@
requires \valid(monitor_state) && \valid_read(monitor_inputs);
requires \separated(monitor_state, monitor_inputs);

requires 0 <= monitor_state->state_duration;

ensures unchanged_inputs: *monitor_inputs == \old(*monitor_inputs);

behavior not_started:
  assumes monitor_state->state \in {INITIAL};
  assumes monitor_inputs->TIC_CONF_DONE == 0;

  ensures monitor_state->state == INITIAL;

behavior begin:
  assumes monitor_state->state \in {INITIAL};
  assumes monitor_inputs->TIC_CONF_DONE == 1;

  ensures monitor_state->state == WAIT;

behavior satisfaction_or_min_time_violation:
  assumes monitor_state->state \in {WAIT};
  assumes monitor_inputs->TIC_INIT_DONE == 1;

  ensures ((monitor_state->state_duration >= MIN_TIME ==>
    monitor_state->state == RESPONSE) ||
    (monitor_state->state_duration < MIN_TIME ==>
    monitor_state->state == MIN_TIME_ERROR));

behavior wait_or_max_time_violation:
  assumes monitor_state->state \in {WAIT};
  assumes monitor_inputs->TIC_INIT_DONE == 0;

  ensures ((monitor_state->state_duration <= MAX_TIME ==>
    monitor_state->state == WAIT) ||
    (monitor_state->state_duration > MAX_TIME ==>
    monitor_state->state == MAX_TIME_ERROR));

behavior stop:
  assumes monitor_state->state \in {MIN_TIME_ERROR, MAX_TIME_ERROR};

  ensures monitor_state->error == 1;

disjoint behaviors;
*/

```

Figure 9: ACSL contracts for proving the correctness of  $t_{INIT}$  monitor by Frama-C.

ing of each specification. For instance, the first contract requires monitor\_state and monitor\_inputs to be valid pointers, where monitor\_inputs is only readable to the monitor. The not-started behavior verifies that as long as TIC.CONFIG\_DONE is deasserted, the monitor stays in the INITIAL state.

## 4.2 Autonomous Emergency Braking

An autonomous emergency braking (AEB) system is a safety feature in modern vehicles that detects potential hazards using sensors and calculates the time to collision (TTC). If the TTC falls below a certain threshold, the system sends a forward collision warning (FCW) to the driver. If the driver fails to act, the AEB system will automatically apply the brakes to mitigate the severity of the collision.

We use an existing example (MathWorks, 2022a) to monitor the behavior of an AEB system in the Simulink environment. The system activates the FCW flag and notifies the driver in case of an emergency. The AEB system automatically initiates deceleration only when the TTC value shows the driver cannot take action in time. Assuming a driver’s reaction time of 1.2 s, GROOT generates a monitor for a TIMESPEC requirement declaring “AEB\_req1: If assert FCW\_activate, assert (ego\_acceleration < -0.5) within 1.2 s.” This monitor ensures that either the driver or the AEB system apply the brakes to avoid an imminent collision. Input\_Handler creates two events: “EVE1:FCW\_activate” and “EVE2:(ego\_acceleration < -0.5)”. The simulation’s time step is 50 ms, therefore the monitor has a monitor\_period of 50 and a time\_value of 1200. If the monitor invocation frequency or timing constraint change, these two definitions can be updated in the monitor header file without resynthesizing the monitor.

Figure 10 illustrates the AEB system simulation outcomes, where the autonomous (ego) car moves at a constant velocity of 8.33 m/s, and a stationary vehicle exists in front of it. At  $t=1$ s, the FCW is triggered and the monitor moves to the WAIT state, resetting state\_duration. The monitor detects a violation if state\_duration exceeds 24 (1200/50). However, at  $t=2$ s, the AEB system applies the brakes and the monitor switches to the RESPONSE state, remaining there until the FCW deactivates. The error variable also changes from 0 to 2, showing the expected behavior is captured. The duration between EVE1 and EVE2 is less than one second, indicating no violation has occurred, which is confirmed by the monitor. To verify the correctness of the synthesized monitor, Frama-C is executed with ACSL contracts similar to Figure 9. The result of this verification is shown in Figure 11.

## 5 RELATED WORK

Runtime monitoring of real-time systems has been the focus of numerous research studies in recent years (Gorostiaga and Sánchez, 2018; Torfah, 2019; Aurandt et al., 2022). This paper focuses on studies relevant to our work, i.e., utilizes natural language to mask formal notations and ease their use.

Various works have attempted to translate informal real-time properties to formal forms (Konrad and Cheng, 2005; Autili et al., 2015). One such tool is DeepSTL (He et al., 2022), which utilizes natural language processing (NLP) to convert unstructured English requirements of cyber-physical systems (CPS) into signal temporal logic (STL) (Maler and Nickovic,

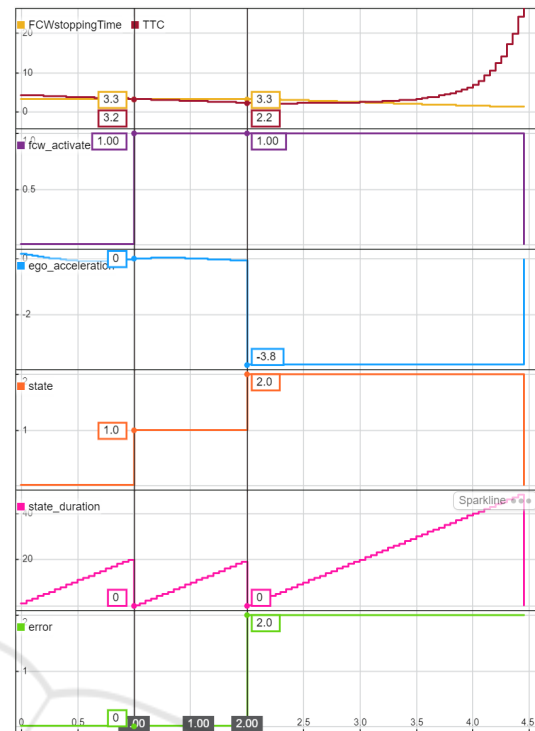


Figure 10: Simulation results of the AEB\_req1 requirement.

```

tor/AutonomousEmergencyBraking$ frama-c -wp -wp-rte AEB.c AEB.h main_AEB.c
[kernel] Parsing AEB.c (with preprocessing)
[kernel] Parsing AEB.h (with preprocessing)
[kernel] Parsing main_AEB.c (with preprocessing)
[rte] annotating function AEB_monitor
[rte] annotating function main
[wp] [CFG] Goal main_assert : Valid (Unreachable)
[wp] main_AEB.c:8: Warning:
  Missing assigns clause (assigns 'everything' instead)
[wp] 56 goals scheduled
[wp] [Cache] Found:24
[wp] Proved goals: 56 / 56
[wp] Qed: 32 (2ms-36ms-280ms)
Alt-Ergo 2.3.3: 24 (9ms-23ms-29ms) (141) (cached: 24)

```

Figure 11: Verification results of AEB\_req1 using Frama-C.

2004). Although these methods could be helpful in industry, none have been used for monitor generation.

Rajhans et al. introduce a user-friendly approach for creating formal temporal formulas in Simulink Test™ using a graphical user interface (GUI) and English patterns (Rajhans et al., 2021). The GUI offers bound checks, trigger response patterns and visualizes the specification data, making it simpler for practitioners to assess the runtime of Simulink models. However, this approach lacks a stand-alone monitor that can be utilized across different platforms or after the system is deployed. The RuSTL tool is another approach that translates structured English statements to STL formulas and generates offline monitors for log files (Khan, 2019). An offline monitor is not executed during runtime and is useful for testing and debugging. The monitor is also written in Python, which is not applicable to embedded systems.

Perez et al. present a toolchain that generates C

monitors from structured English statements using Copilot and FRET (Perez et al., 2022). Copilot is a stream-based RV that verifies system behavior by analyzing the dependencies between input and output streams, making it suitable for systems that process large amounts of data. While this work is similar to GROOT in terms of using structured English requirements and targeting real-time embedded systems, GROOT is an automata-based approach that is better suited for systems with a finite set of states and for detecting errors in the system’s control flow. Automata-based RV is simpler to implement and easier to understand and visualize. The suitability of a particular approach depends on the specific requirements and constraints of the system being verified.

## 6 CONCLUSIONS

Runtime verification (RV) can improve trust in real-time systems with critical timing constraints by generating monitors from formal system specifications. However, a lack of standard specification languages makes RV adoption challenging for practicing engineers without formal method backgrounds. To address this issue, we propose GROOT, a monitor generation tool that uses structured English statements to automatically synthesize monitors in C using two distinct flows for functional and timing requirements. This paper focuses on timing constraints through the introduction of the TIMESPEC dialect, and highlights GROOT’s ability to automate formal analysis of monitor correctness using a theorem prover. By providing an accessible approach to RV, GROOT can help the adoption of formal methods in industry, leading to safer and more reliable real-time systems.

In future work, we plan to add more TIMESPEC templates to support complex requirements containing several timing constraints. GROOT monitors will be tested by generating timing and functional specifications for drones. We intend to compare the performance of the GROOT’s monitors with other frameworks; however, documentation regarding other RV tools is scarce.

## ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 2123550. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

## REFERENCES

- Alur, R. and Dill, D. L. (1994). A theory of timed automata. *Theoretical Comput. Sci.*, 126(2):183–235.
- Aurandt, A. et al. (2022). Runtime verification triggers real-time, autonomous fault recovery on the CySat-I. In *NASA Formal Methods*, pages 816–825. Springer.
- Autili, M. et al. (2015). Aligning qualitative, real-time, and probabilistic property specification patterns using a structured English grammar. *IEEE Trans. Softw. Eng.*, 41(7):620–638.
- Bartocci, E. et al. (2018). *Introduction to Runtime Verification*, pages 1–33. Springer, Cham.
- Baudin, P. et al. (2008). ACSL: ANSI/ISO C specification language.
- Cuoq, P. et al. (2012). Frama-C: A software analysis perspective. In *Proc. Int. Conf. Softw. Eng. and Formal Methods*, SEFM’12, page 233–247, Berlin. Springer.
- Falcone, Y. et al. (2021). A taxonomy for classifying runtime verification tools. *Int. J. Softw. Tools for Technol. Transfer*, 23(2):255–284.
- Giannakopoulou, D. et al. (2020). Formal requirements elicitation with FRET. In *REFSQ Workshops*.
- Gorostiaga, F. and Sánchez, C. (2018). Striver: Stream runtime verification for real-time event-streams. In *Runtime Verification*, pages 282–298, Cham. Springer.
- He, J. et al. (2022). DeepSTL: From English requirements to Signal Temporal Logic. In *Proc. Int. Conf. Softw. Eng.*, ICSE’22, page 610–622, New York, USA.
- Khan, W. (2019). RuSTL: Runtime verification using STL. Master’s thesis, University of Waterloo.
- Konrad, S. and Cheng, B. H. C. (2005). Real-time specification patterns. In *Proc. Int. Conf. Softw. Eng.*, ICSE’05, page 372–381, New York, USA.
- Maler, O. and Nickovic, D. (2004). Monitoring temporal properties of continuous signals. In *Formal Techn., Modelling and Anal. of Timed and Fault-Tolerant Syst.*, pages 152–166, Berlin. Springer.
- MathWorks (2022a). *Autonomous Emergency Braking with Sensor Fusion*. Natick, MA, USA.
- MathWorks (2022b). *Simulink: A graphical programming environment for modeling, simulating, and analyzing dynamic systems*. Natick, MA, USA.
- Perez, I. et al. (2022). Automated translation of natural language requirements to runtime monitors. In *Tools and Algorithms for the Construction and Anal. of Syst.*, pages 387–395, Cham. Springer.
- Pnueli, A. (1977). The temporal logic of programs. In *Annu. Symp. Found. Comput. Sci. (SFCS’77)*, pages 46–57.
- Rajhans, A. et al. (2021). Specification and runtime verification of temporal assessments in Simulink. In *Runtime Verification*, pages 288–296, Cham. Springer.
- Texas Instruments (2015). *OPT9221 Time-of-Flight Controller*. Dallas, TX, USA.
- Torfah, H. (2019). Stream-based monitors for real-time properties. In *RV*, pages 91–110, Cham. Springer.
- Valmari, A. (1998). *The state explosion problem*, pages 429–528. Springer, Berlin.