

A Monitoring Methodology and Framework to Partition Embedded Systems Requirements

Behnaz Rezvani^a and Cameron Patterson^b

Bradley Dept. of Electrical and Computer Engineering,
Virginia Tech, Blacksburg VA 24060, U.S.A.

Keywords: Runtime Verification, Monitor, Functional Requirements, Timing Requirements.

Abstract: The adoption of runtime monitoring has historically been limited to experts, primarily due to the intricate complexities associated with formal notations and the verification process. In response to this limitation, this paper introduces GROOT, a methodology and framework specifically designed for the automated synthesis of runtime verification monitors from structured English requirements. GROOT is tailored to address the challenges of adhering to both functional and timing constraints within complex real-time embedded systems. It accomplishes this through a dual approach that handles functional and timing requirements separately, allowing customized verification processes for each category. To demonstrate GROOT's practical utility, its monitors are applied to an autonomous system modeled in Simulink.

1 INTRODUCTION


Real-time embedded systems are used in a wide range of safety-critical applications such as avionics, robotics, and autonomous systems, where system performance depends on timely responses. These systems must meet both functional and timing requirements. Common verification techniques such as testing can efficiently detect straightforward errors. However, exhaustive and time-consuming test cases are required to achieve maximum coverage, and yet it is possible to miss a subtle error in a large and complex system. Runtime verification (RV) is a dynamic verification approach that utilizes monitors derived from formal system requirements to determine whether the real-time behaviors of a system adhere to its specifications (Leucker and Schallhart, 2009).


Formal approaches such as metric temporal logic (MTL) (Koymans, 1990) and TeSSLa (Leucker et al., 2018) have been developed to address both functional and timing specifications. However, the lack of a standardized specification language complicates the practical application of these methods (Dwyer et al., 1999). Practitioners are often required to have a deep understanding of formal methods and domain-specific tool notations. In contrast to functional requirements,

timing requirements often have fewer variations, enabling the reuse of monitors and minimizing resource usage. This could be achieved by applying suitable formalisms for each type of requirement.

To facilitate RV adoption without formal methods training, this paper introduces GROOT (Generalized Runtime mOnitOring Tool), a novel methodology and framework to automate the synthesis of monitors from structured English specifications. GROOT achieves its goals by offering a dual approach for functional and timing requirements, enabling customized verification for precise validation of runtime system behavior. This fully automated process involves translating English properties to formalisms, converting them to monitor automata, and formally verifying the monitors.

Embedded systems may combine software programming with hardware elements such as FPGAs to perform dedicated functions. These systems have a wide range of requirements from high-level system functionalities to detailed low-level aspects such as signal generation and clock timing complexities. For the functional requirements, the NASA FRET tool (Giannakopoulou et al., 2020) is used to formalize structured English expressions into linear temporal logic formulas (LTL) (Pnueli, 1977). For timing requirements, we introduce TIMESPEC, a structured English language specifically designed to capture metric time constraints. GROOT transforms

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

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

these natural language statements into deterministic automata through an automated synthesis process. The generated monitors are executed outside the software/hardware system, treating the application as a black box. This makes GROOT a versatile and adaptable framework for RV. Monitor inputs and violation handling are managed separately in external modules to keep monitor structure simple, easing formal analysis. This framework establishes monitor correctness through rigorous static formal verification. The monitors can aid debugging during development or increase application trust after system deployment.

In this paper, the practical aspects of GROOT are shown through its application to an autonomous system modeled in Simulink® (The MathWorks Inc., 2023). The main contributions are:

- *Automated three-step methodology*: GROOT employs a sequential and automated process involving the translation of English properties into formalisms, transformation into monitor automata, and formal verification of these monitors. This helps to make RV more accessible and understandable to practitioners by hiding formal notations and automating the entire process of monitor generation and verification.
- *Dual approach*: GROOT uses a novel approach that separately addresses functional and timing requirements. This separation facilitates the customization of monitor synthesis and formal analysis according to the specific needs of each requirement type, thereby enhancing the verification's effectiveness.
- *Application versatility*: As standalone entities, GROOT monitors can either use system resources or execute on independent resources, offering flexibility in implementation. The generated monitors can be applied during both the development phase and post-deployment. This makes GROOT a valuable tool in various stages of the system development life cycle.

2 METHODOLOGY

GROOT aims to make RV more accessible and practical for a broader range of users, especially those who are not experts in the field. It achieves this by expressing system requirements in structured English to mask the complexities of formal notations. As shown in Figure 1, the monitor synthesis flow consists of three steps: formalization, monitor generation, and formal analysis. To aid practitioners, the entire process is mostly automated. Verification engineers are only re-

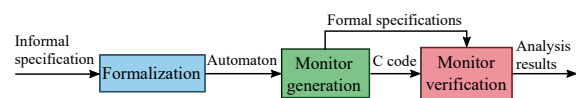


Figure 1: GROOT flow visualization.

quired to provide the initial properties and offer occasional guidance throughout the process to ensure the generated monitors align with their expectations.

2.1 Formalization

The formalization step forms the foundation for subsequent stages, and its goal is reducing ambiguity by automating the translation of structured English properties into formalisms, specifically automata. It also provides visual models of automata for an intuitive comprehension of monitor dynamics. In this phase, various structured English patterns are tailored for functional and timing requirements. Specification patterns, inherently amenable to automation, make GROOT an efficient method by significantly reducing the manual effort, ensuring consistent representation of similar concepts across specifications and enhancing the reliability of monitor synthesis.

2.2 Automated Monitor Generation

The generated formal models are automatically transformed into finite state machines (FSMs) during this step. FSMs are a familiar abstraction to most computer engineers and fulfill GROOT's objective of creating clear and understandable monitors. They are particularly effective in tracking event sequences, making them well-suited for monitoring system transitions. The widespread use of FSMs, compared to other formal languages (Havelund, 2008), reduces GROOT's learning curve.

2.3 Monitor Verification

Monitors are generated using several transformations on the specified requirements. To ensure correctness of these synthesized monitors, GROOT employs static formal verification techniques including model checking (Baier and Katoen, 2008) and theorem proving (Harrison et al., 2014). This process checks for logical errors and design flaws in the final translation using rigorous mathematical methods without modifying the source code. The use of specification patterns results in standardization of monitor code structure and also consistency and predictability in monitor behavior. Leveraging these patterns, GROOT automates the generation of formal specifications essential for the execution of model checking and theo-

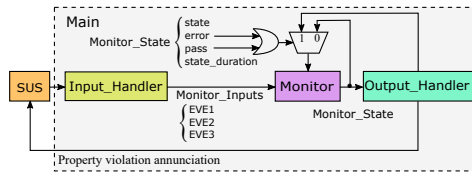


Figure 2: Structure of GROOT monitoring process.

rem proving. Embedding formal analysis ensures that monitors strictly adhere to state transitions defined in the abstract automata.

3 MONITOR STRUCTURE

To ensure adequate performance in systems with constrained resources, it is crucial to add the monitors with care. GROOT allows flexible deployment of monitors on the same or separate hardware to minimize overhead and prevent interference with system timings. An example would be implementing monitors directly in digital hardware rather than on an instruction set processor. Monitors handle inputs and responses through external entities, which can be shared and simplified to reduce resource usage and better suit formal analysis. Figure 2 illustrates the monitor structure containing the auto-generated Main block and its fundamental components.

The Input_Handler module acts as an intermediary between the system under scrutiny (SUS) and the monitoring framework, translating system data into Boolean atomic propositions (APs) representing specific events. The Monitor_Inputs structure captures these events and is exclusively updated by Input_Handler. The Monitor component, synced with the system’s time base, is the core of the monitoring process. The Monitor_State structure provides a snapshot of the current state, with error and pass variables indicating violation or successful event sequencing, respectively. The state.duration counter keeps track of self-transitions from the current state. Upon reaching a verdict, Monitor triggers Output_Handler to inform the SUS and reset Monitor_State. The architecture can adapt to various requirements, with Monitor being the variable component for different requirements.

4 FRAMEWORK

In our experience, employing a single logic or type of automaton for both functional and timing specifications introduces complications that may adversely impact formal analysis outcomes. In GROOT, these two classes of requirements are handled in different ways,

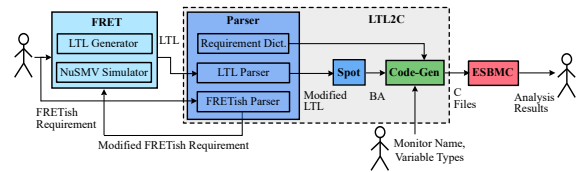


Figure 3: GROOT functional requirements flow.

which is consistent with the conventional practice in hardware engineering where the documentation and design of functionality and timing aspects are treated separately. These two flows are now described.

4.1 Monitor Synthesis for Functional Requirements

Figure 3 depicts the monitor synthesis process within the functional requirements branch of GROOT. Similar to a compiler, there is a front-end and back-end architecture. The front-end utilizes NASA’s FRET tool to capture system specifications phrased in a structured English language and transform them into corresponding LTL formulas (formalization step). Subsequently, the back-end parses and translates these LTL formulas into Büchi automata (BA) (Büchi, 1990) using the Spot tool (Duret-Lutz et al., 2016). The generated automata are then converted to FSMs expressed in C (monitor generation step). Finally, the ESBMC model checker (Gadelha et al., 2018) is applied to confirm that state transitions within the monitors comply with a set of generated assertions (monitor verification step).

4.1.1 Synthesis Front-End

One way to improve the accessibility of formal methods for practitioners is to define a set of property patterns tailored to common types of requirements. NASA’s FRET tool offers the structured English language FRETish to conceal the LTL patterns. The FRETish requirement structure consists of six sequential fields: SCOPE, CONDITIONS, COMPONENT, SHALL, TIMING, and RESPONSES. While COMPONENT, SHALL and RESPONSES are mandatory, the other fields are optional. FRET also provides requirement visualization using the integrated NuSMV simulator (Cimatti et al., 2002).

4.1.2 Synthesis Back-End

Büchi automata (BA) are commonly used to generate monitor automata from LTL requirements (Gianakopoulou and Havelund, 2001). To facilitate this conversion process, the LTL2C module has been developed and consists of three core elements: (i) the

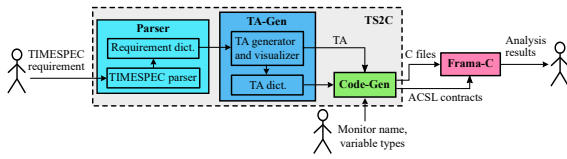


Figure 4: GROOT timing requirements flow.

Spot API, (ii) LTL2C parser, and (iii) Code-Gen. Afterwards, LTL2C invokes ESBMC to ensure the monitor implementations adhere to the generated BA.

Spot API. Spot is a robust open-source tool able to translate LTL formulas into BA, with its maturity, options and efficiency suiting industrial applications. Its Python interface allows LTL2C to generate and visualize the associated BA.

LTL2C Parser. FRET allows requirements to be specified with basic arithmetic/comparison operations and numeric variables. However, Spot and NuSMV exclusively handle Boolean APs. To reconcile this, LTL2C employs a specialized parser, which replaces arithmetic/comparison operations with Boolean variables. This parser generates LTL formulas compatible with Spot and FRETish requirements compatible with NuSMV, as depicted in Figure 3.

Code Generator. The Code-Gen module translates the generated automaton into a concrete FSM implemented in C. It also handles undefined behavior, which directs the FSM to an error state. Human intervention is only required to assign the monitor’s name and specify the types of monitor’s arguments. This phase automatically generates the source code and header files for Main, Input_Handler, Output_Handler, and Monitor. To ease the model checking step, Code-Gen provides assertions extracted from the produced BA to scrutinize the final monitor for any deviation from the expected behavior.

Formal Analysis. ESBMC is a model checker designed for validating C and C++ programs against established properties such as pointer validity and deadlock prevention. It also allows for user-defined assertions. To complete the synthesis process, ESBMC verifies the generated assertions, ensuring consistent adherence to correct state and transitions throughout the monitoring process.

4.2 Monitor Synthesis for Timing Requirements

The GROOT timing flow defines TIMESPEC, a structured English language for articulating timing spec-

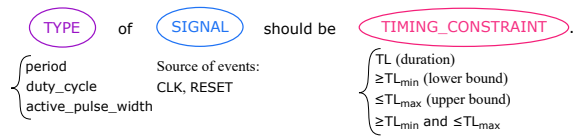


Figure 5: TIMESPEC pulse duration template.

ifications. This language enables engineers to easily capture timing constraints without needing any knowledge of formal methods. Similar to functional requirements, monitor generation and verification are automated, reducing manual effort and error risk.

Figure 4 depicts the monitor synthesis process for timing properties. It starts with parsing TIMESPEC statements to extract essential information. This data is used to generate timed automata (TA) (Alur and Dill, 1994), which represent the formalized version of timing requirements (formalization phase). The TA is transformed into FSMs implemented in C (monitor generation step). Lastly, the Frama-C theorem prover (Cuoq et al., 2012) checks that the monitor implementations adhere to predefined set of behavioral properties which establish correct state sequencing (monitor verification step). This paper provides a brief overview of the GROOT flow for timing requirements, with a more comprehensive explanation available in (Rezvani and Patterson, 2023).

4.2.1 TIMESPEC

Ensuring metric time constraints is vital in safety-critical systems, where any violation could lead to system failure. Timing specifications are captured with TIMESPEC requirements. TIMESPEC aids practitioners by providing a collection of natural language templates for common timing constraints.

Pulse Duration Template. In digital systems, correct functionality relies heavily on signals and clock timings. Timing requirements often involve pulse width for signals, which must meet specific criteria to trigger intended events. To manage these requirements, a template is used to specify pulse duration of signals or period and duty cycle of clock signals. Figure 5 shows the pulse duration template and its supported values. The TYPE field defines the requirement type, while SIGNAL indicates either a clock signal or an event. TIMING_CONSTRAINT specifies the time limit, with each time limit (TL/TL_x) consisting of a numerical value and time unit. While most TYPES are in absolute units like nanoseconds (ns), duty cycle limits are typically expressed as percentages.

Causality Template. In protocols such as handshaking, events are often causally linked, where one



Figure 6: TIMESPEC causality template.

event triggers another within a specific time frame. This behavior is addressed with the causality template, depicted in Figure 6. The ACTION field specifies the type of event (like asserting or deasserting signals, or starting a clock), and SIGNAL identifies the events involved. The TIMING_CONSTRAINT field defines the time-dependent relationships between these events.

4.2.2 Synthesis Workflow

FRET translates certain timing properties into MTL formulas, but its effectiveness is limited by partial support for timing constraints and the lack of a framework for transforming MTL to automata. Acknowledging the importance of TA in capturing system temporal behaviors, GROOT automates the conversion of TIMESPEC statements into TA with TS2C. This tool generates C-based monitors via three modules: (i) TS2C parser, (ii) TA-Gen, and (iii) Code-Gen, also providing necessary specifications for formal analysis of the generated monitors.

TS2C Parser. This module is designed to process TIMESPEC expressions and extract essential details from temporal requirements. It creates a dictionary containing Boolean events (APs), input variables, types of timing constraints, and corresponding time values and units. The parser can manage various SIGNAL expressions including Booleans, integers, and logical/arithmetic/comparison operations, allowing a wide range of temporal behaviors to be captured.

TA Generator. After parsing TIMESPEC requirements, TA-Gen maps each statement to a particular TA. This template-based approach reduces the variety of TA variations required, thereby providing both efficiency and structural consistency. Each automaton has a single clock to enhance clarity and ease of analysis, with transitions determined by Boolean events or temporal limits. The `state.duration (sd)` parameter records the time spent in a specific state, facilitating detection of deadline violations. TA-Gen also visualizes the generated TA to improve comprehension.

Code Generator. Similar to LTL2C, the Code-Gen unit converts the TA into a C-based FSM. It defines `monitor_period` in the header file to set the monitor invocation frequency, enabling their reuse in environ-

ments with different timebases. Code-Gen also generates specifications required for theorem proving.

Formal Analysis. Formal verification is essential to establish monitor trustworthiness, however, traditional model checkers struggle with the complexity of integer arithmetic in time measurements due to the state explosion problem (Clarke et al., 2012). GROOT uses the Frama-C theorem prover along with formal contracts generated by Code-Gen, which are formulated in the ANSI/ISO C specification language (ACSL) (Baudin et al., 2008). Each TIMESPEC requirement template includes a set of predefined and parameterized ACSL specifications. These specifications cover various aspects of monitor behavior such as the validity of pointers or transition verification within the monitor (Rezvani and Patterson, 2023).

5 FRAMEWORK DEMONSTRATION

Adaptive cruise control (ACC) systems enhance traditional cruise control (CC) systems by autonomously regulating a vehicle's speed to match the driver's preference. Unlike conventional CC systems, ACC automatically decelerates to maintain a safe following distance (D_{safe}) when it detects a lead vehicle in the same lane ahead of the autonomous (ego) car. A critical parameter in ACC is the headway time ($\tau_{headway}$), which represents the constant time gap maintained between the ego and lead vehicles. This parameter confirms that the ego vehicle has sufficient time to react and decelerate if the lead vehicle suddenly stops. To ensure safety, an offset distance (D_{offset}) is considered, and D_{safe} can be written as:

$$D_{safe} = V_{ego} \times \tau_{headway} + D_{offset} \quad (1)$$

Table 1 shows two requirements of the ACC system. The first requirement, R1, mandates that the relative distance ($D_{relative}$) between the ego and lead vehicles must not fall below D_{safe} throughout the entire operation. R1 is articulated in FRETish and the associated LTL formula is processed by LTL2C. The BA generated by Spot is depicted in Figure 7. It demonstrates that upon violation of the condition, the BA moves to state 1 and remains there. Subsequently, Code-Gen transforms the BA into C code and generates assertions, as illustrated in Figure 8. These assertions validate that monitor state transitions comply with those specified in the generated BA. ESBMC is then applied to ensure these assertions hold true.

The second requirement, R2, guarantees that the ego vehicle's velocity (V_{ego}) adjusts to the set tar-

Table 1: ACC system requirements.

R1: Ego vehicle should always have a safe space from lead vehicle.	
FRETish	ACC shall always satisfy ($D_{relative} \geq D_{safe}$).
LTL	(LAST V ($D_{relative} \geq D_{safe}$))
R2: If relative distance is safe, ACC should reach set velocity within 3 seconds.	
TIMESPEC	If assert ($D_{relative} \geq D_{safe}$), assert ($V_{ego} \geq V_{set.n}$ && $V_{ego} \leq V_{set.p}$) within 3 s.

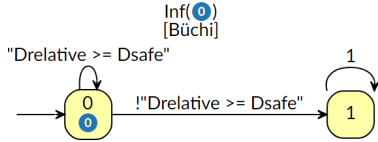


Figure 7: BA built by the Spot tool for R1.

```

assert(state == INITIAL || state == ERROR);
assert((state == ERROR) == error);
assert(state_duration >= 0);

if (state == INITIAL) {
    assert(!EVE0 == (state == ERROR && error));
    assert(EVE0 == (state == INITIAL && !error));
}
    
```

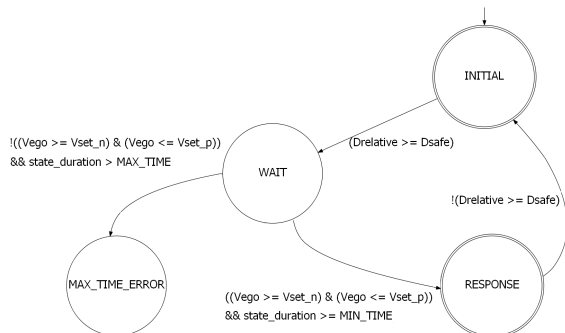
Figure 8: Assertions provided by Code-Gen for R1.

get velocity (V_{set}) within a specific time frame, but only when the relative distance is considered safe. To ensure compliance with this requirement, a velocity threshold (V_{th}) is established to verify that V_{ego} remains within an acceptable range, shown as follows.

$$V_{set.n} \leq V_{ego} \leq V_{set.p}, \quad (2)$$

where $V_{set.n} = V_{set} - V_{th}$ and $V_{set.p} = V_{set} + V_{th}$

This requirement is formalized using a TIME-SPEC causality template, as detailed in Table 1, and subsequently processed by TS2C. For a clearer understanding of the monitor's behavior, Figure 9 shows the TA generated by TA-Gen. Following the transformation of the TA into C code, Code-Gen finalizes this process by generating the ACSL specifications pertaining to R2, as illustrated in Figure 10. The Framac theorem prover is invoked to validate that the monitor sequences through the states correctly.



If assert ($D_{relative} \geq D_{safe}$), assert ($V_{ego} \geq V_{set.n}$ & ($V_{ego} \leq V_{set.p}$) within 3 s.

Figure 9: TA built by TA-Gen for R2.

```

/*@
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->EVE0 == 0;
    ensures monitor_state->state == INITIAL;

behavior begin:
    assumes monitor_state->state \in {INITIAL};
    assumes monitor_inputs->EVE0 == 0;
    ensures monitor_state->state == WAIT;

behavior satisfaction:
    assumes monitor_state->state \in {WAIT};
    assumes monitor_inputs->EVE1 == 1;
    ensures monitor_state->state == RESPONSE;

behavior wait_or_max_time_violation:
    assumes monitor_state->state \in {WAIT};
    assumes monitor_inputs->EVE1 == 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 reset_trigger:
    assumes monitor_state->state \in {RESPONSE};
    assumes monitor_inputs->EVE0 == 1;
    ensures monitor_state->state == RESPONSE;

behavior reinitialize:
    assumes monitor_state->state \in {RESPONSE};
    assumes monitor_inputs->EVE0 == 0;
    ensures monitor_state->state == INITIAL;

behavior stop:
    assumes monitor_state->state \in {MAX_TIME_ERROR};
    ensures monitor_state->error == 1;

disjoint behaviors;
*/
    
```

Figure 10: Ascl contracts provided by Code-Gen for R2.

In this paper, a Simulink model of an ACC system (The MathWorks Inc., 2024) is used to demonstrate the GROOT workflow which illustrates both functional and timing requirements. We integrate the two generated monitors into the ACC model to check that a safe following distance is maintained and ensure the ACC system accelerates in a correct manner. As shown in (3) and (4), the two events EVE0 and EVE1 are defined and managed by the Input.Handler module. Table 2 shows the default parameter values used to determine these two events.

$$EVE0 : D_{relative} \geq D_{safe} \quad (3)$$

$$EVE1 : V_{ego} \geq V_{set.n} \ \&\& \ V_{ego} \leq V_{set.p} \quad (4)$$

Simulation results for the R1 monitor are depicted in Figure 11. Initially, the ego vehicle maintains a safe following distance ($EVE0 = 1$). Around $t = 12.5$ s, an-

Table 2: ACC model parameters default values.

Parameter	Description	Value
τ_{headway}	Headway time	1.5 s
D_{offset}	Offset distance	15 m
V_{set}	Set velocity	21.5 m/s
V_{th}	Velocity threshold	1 m/s

other vehicle cuts into its lane, causing D_{relative} to decrease. At $t = 16.6$ s, D_{relative} falls below D_{safe} (EVE0 = 0), triggering an error condition. This critical event is immediately detected by the R1 monitor, which sets the error flag until D_{relative} returns to the safe zone. In this particular scenario, the monitor's error signal could be used to activate an alert system, ensuring the driver is promptly informed in case the ACC system fails to respond as expected.

Figure 12 presents the simulation results for the R2 monitor. Given the simulation time step of 100 ms, monitor_period is set to 0.1, and MAX.TIME is automatically calculated as 30, corresponding to time limit of 3 seconds. At $t = 19.1$ s, the lead vehicle from the preceding scenario changes lanes, leading to a noticeable increase in D_{relative} (EVE0 = 1). This change prompts the R2 monitor to check whether V_{ego} reaches the specified range within the desired time frame. By $t = 22.1$ s, the velocity aligns with the target range (EVE1 = 1), indicating a successful response before the deadline expires. At this point, the monitor transitions to the RESPONSE state, and the pass flag is set to true. The monitor remains in this state as long as the relative distance is within the safe margin. This monitoring approach is particularly useful during the debugging phase, which provides a means to confirm the ACC system's ability to safely and promptly resume acceleration after an obstruction clears.

6 RELATED WORK

The RV field has developed diverse methodologies, which primarily focus on the synthesis of monitors from system specifications. In a comprehensive survey, Falcone et al. profile over 60 software monitoring frameworks (Falcone et al., 2021). These tools predominantly use temporal logic, often a variant of LTL, to translate system specifications into monitors, with implementations typically in Java, C, or C++ (Havelund and Roşu, 2004; Navabpour et al., 2013; Cimatti et al., 2019). Despite the large number of available tools, a common barrier persists: practitioners are often expected to have a deep understanding in formal methods or a specific tool syntax, which limits their accessibility. Furthermore, only a small number support requirements with timing constraints (Pinisetty et al., 2017; Rajhans et al., 2021).

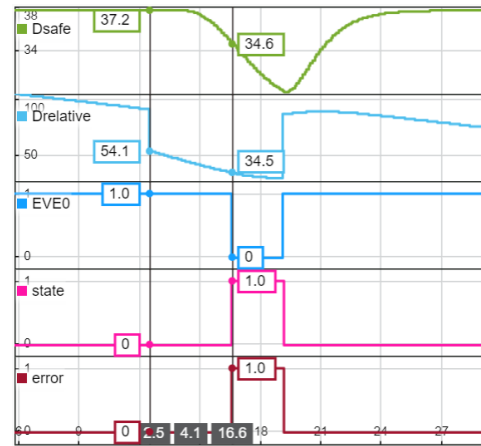


Figure 11: Simulation results for R1 monitor.

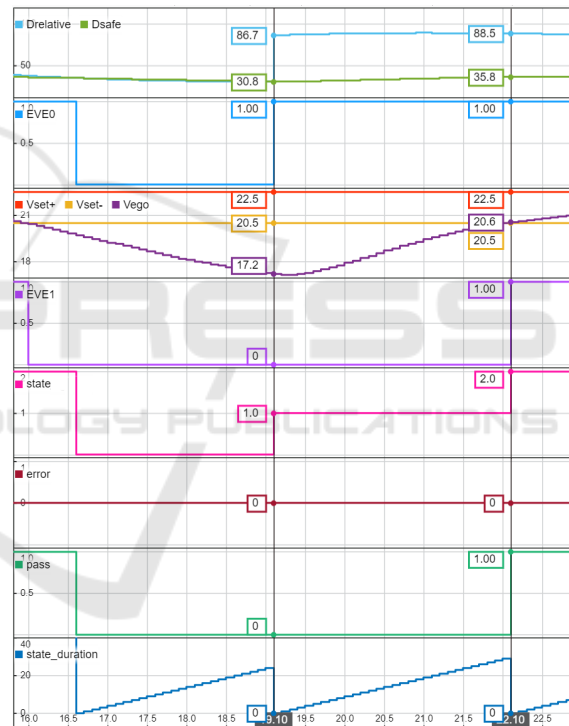


Figure 12: Simulation results for R2 monitor.

Certain frameworks address both functional and timing requirements. For example, RuSTL translates structured English templates into Python monitors, though its application is primarily confined to log files analysis (Khan, 2019). Another example is a toolchain which converts FRETish requirements into C monitors using the FRET, OGMA, and Copilot tools (Perez et al., 2022). This approach is based on analyzing data streams, which is best suited to scenarios with large data transfers. In contrast, GROOT generates automata-based monitors, which have the virtue of simplicity that assists both formal analysis

and comprehensibility by most verification engineers.

7 CONCLUSIONS

This paper presents GROOT, a methodology and framework for automating synthesis and formal verification of RV monitors from structured English specifications, enhancing the accessibility and comprehensibility of RV for practitioners. It incorporates a dual approach for functional and timing requirements. This framework introduces TIMESPEC, a structured English dialect to articulate timing constraints. Monitors may be used during development and/or deployment. This approach bridges the often daunting gap between formal methods and their practical use for real-time embedded systems.

Future work will integrate several monitors, covering both functional and timing aspects, combined with a “monitor of monitors”. We will also conduct a comparative analysis of GROOT-generated monitors with those from alternative methodologies.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation (NSF) 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.
- Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. The MIT Press, Cambridge, MA, USA.
- Baudin, P. et al. (2008). ACSL: ANSI/ISO C specification language.
- Büchi, J. R. (1990). *On a Decision Method in Restricted Second Order Arithmetic*, pages 425–435. Springer.
- Cimatti, A. et al. (2002). NuSMV 2: An open source tool for symbolic model checking. In *Comput. Aided Verification*, pages 359–364, Berlin. Springer.
- Cimatti, A. et al. (2019). NuRV: a nuXmv extension for runtime verification. In *Int. Conf. on Runtime Verification*, pages 382–392. Springer.
- Clarke, E., Klieber, W., Nováček, M., and Zuliani, P. (2012). *Model Checking and the State Explosion Problem*, pages 1–30.
- Cuoq, P. et al. (2012). Frama-C: A software analysis perspective. In *Proc. Int. Conf. Softw. Eng. and Formal Methods*, page 233–247, Berlin. Springer.
- Duret-Lutz, A. et al. (2016). Spot 2.0 – a framework for LTL and ω -automata manipulation. In *Proc. Int. Symp. on ATVA*, volume 9938, pages 122–129. Springer.
- Dwyer, M. B. et al. (1999). Patterns in property specifications for finite-state verification. In *Proc. Int. Conf. Softw. Eng.*, pages 411–420.
- Falcone, Y. et al. (2021). A taxonomy for classifying runtime verification tools. *Int. J. Softw. Tools for Technol. Transfer*, 23(2):255–284.
- Gadelha, M. R. et al. (2018). ESBMC 5.0: An industrial-strength C model checker. In *Proc. ACM/IEEE Int. Conf. Automated Softw. Eng.*, page 888–891.
- Giannakopoulou, D. and Havelund, K. (2001). Automata-based verification of temporal properties on running programs. In *Proc. Int. Conf. ASE*, pages 412–416.
- Giannakopoulou, D., Pressburger, T., Mavridou, A., Rhein, J., Schumann, J., and Shi, N. (2020). Formal requirements elicitation with FRET. In *REFSQ Workshops*.
- Harrison, J. et al. (2014). *History of Interactive Theorem Proving*, volume 9, pages 135–214.
- Havelund, K. (2008). Runtime verification of C programs. In *Testing of Software and Communicating Systems*, pages 7–22. Springer.
- Havelund, K. and Roşu, G. (2004). An overview of the runtime verification tool Java PathExplorer. *Formal methods in system design*, 24(2):189–215.
- Khan, W. (2019). RuSTL: Runtime verification using Signal Temporal Logic. Master’s thesis, University of Waterloo. Available at: <https://uwspace.uwaterloo.ca/handle/10012/14552>.
- Koymans, R. (1990). Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2(4):255–299.
- Leucker, M. et al. (2018). TeSSLa: Runtime verification of non-synchronized real-time streams. In *Proc. Ann. ACM SAC*, page 1925–1933.
- Leucker, M. and Schallhart, C. (2009). A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293 – 303.
- Navabpour, S. et al. (2013). RiTHM: A tool for enabling time-triggered runtime verification for C programs. *ESEC/FSE 2013*, page 603–606.
- 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.
- Pinisetty, S. et al. (2017). Predictive runtime verification of timed properties. *J. of Syst. and Softw.*, 132:353–365.
- Pnueli, A. (1977). The temporal logic of programs. In *18th Annu. SFCS*, page 46–57, USA. IEEE.
- Rajhans, A. et al. (2021). Specification and runtime verification of temporal assessments in Simulink. In *Runtime Verification*, pages 288–296, Cham. Springer.
- Rezvani, B. and Patterson, C. (2023). Differentiated monitor generation for real-time systems. In *Proc. IC-SOFT*, volume 1, pages 353–360. SciTePress.
- The MathWorks Inc. (2023). Simulink: 10.7 (R2023a). <https://www.mathworks.com/products/simulink.html>.
- The MathWorks Inc. (2024). Adaptive cruise control with sensor fusion. <https://www.mathworks.com/help/mpc/ref/adaptivecruisecontrolsystem.html>.