Applying Formal Methods to Build a Safe Continuous-Control Architecture for an Unmanned Aerial Vehicle

Leandro Buss Becker (leandro.bussbecker@manchester.ac.uk)
University of Manchester
Fernando Silvano Gonçalves
Federal Institute of Santa Catarina
Elton Ferreira Broering
Universidade Federal de Santa Catarina
Henrique Amaral Misson
Polytechnique Montréal
Lucas Cordeiro
University of Manchester

Research Article

Keywords: UAV, Formal Verification, Model Checking, Schedulability analysis, Time-correctness, Safety, Implementation-errors analysis, Runtime Monitoring

Posted Date: December 15th, 2023

DOI: https://doi.org/10.21203/rs.3.rs-3668418/v1

Additional Declarations: No competing interests reported.

Applying Formal Methods to Build a Safe Continuous-Control Architecture for an Unmanned Aerial Vehicle

Leandro Buss Becker[1,2*], Fernando Silvano Gonçalves[2,3], Elton Ferreira Broering[2], Henrique Amaral Misson[2,4] and Lucas Cordeiro[1]

[1]Department of Computer Science, University of Manchester, Manchester, M13 9PL, UK.
[2*]Graduate Program on Automation and Systems Engineering, Federal University of Santa Catarina (UFSC), Florianópolis, 88040–900, SC, Brazil.
[4]Extension, Research and Education Department, Federal Institute of Santa Catarina (IFSC) - Campus Tubarão, Deputado Olices Pedra de Caldas, 480, Tubarão, 88704–296, Santa Catarina, Brazil.
[4]Computer Engineering and Software Engineering Department, Polytechnique Montréal, 2500 Chemin de Polytechnique, Montréal, H3T 1J4, Québec, Canada.

*Corresponding author(s). E-mail(s): leandro.bussbecker@manchester.ac.uk;
Contributing authors: fernando.goncalves@ifsc.edu.br; eltonbroering@gmail.com; henriquemisson@gmail.com; lucas.cordeiro@manchester.ac.uk;

Abstract

Cyber-Physical Systems (CPS) are systems composed of computational and physical processes where constant interaction with the surrounding environment exists. Unmanned Aerial Vehicles (UAVs) can be highlighted as a typical example of CPS. It contains devices that sense the surrounding environment (e.g., IMU, GPS) and provide data for the embedded continuous-control software to compute the CPS reactions.

Such reactions are, in fact, actions in the physical (electro-mechanical) process, which occur using actuators (e.g., motors' speed controllers). Such a CPS is typically classified as safety-critical because a failure might have severe implications. Therefore, providing safety guarantees is of utmost importance when designing this application. This paper presents a solution for offering safety guarantees during the design of the continuous-control architecture, which is one of the most critical parts of the CPS. The present proposal applies formal verification (FV) techniques to detect software errors and verify if the architecture is suitable to cope with the real-time requirements coming from the system specification. The first verification round targets individual elements of the architecture, especially the continuous-control algorithm. Therefore, the ESBMC model checker is used; it receives the element's source code as input and can check for a set of language-specific properties, such as memory safety and concurrency vulnerabilities. After making all the individual analyses and performing subsequent corrections, another verification process is started using the UPPAAL model checker, aiming to make the schedulability analysis of the proposed architecture. Finally, we conduct a runtime monitoring analysis using our recently developed RMLib tool. This proposal was successfully used within the design process of a UAV, where different classes of design and implementation problems were detected and further corrected, as detailed in the paper.

Keywords: UAV, Formal Verification, Model Checking, Schedulability analysis, Time-correctness, Safety, Implementation-errors analysis, Runtime Monitoring

1 Introduction

Cyber-physical systems (CPS) are composed of computational and physical processes, constantly interacting with the surrounding environment [1]. A typical CPS example is the autonomous robots/ vehicles, such as an Unmanned Aerial Vehicles (UAV). UAVs, for instance, contain an embedded computational system that "senses" the surrounding environment through sensors (e.g., Accelerometers, Magnetometers, Sonar). It acts in the physical (electro-mechanical) process using actuators (e.g., Brushless-motors with speed controllers) [2]. Search and rescue (SAR), food/ goods delivery, environmental monitoring, power line inspection, and precision agriculture are, among others, typical non-military application examples for UAVs [3].

UAV-related applications are considered safety-critical because a failure might have severe implications, threatening human lives, natural resources, and expensive equipment [4]. Typically, UAV/CPS applications are subject to hard real-time constraints [5], implying that deadlines cannot be violated; otherwise, erroneous actions might occur. For such reason, it becomes interesting, even necessary, to enrich the design process with methods that can assure the time-correctness of the system under design. In this regard, formal verification along

with the UAV/CPS design process becomes necessary, especially during the earlier phases of the development cycle [3].

The paper presents an experience report from applying formal verification to build a safe continuous-control architecture for an UAV. The conducted work uses software tools and models developed by the present co-authors in related works that are further addressed. The aim is to assure the safety and the time-correctness of the generated software components, which is done using implementation errors and schedulability analyses. This initiative is not intended to be seen as an exclusive way of tackling UAV design using formal verification but as a novel approach that has shown to be a valuable asset for the UAV design process, as further detailed.

The core elements of the verification process presented here are described as follows. The Efficient SMT-based Bounded Model Checker (ESBMC) [6, 7] is a context-bounded model checker suitable to detect user-specified assertion failures and additional classes of problems, such as out-of-bounds array access, illegal pointer dereferences, integer overflows, NaN (Floating-point), division by zero, and memory leaks. The UPPAAL tool [8] is used to formally verify the scheduling algorithm specifically for supporting the continuous-control architecture presented in this work. The RMLib [9] is a tool that supports runtime monitoring on top of the FreeRTOS. It can be used to analyze execution traces in offline mode, as well as to perform online analyses.

1.1 Paper Contributions

The novelties/contributions addressed in this paper can be summarized as follows:

- It presents an extended FV process compared to our previous studies [10, 11], which adopts static verification to analyze the source code and the architectural model and also allows the use of runtime monitoring to check for timing faults.
- It details static code verification to search for implementation errors in continuous-control algorithms.
- It describes the results obtained from using this FV process within the design of the continuous-control architecture of a UAV.
- It details the developed continuous-control architecture, which can be used as a reference – even completely reused – within other CPS.

1.2 Paper Organization

This paper is organized as follows. In Section 2, it is presented an overview of the adopted development method, which is the basis for the proposal presented in this work. Section 3 presents the proposed FV process, addressing its workflow and describing each verification step. In section 4, we detail the runtime monitoring tool. The application of the FV process and the runtime monitoring tool for the design of the continuous-control architecture of a UAV, highlighting obtained results, with achievements and limitations, is discussed

in Section 5. Section 6 discusses related studies where FV techniques were applied for CPS/UAV design. Finally, we present our conclusions and possible directions for future work perspectives in Section 7.

2 Overview of the Adopted Development Method

This section provides an overview of the model-based development method for CPS design presented in [12] and that was followed in this work. It originally contained four design steps, but a fifth step is introduced in the present work. Follows a summary of such design steps: (i) system requirements definition; (ii) preliminary design; (iii) detailed design; (iv) implementation, and (v) runtime analysis. Fig. 1 depicts these five steps, including the resulting actions (inside the blocks) and the provided outputs.

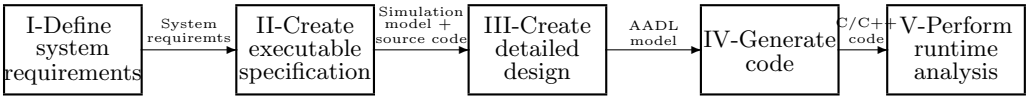

Fig. 1 Main activities and artifacts of the adopted development method (adapted from [12]). The *system requirements* provided in step-I guide the creation of an executable spec. in step-II, which includes the source code of a continuous-control algorithm; then step-III generates an *AADL model* that is suitable to be analyzed; the complete application code is generated in step-IV; finally, in step-V a runtime analysis is performed.

An essential aspect highlighted is that this method suggests adopting different modeling languages to represent systems functionalities and architecture. The rationale is related to the available tools to simulate system functionalities. CPS designers typically prefer using tools that support mathematical modeling and include simulation capacities, like Simulink, Labview, Scilab, and Ptolemy. However, as discussed in [13], such tools are not appropriate to represent the system architecture. For this reason, the adopted development method suggests using a different modeling language to represent the system architecture.

2.1 Definition of the system requirements

The first step of the method is eliciting the system requirements from the sources (typically the stakeholders). This is a non-trivial step since it implies discovering and unveiling the users' needs concerning the system being built. Many techniques can be used in this step, like interviews, task analysis, domain analysis, introspection, brainstorming, observation, and personas. A good survey about elicitation techniques can be found at [14]. There are no hard rules to deciding which techniques to use in these situations. The techniques to be used depend on the judicious evaluation made by the engineer. The result of this step is a specification of the user requirements, both functional and

non-functional, written in natural language (e.g., English). This specification serves mainly as a communication medium between the users and the project members.

2.2 Preliminary design

The next step is to create a preliminary design for the CPS, focusing on obtaining a specification suitable for simulation, both the continuous controller and the related environment. An additional exciting feature regards structuring the CPS regarding components and subsystems. Moreover, the adopted language/modeling formalism could also aid the design team in modeling the possible operation modes of the CPS under design.

The work in [12] heavily relied on using the Simulink simulation tool at this design step, given that it provides a block-diagram language to structure the modeled system. The functional specification can then be represented using block diagrams (blocks from a library or user-defined), Matlab or C source code (used to specify user-defined blocks), and state machines. The work presented a strategy to transform the automatic model from a Simulink specification to an AADL model.

Besides being a commercial tool, Simulink did not provide facilities to build realistic simulation models that could be adapted for different aircraft models. Therefore, the Provant team decided to invest in creating its simulation structure. The first version of this simulator, which is based on ROS 1/Gazebo, was presented in [15]. It is crucial to notice that the continuous-control algorithms are described in C/C++ language, maintaining the practice adopted while using Simulink. A new version of this simulator, based on ROS 2, is currently under development.

2.3 Detailed design

Creating a detailed design is mandatory in any engineering project [16], and it is not different from designing a CPS. At this step, the design team must decide about allocating the functionalities into processes and threads and the reason for deploying such tasks into a target platform, which must also be defined in this design step. Moreover, the different constraints related to the system implementation must be considered, like timing and energy consumption restrictions.

Using different modeling languages in the previous and this design step should allow the development team to gradually change the system representation written in one (more abstract, more informal) language into another (more concrete, more rigorous) representation. The resulting model should have enough details to make it suitable for the code generation phase. Such a model should also be suitable for model-based analysis so that possible design mistakes can be detected and adequately corrected before the code is generated.

The Architecture Analysis and Design Language (AADL) [17] was chosen to be used in this step because it allows expressing in detail the software organization and its target platform. Besides, AADL contains adequate tool support to perform various types of model-based analysis and a proper abstraction level to allow its implementation in a given programming language. As described in [10], it is possible to transform an AADL model into an automaton model suitable to be processed by the UPPAAL verification tool.

2.4 Implementation

For the implementation phase to be conducted appropriately, the model resulting from the previous design step should have enough details to generate code from it and becomes straightforward. This means that programmers or code generation software might be able to interpret it and generate the respective program code in a given target language. For instance, the Ocarina tool [18] can automatically generate code from an AADL model to C, C++, or ADA languages. In the present work, we target the C language.

2.5 Runtime Analysis

The runtime analysis step is devoted to performing execution traces of the program and checking if the desired properties are fulfilled. It can, for example, observe a certain task executed for a longer time than the worst-case execution time established, or it can also observe if the task violated its deadline. The properties check can be done online, i.e., in parallel with the program execution, or offline, i.e., by analyzing the collected execution trace.

3 Proposed Formal Verification Process

Essentially, the proposed FV process involves applying Model-Checking (MC) at the end of steps II and III of the method presented in the previous section, illustrated in Fig. 1. Therefore, two different static verification techniques/-tools are used. The first part of this process relates to static code verification, looking after implementation errors. It uses the source code generated as output from step-II, the continuous-control algorithm, as input. Next, it performs architecture analysis using the AADL model generated as output from step-III as input. In what follows, these two techniques are presented in detail.

3.1 Verifying Implementation Errors in Control Algorithms

This phase conducts model checking on the continuous-control algorithm's source code, utilizing the C programming language as input. The objective is to identify implementation errors, including but not limited to pointer safety, arithmetic overflow, and division-by-zero—issues that are often overlooked by conventional software testing methods [19].

We employ the Efficient SMT-based Bounded Model Checker (ESBMC) [6] for this verification process. ESBMC is an open-source, permissively licensed tool grounded in satisfiability modulo theories (SMT), designed to verify single and multi-threaded C/C++ programs. Within ESBMC, diverse software verification techniques and solvers, such as incremental BMC, k-induction, and SMT solvers like Bitwuzla, Boolector, CVC4, MathSAT, Yices, and Z3, can be applied for program analysis. ESBMC is versatile in detecting various software errors, including out-of-bounds array access, illegal pointer dereferences, integer overflows, NaN (Floating-point), division-by-zero, and memory leaks [20]. Nevertheless, it is important to note that, being a BMC tool, ESBMC is susceptible to resource exhaustion issues, and addressing the scalability of BMC tools for large-scale software remains an ongoing challenge [21].

To mitigate scalability challenges, we offer an additional feature wherein, in addition to the continuous-control algorithm's source code, designers can input range values (lower and upper bounds) for the variables used by the control algorithm. These specified intervals boost the verification process and yield counterexamples in the event of a property violation [22]. By exploring the state space that represents the operational region of the control system, we can achieve more efficient verification outcomes. Furthermore, designers can incorporate user-defined assertions to verify the accuracy of the generated outputs.

3.1.1 Incremental Verification

Here, we exploit the incremental verification algorithms implemented in ESBMC [7, 23]. Consider a C program P be modeled as a finite transition system M, which is defined as follows:

- $I(s_n)$ and $T(s_n, s_{n+1})$ represent the equations over program's state variable $s_i \in S$, which constrains the initial states and transition relations of M;
- $\phi(s)$ represent the equation that encodes states satisfying a given safety property, which verifies language-specific properties and user-defined properties;
- $\psi(s)$ represents the equation that encodes states satisfying the completeness threshold, i.e., states corresponding to the termination. Note that $\psi(s)$ consists of unwindings no more profound than the maximum number of loop iterations in P.

In each step k of the incremental verification algorithm, two checks are conducted: the base case $B(k)$ and the forward condition $F(k)$. $B(k)$ represents the standard BMC and it is satisfiable *iff* P has a counterexample of length k or less:

$$B(k) \Leftrightarrow I(s_1) \wedge \left(\bigwedge_{i=1}^{k-1} T(s_i, s_{i+1}) \right) \wedge \left(\bigvee_{i=1}^{k} \neg\phi(s_i) \right). \tag{1}$$

The forward condition verifies termination. It checks the completeness threshold $\psi(s)$ that must hold for the current k. Note that if $F(k)$ is

unsatisfiable, P has terminated:

$$F(k) \Leftrightarrow I(s_1) \wedge \left(\bigwedge_{i=1}^{k-1} T(s_i, s_{i+1}) \right) \wedge \neg\psi(s_k). \tag{2}$$

Note further that no safety property $\phi(s)$ is verified in $F(k)$ since they are checked for the current k in $B(k)$. Lastly, the inductive check $S(k)$ is unsatisfiable if, whenever $\phi(s)$ holds for k unfoldings, it also holds for the next unfolding of P:

$$S(k) \Leftrightarrow \exists n \in \mathbb{N}^+. \bigwedge_{i=n}^{n+k-1} (\phi(s_i) \wedge T'(s_i, s_{i+1})) \wedge \neg\phi(s_{n+k}). \tag{3}$$

Here $T'(s_i, s_{i+1})$ is the transition relation after eliminating the loop variables [24].

Through $B(k)$, $F(k)$, $S(k)$, and $\pi(k) \Leftrightarrow B(k) \vee [F(k) \wedge S(k)]$, the incremental verification algorithm bmc_{inc} to falsify or verify programs at a given k is:

$$\text{bmc}_{\text{inc}}(P, k) = \begin{cases} P \text{ is unsafe}, & \text{if } B(k) \text{ is SAT}, \\ P \text{ is safe}, & \text{if } \pi(k) \text{ is UNSAT}, \\ \text{bmc}_{\text{inc}}(P, k+1), & \text{otherwise}. \end{cases} \tag{4}$$

3.2 Architecture Verification (Schedulability analysis)

This step performs model checking using as input the software architecture model (AADL model) designed at the end of step-III of the adopted design method. This step requires the AADL model to be transformed into a set of Timed Automata (TA). To transform an AADL model into TA it is adopted the model transformation mechanism described in [10].

The static verification process presented here is performed using UPPAAL [8], an integrated environment for modeling, simulation, and verifying real-time systems. UPPAAL is an intuitive tool and, at the same time, very efficient, allowing validation through graphic simulation and verification through automatic model checking. Its interface is implemented in Java and allows the modeling of systems in TA extended with integer variables, structured data types, and channel synchronization.

This consists of finite state machines with a clock, in which the model is composed of states (locations) that represent situations or modes of operation and of transitions that connect two or more states. This network of states and transitions forms the system's behavior, where for each transition, it is possible to define a set of rules and conditions that the system must meet to enable a state change. The simulation stage of the model through UPPAAL allows the developer to test the various paths the system can take during its execution and observe whether it is not working as planned [25].

In the context of an MC technique, let M be a Kripke structure (the system model), and considering f be a formula of temporal logic (the desired property), the problem is to find all states s of M such that $M,s \models f$ [26].

Properties in UPPAAL are formalized using the Timed Computation Tree Logic (TCTL) language [27], which is a fragment of CTL [28] that uses clock variables and clock constraints to specify timing behaviors reasoning about properties of the TA. TCTL allows to construct the queries using the quantifiers A (for all paths) and E (exists a path), the temporal operators $\diamond$ (eventually) and $\Box$ (always), beyond the Boolean connectives (*and, or, implication*).

To facilitate verification analysis, the properties can be organized in patterns as follows:

- **Reachability**: it allows us to evaluate if there is a path where at least one state satisfies a given α property. A possible notation using CTL is:

  $$E \diamond \alpha$$

  where E is the quantifier that means "exist a path", the temporal operator $\diamond$ represents "eventually", and α is the property. For example, considering that the system has a state called *Running*, this is an essential state for the correct operation of the application. It is possible to check the property against the model using the reachability classification as the expression:

  $$E \diamond model.Running$$

- **Safety**: this property states that "something bad never happens", meaning that the system should never reach the state where this property is satisfied. TCTL can express the following formulae:

  $$A \Box \alpha$$

  where the quantifier A means "for all paths", the temporal operator $\Box$ represents "always" and α is the desired property. A typical example of this property is the absence of deadline misses on the system threads:

  $$A \Box \; not \; deadline\text{-}miss$$

- **Liveness**: this enables checking whether, considering all paths, states exist that satisfy a given α property. It uses the following TCTL formulae:

  $$A \diamond \alpha$$

  where A is the quantifier "for all paths", the operator $\diamond$ represents "eventually" and α is the desired property. For example, considering a system has a "*Ready*" state, it is possible, through the liveness specification, to check for all execution paths if this state is reached in one moment in the future. This property can be expressed by:

$$A \diamond model.Ready$$

MC in UPPAAL is usually fast and intuitive, especially considering the counterexamples generation if the model does not satisfy the property. However, it can be subject to state-space explosion, which is observed on models with a high level of complexity and generates many possible state combinations; thereby, the tool cannot evaluate the property.

To avoid this state explosion situation, some design techniques can be applied. One of these techniques is creating an Observer Automata, restricting a system's legitimate behavior to those accepted by such observers. This technique is used in the schedulability analysis model presented in the next section to verify whether higher-priority threads will always be executed before lower-priority ones.

4 Runtime Monitoring Library

In this section, we present our proposed RMLib[1], a lightweight and non-intrusive solution for implementing runtime monitoring (RM) in embedded systems, allowing users to perform customization based on specific architecture characteristics. Although architecture-independent and compatible with multiple platforms, the proposed tool was developed to support the FreeRTOS operating system. Overall, the developed library allows to address the following issues:

1. Monitor if all tasks in the system are executed without missing deadlines. This is important to assess the real-time performance of the system and validate the scheduling and timing properties.
2. Measure (and reason about) the execution times of the tasks. This helps identify potential bottlenecks or performance issues and provides insights into the system's timing behavior.

RMLib can support two distinct operation modes: online verification and offline verification. In offline verification mode, the monitors run concurrently with the application code, collecting timestamps at specific points of interest. These timestamps are then stored for later exportation and evaluation of monitored constraints using external software. Offline verification is less intrusive and imposes less overhead on the system, but it is not helpful to identify situations where immediate response is required.

On the other hand, online verification mode operates similarly to offline verification regarding event monitoring. However, verifying system restrictions occurs online and does not require transferring all monitored events – only the verification results are exported. Online verification offers additional features such as timestamp analysis, exporting only cases of timing violations, and exporting system information when violations occur. However, it imposes the

[1] https://github.com/EltonBroering/RMLib

system's overhead and, depending on the configuration, it can impact the system's performance.

5 Case Study: UAV Continuous-control Architecture Design

This section describes the application of the proposed FV process within the context of a project aiming for designing the UAV which dynamics is detailed in [29]. More specifically, it aims to present the design of a safe continuous-control architecture suitable not just for the UAV under the design at our project but also for other CPS, as further detailed. All the assets used in this study, which include the developed models and the verification tools presented in section sec. 3 are left publicly available[2].

The software part of the proposed continuous-control architecture is illustrated in the AADL model presented in Fig. 2. It consists of five concurrent Processes/Threads, as follows. The *High-Level Command (HLC)* is a non-preemptive thread responsible for interfacing with higher-level control entities, such as an intelligent agent, which typically runs in separate hardware. This task is critical because, whenever activated, it must execute before the control thread so that it provides input data for the control algorithm. The *Sensing* is a non-preemptive thread responsible for interfacing with the sensors, collecting data, and making its pre-processing so that this data becomes ready for input by the control algorithm. The *Control* is a non-preemptive thread executing the continuous-control algorithm. It uses as input sensor data generated by the *Sensing* thread and commands from the *HLC* thread. It results in actuation values that must be sent to the *Actuation* thread. The *Actuation* is a non-preemptive thread responsible for interfacing with the actuation components, such as the two Electronic Speed Controllers (ESC) attached to the brushless motors. Given that the ESC demand periodically receives a command, the *Actuation* thread is critical because its slack time (time between being ready for execution and starting executing) must be close to zero. The *Telemetry* is a preemptive thread that sends monitoring data to a remote base station. It is up to the UAV operation team to set up the amount of data transmitted by this thread, which can lead to very high execution times. Here, we have established the worst-case execution time.

Although the graphical AADL model representation in Fig. 2 does not show the timing properties of the model elements (threads), such information is indeed present in the AADL code. Tab. 1 summarizes the timing and scheduling properties of the proposed continuous-control architecture threads. If we use the maximum allowed execution time for the *Telemetry* (T5) thread, this set achieves a processor utilization factor of 100%. Regardless of the *Telemetry* thread execution time (either maximum or reduced), there are critical instants that let the processor fully utilized. In this study, the critical instant last $12ms$,

[2] https://provant.paginas.ufsc.br/formal-verification/

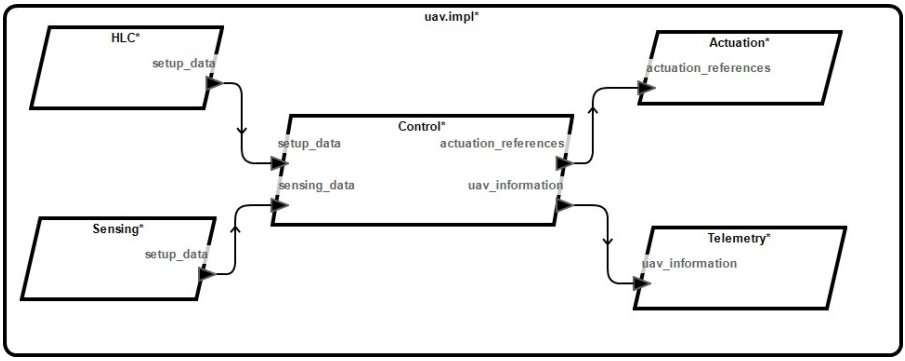

Fig. 2 AADL model of the UAV continuous-control architecture. In the center, there is the *Control* thread, which receives input from *Sensing* and *HLC* threads. Its output is sent to *Telemetry* and *Actuation* threads.

Table 1 Thread-set of the proposed continuous-control architecture.

	Period	Deadl.	WCET	Utilizat.	Prior.	Preemptive
T1-Actuation	12	12	2	0.166	5	NO
T2-Sensing	12	12	3	0.250	4	NO
T3-Control	12	12	4	0.333	2	NO
T4-HLC	120	120	4	0.033	3	NO
T5-Telemetry	600	600	130	0.216	1	YES

and happens every $120ms$. Fig. 3 illustrates the Gantt diagram that was generated with the Cheddar [30] plugin of the AADL editor. Please observe that the threads in the diagram are in the same order as presented in Tab. 1, with T1 (Actuation) on top and T5 (Telemetry) on the bottom. In this model the critical instant happens every $120ms$, i.e., whenever the *HLC* (T4) thread gets ready for execution, so it must be executed before the *Control* (T3) thread. It is worth calling readers' attention to the $12ms$ timing windows immediately before (starting at $108ms$) and immediately after (starting at $132ms$) the critical instant. Such windows are used to execute the single preemptable task, *Telemetry*. Finally, one should observe the fact that the *Actuation* thread executes without jitter.

As an embedded computing platform for the proposed architecture, it adopted the *STM32F4DISCOVERY* development board [31], which has a single-core 32-bit ARM Cortex-M4 processor with floating-point-unit (FPU) [32]. The threads' software was developed in the C programming language and runs on top of the FreeRTOS Operating System (version 7.5.2) [33]. It is a low footprint OS that is proper for executing real-time threads in a priority-driven manner.

The following two subsections provide details about the application of the FV in the proposed architecture. In sec. 5.1 it is presented the use of ESBMC to detect possible unwanted implementation errors and in sec. 5.2 it is presented the use of UPPAAL to detect specifications problems related to deadline

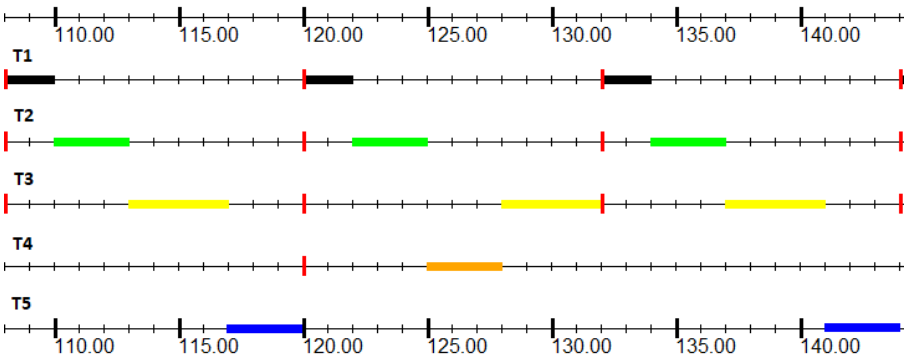

Fig. 3 Gantt diagram of the proposed thread-set execution, with the critical instant in the center ($120ms - 132ms$).

misses. To finish the study, sec. 5.4 discusses how the proposal can be used in different CPS applications.

5.1 Verification in ESBMC

The *Code Verification Phase*, as outlined in Section 5.1, is now implemented. This phase focuses on the Linear Quadratic Regulator (LQR) control algorithm, as described in [29, 34], developed in the C programming language and previously validated using the Simulink tool. The algorithm is integrated into the *Control* thread illustrated in the AADL model in Fig. 2.

The static code verification process, executed using the ESBMC tool, specifically targets the aforementioned LQR control algorithm. The program analysis is conducted in an open-loop fashion, with a single execution of the control loop for each input configuration. Note that ESBMC tests all possible values of the input variables within the control loop. These variables, encapsulated within a data structure named *iInputData*, consist of 48 floats, 18 integers, and 4 unsigned integers. These values originate directly from UAV sensors or indirectly from such measurements. To enhance the efficiency of the verification process, bounds (upper and lower limits) were defined for each of these 70 variables based on the simulation analysis conducted in the ProVant project.

The evaluation of properties listed in Table 2 encompasses various checks ESBMC supports. *Floatbv* indicates the use of floating-point computation by ESBMC when scrutinizing assertions in the code. *No bounds* signifies the absence of array out-of-bounds violations detected by ESBMC. *No assertions* implies that ESBMC did not examine built-in or user-provided assertions in its operational models. *No div by zero* denotes the absence of division by zero computations detected by ESBMC. *Pointer* indicates that ESBMC found no memory safety violations. *No align* means ESBMC did not identify any pointer alignment issues. *No pointer relation* indicates that ESBMC did not assess

Table 2 Evaluated Properties and Results.

	Property	Solver	
		Incremental	K-induction
1	floatbv	Passed	Passed
2	no bounds	Passed	Passed
3	deadlock	Passed	Passed
4	no assertions	Passed	Passed
5	no div by zero	Passed	Passed
6	no pointer	Passed	Passed
7	no align	Passed	Passed
8	no pointer relation	Passed	Passed
9	memory leak	Passed	Passed
10	NaN	Failed	Failed
11	overflow	Failed	Failed
12	data races	Passed	Passed
13	lock order	Passed	Passed

whether two pointers point to the same object. *Memory leak* checks for memory leaks in the underlying program. *NaN* suggests issues identified by ESBMC related to floating-point computation, particularly instances of Not-a-Number, which may be associated with division by zero, for example. *Overflow* checks for arithmetic over- and underflows. *Data race* checks for simultaneous reading and writing to a common memory location by different threads. *Lock order* assesses the ordering of lock acquisition by different threads.

The table also provides insights into the verification strategy and outcomes. The term *Incremental BMC* denotes a process where the program is incrementally unwound until a bug is uncovered or until the completeness threshold is met. This approach ensures that smaller problems are addressed sequentially rather than relying on an arbitrary upper bound for verification. However, the incremental algorithm has its limitations. Notably, the BMC must redo parsing, generation, and solving for each bound k without leveraging records of previous steps *1* to *k-1* when addressing k. Despite the advent of incremental solving in the 1990s [35], the challenge of efficiently reusing information learned from previous instances persists.

k-Induction employs BMC to identify property violations and establish program correctness. At each step k of the k-induction algorithm, three checks are performed: the base case $B(k)$, the forward condition $F(k)$, and the inductive step $S(k)$. In the base case $B(k)$, ESBMC endeavors to identify a counterexample with up to k loop unwindings. In the forward condition $F(k)$, ESBMC verifies whether loops have been completely unrolled and whether the specified property holds in all states reachable within k unwindings. The inductive step $I(k)$ ensures that if the property holds for k unwindings, it also holds after the subsequent unwinding of the system [24].

As shown in Table 2, ESBMC has found violations related to arithmetic overflow and "Not a Number". In particular, arithmetic overflow occurs when

the result of an arithmetic operation exceeds the range that can be represented by the data type used to store the result. In this case, ESBMC has found a computation result that is too large to be represented within the constraints of the underlying control algorithm data type, causing the value to wrap around or become undefined. For the "Not a Number" violation, which is a special floating-point value that represents the result of an undefined or unrepresentable operation, ESBMC has found a mathematical operation in the underlying control algorithm that cannot produce a meaningful result.

Overall, we can also analyze counterexamples generated by ESBMC against the new instrumented program to pinpoint faulty lines in the UAV controller implementation, as previously done by Alves et al. [36]. This involves searching for diagnostic values corresponding to actual lines in the embedded UAV software. Such an approach can further enhance debugging processes for UAV-embedded software by indicating which lines require correction and identifying the values leading to successful execution. However, we defer the exploration of this research direction to future work focused on automated UAV software fault localization and debugging.

5.2 Verification in UPPAAL

Overall, three steps are required to perform the envisioned formal verification: *modeling*, *formalization of properties*, and *verification*. Regarding the modeling in the present study, as stated in the previous section, the modeling step begins with the AADL model generated as output from the step-III of the adopted development method (see Fig. 1). It was also explained that such AADL model must be transformed into a set of Timed Automata (TA) to be processed by UPPAAL.

As a result of this transformation, templates that describe our architecture were generated. It consists of an auxiliary scheduler TA and five instances of the TA representing the generic thread behavior, with specific parameters for each thread (see Tab. 1). It was adopted in this work as a simplified thread template compared to the one presented in [11], as it does not consider shared resources. However, this simplification does not mean that we lost analysis capacity, as further explained.

Most of the threads share data among them and, therefore, are subject to race conditions [37]. However, as just mentioned, it was adopted here a simplified thread model that does not support shared resources. The solution to this problem is avoiding preemption within those threads accessing shared data, thus creating implicit mutual exclusion areas. Such a solution mixes preemptable and non-preemptable threads, typically making analytical schedulability analysis much more complex, sometimes impossible. However, this is not an issue in the present work, given the use of Model Checking.

The AADL model used in the present study is depicted in Fig. 2, and its source is available in [38]. It was transformed into TA using the AADL–to–TA transformation tool presented in [10]. The generated TA corresponding to the simplified thread template is illustrated in Fig. 4. It represents periodic and

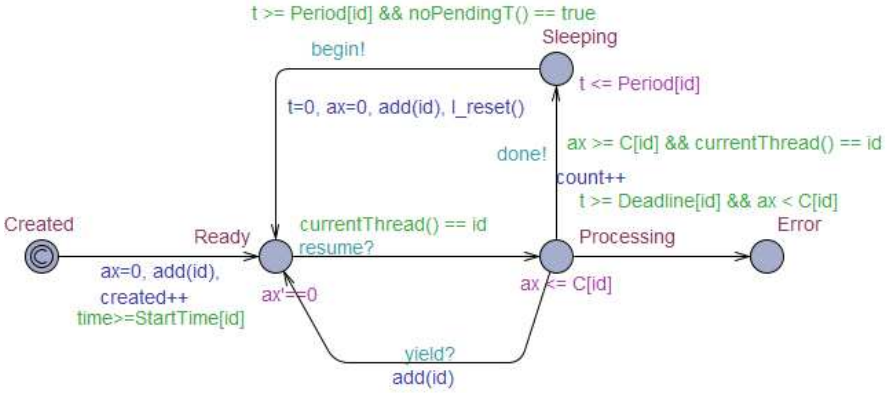

Fig. 4 TA of the thread model on UPPAAL. When in *Ready* state, the thread can be selected for execution. When selected, it enters the *Processing* state. If it completes the execution, it goes to *Sleeping*, where it waits for the next period according to the adopted periodic model. If it is not completed by the deadline, it reaches the *Error* state. It can also be preempted (*yield* transition).

preemptive thread execution, monitoring deadlines missed. As the thread is created and its start-time is reached, it enters the *Ready* state, meaning that it is added to the scheduling queue and can be selected for execution. When selected for execution, the thread goes to the *Processing* state. If preempted by a higher priority thread, it returns to *Ready*. If the deadline is reached without completing the execution, it enters the *Error* state. If the thread completes its execution, it goes to *Sleeping*, going back to *Ready* as soon as its period is once again reached.

Different from the modeling step, which is performed automatically, the *formalization of properties* is usually carefully conducted by experts. As previously mentioned, in UPPAAL the set of properties to be formally verified must be specified in TCTL. Table 3 presents the set of properties verified in the present study. They are characterized as safety properties essential for the system's reliability.

The first property analyzes the schedulability of the task set, proving that there are no deadline misses, i.e., no thread reaches the *Error* state in the proposed architecture. The second property proves that the adopted strategy to avoid race conditions is successful, i.e., those threads that share memory (T1–T4) will not be preempted once they start executing. When changing the query to include T5 the result shows that the property is not satisfied, and the counter-example points to the moment that T5 is preempted.

The third property relates to the *T1–Actuation* thread, which cannot have jitter to start executing. This comes from the fact that actuation within a UAV is very timing-sensitive and does not allow experiencing delays. The derived property states that while in the Processing state, the thread general clock t

Table 3 Properties formally verified with UPPAAL.

Spec.1	Threads T1–T5 exec. time will never exceed their deadline, i.e., deadlines will not be missed (*Error* state will not be reached).
Query	$A \square\ forall(i : 0 - 4)\ not\ T\_thread(i).Error$
Result	Property is satisfied
Spec.2	Threads T1–T4 will never be preempted, i.e., they will not be in *Ready* state with execution time (ax) greater than zero.
Query	$A \square\ forall(i : 0 - 3)\ not(T\_thread(i).Ready\ and\ T\_thread(i).ax > 0)$
Result	Property is satisfied
Spec.3	While executing, the overall clock (t) of thread T1 will be synchronized with its execution clock (ax), meaning that it will not experience execution jitter.
Query	$A \square\ not(T\_thread(0).Processing\ and\ T\_thread(0).t > T\_thread(0).ax)$
Result	Property is satisfied
Spec.4	Covering all the threads execution period ($600ms$), thread T1 will always be completed before the threads T2–T5.
Query	$(time < 600) - - > forall(i : 1 - 4)\ T\_thread(0).count >= T\_thread(i).count$
Result	Property is satisfied

is synchronized with the clock related to the execution time ax. This property does not hold for the other threads (T2–T5).

The last verified property verifies the order in which the threads are executed so that it becomes possible to prove that they are executed exactly as depicted in Fig. 3. In other words, it is necessary to guarantee that higher-priority threads will always have preference over lower-priority ones. To verify this property, it was necessary to create an observer, which consists of an additional mechanism to "count" the thread's execution. It is indeed an overhead for the verification process, increasing the number of states. However, it does not significantly increase the execution time of the present study. Moreover, it was not possible to derive a query that covers the overall cases, like in the first three properties. The example provided shows that instance i of T1 will have preference over the same instances i of T2–T5. Additional similar queries were provided for threads T2, T3, and T4 to complete our analysis.

The conducted schedulability analysis, which shows that the proposed task set is indeed schedulable, would be very difficult to be done in analytical means. This comes from the fact that the adopted parameters for threads do not suit a single scheduling policy. Taking the well know Cheddar [30] scheduling-analysis tool, for example, it is not possible to mix preemptable and non-preemptable threads. Instead, either one uses a scheduling policy that does not allow preemptions, or preemptions should be allowed for all tasks.

For instance, the designer can select a preemptable scheduling policy and establish critical sections that last the execution of those non-preemptable threads. However, there is no means to make an analytical schedulability analysis for such a situation, so only simulations would be possible. However, simulations do not provide strong guarantees. Besides, the results (simulation logs) can be challenging to be analyzed.

5.3 Runtime Monitoring Results

Our RTMLib tool was used to monitor the timing behavior from all 5 threads presented in Table 1. The present analysis allowed us to:

- Validate the two execution modes of the library: Online and Offline. This helped to ensure that the library works properly and produces accurate results in both operation modes.
- Validate the scaling model of the system by examining the behavior of the Communication Task in periodic or continuous operation modes. This allowed to evaluate the impact of different execution configurations on the system's overall performance and responsiveness.
- Validate the system's behavior in the presence of an additional asynchronous task, which served as a "disturbance" to the system. It allowed to assess the system's robustness and ability to handle unexpected events or task interactions.

More importantly, the conducted analysis involved examining the tasks' executions and judging if they were executed within their WCET and deadline restrictions. The verdict is determined based on the following criteria:

- If the difference (Δ) between task's finish and release times is less than or equal to the task's deadline, the verdict is $true$. Otherwise, if Δ exceeds the deadline, the verdict is $false$.
- Another Δ is calculated as the difference between tasks' start and finish times. If it is less than or equal to the WCET the verdict is $true$, otherwise it is $false$.

Another specific point of analysis relates to the Actuation task ($T1$), given the need for periodic reception of a command by the ESC. Task $T1$ is the most critical in our proposed architecture because the UAV might fall if it does not work properly (within the defined time-limits).

In summary, the results from all experiments (on the different operation modes of our tool) showed no timing-constraints violations during the executions. It can also be observed that, in most cases, the Actuation task ($T1$) is executed immediately after it is released. The maximum waiting time observed was of 3 ms, which indeed is not a problem, as it allows $T1$ to complete its execution much ahead of its deadline. Follows more details about the conducted experimentation.

Offline Mode: in this experiment the Communication task ($T5$) was executed whenever the higher-priority tasks were not executing, so that the processor was fully utilized – it never became idle. Using the offline operation mode in this way is advantageous because it allows to handle large amounts of log events in an efficient manner, reducing memory utilization for storing events.
Online Mode: in this experiment the task $T5$ was executed for no longer than its defined WCET. This execution model showed to be highly effective for the Online Verification Mode, given that only a minimal amount of data

needed to be exported, with the verdicts being the only exported data in this case. Once again, the task model and scheduling policy proposed in this work showed to perform properly, as no time constraint violations were observed.

Online Mode with Non-Periodic Task: in this experiment, a new test scenario was created to examine the proposed scheduling model's operation limits and assess its performance when faced with disturbances. The aim was to create a scenario where temporal violations will occur in the system under execution due to unforeseen circumstances. This scenario is also helpful for two other reasons: (i) it allows testing the monitoring of non-periodic tasks, and (ii) it validates additional features for RMLib, such as the Online Verification feature of exporting only cases with timing constraints and the ability to export the status of the system tasks.

This scenario was materialized by manually inserting a higher-priority, non-periodic task into the system. Such a task, which has $ID - 6$, has a WCET of 8 ms and a deadline of 15 ms, and is triggered whenever a critical external event occurs, serving as a failsafe action. Listing 1 presents the system output where the asynchronous task can be observed, showing a timing violation (*StatusDeadline* variable set to 1). This highlights the ability of the RMLib to capture and analyze such corner cases.

Listing 1 Temporal restriction violation illustration at the exported data log. Having the *StatusDeadline* variable set to 1 means deadline violation.

```
{'TaskIdentifier':1,'TimeStamp':41442,'ExecTime':9,
'CtdTask':8916,'StatusWCET':0,'StatusDeadline':1}
Dummy Actuation R 4 223 5
Asynchronous Dummy Sensing R 4 223 4
IDLE R 0 237 7
Led B 3 223 3
Communication B 1 220 2
Controller B 2 223 1
{'TaskIdentifier':1,'TimeStamp':55317,'ExecTime':7,
'CtdTask':15535,'StatusWCET':0,'StatusDeadline':1}
Asynchronous IDLE R 0 237 7
Dummy Sensing B 4 223 4
Dummy Actuation B 4 223 5
Led B 3 223 3
Communication B 1 166 2
Controller B 2 83 1
```

5.4 Reusing the Proposed Architecture

The proposed continuous-control architecture can be reused in different CPS applications. For instance, in Provant, a new UAV is under construction, and the aim is to reuse this proposal. Besides, there is a new project in the scope of smart energy grids being started at UFSC, and it should also benefit from the architecture presented here.

The first essential point to reusing the proposed architecture is designing a proper control algorithm. This implies modifying the inputs and outputs of the control algorithm. Possibly, the algorithm itself will also change. Consequently, all the steps related to the control algorithm analysis using ESBMC, presented in sec. 5.1, must be re-conducted.

Modifying the inputs and outputs of the control algorithm necessarily affects the connections among the AADL process presented in Fig. 2. However, the most critical adjust for performing the schedulability analysis relates to properly modifying the timing parameters of the system, especially the execution times.

When reusing the proposed-architecture in the same CPS domain, such as using it in a new UAV, it is likely that the periods and deadlines of the core threads (T1,T2,T3) will not change, given that the system dynamics are mostly the same. However, when the system dynamics are different, like the case in the smart energy grid, it is required that such core threads' periods and deadlines change as a multiple of the original ones. It is expected that such new periods will be higher, not smaller, than then originals. This implies increasing the critical instant at a multiple of the $12ms$ one used in the study previously presented. Once this adjustments are finished, the designer can conduct the schedulability analysis with UPPAAL presented in sec. 5.2.

6 Related Works

Several related studies are covering the use of FV in scenarios involving CPS and UAVs (see [39–43]). Analyzing such works makes it possible to state that the MC technique is widely used, but there is no consensus on which tool is the best in each case.

In [39], authors apply FV to validate the model of the related CPS, named "environment". Next, the authors perform runtime verification to validate the model against the real environment. However, this solution does not cover the continuous-control layer of the software architecture as it tackles higher-level decision-making, i.e., the system's discrete behavior. Furthermore, it does not analyze timing aspects, such as deadline violations.

A model-based integration framework for modeling and verifying the timing properties of the UAV flight control system is proposed in [40]. This study models the software system using a class diagram and state chart, aided by a model transformation process that could be verified using formal verification tools. In particular, the formal verification was applied by Z/EVES and PVS, tools for analyzing formal specifications using theorem-proving techniques [44]. To check the system, a PTA (Probability Timed Automata) model was used, and a mathematical model of real-time reliability was created to perform the formal proof.

Authors in [45] proposed the application of Bounded Model Checking (BMC) using their proposed DSVerifier tool and targeting system control [46],

especially regarding low-level implementation errors related to digital controllers and hardware compatibility. The proposed tool is applied in UAVs and aims to investigate implementation problems in digital controllers designed for aircraft systems. The obtained results showed flaws in UAV attitude control software applied to aerial surveillance. This type of verification is efficient in finding errors, similar to the testing techniques. However, unlike our proposed approach, it does not prove the absence of implementation flaws. This proof is essential to increase the system's reliability. Additionally, such study also does not consider the schedulability analysis of the system's tasks, thus providing no formal guarantees concerning the timing constraints.

In [41], the authors have used MC to verify the correctness of the UAV behavior in a scenario of a given mission performed by multiple UAVs in cooperation. The properties were written in the formal language CTL, which allows expressing behaviors in a branching time. The verification was performed using the SMV model checker [47]. However, despite presenting some properties to be checked, it was observed that the authors did not present the results of this verification. Consequently, nothing can be concluded about the correctness of the proposed (deployed) system.

Regarding the application of verification techniques in scenarios of multiple UAVs for specific missions, it should also be mentioned [42]. In this case, the author used the MC technique using the SPIN model checker [48]. The requirements definition was formalized through linear-time temporal logic (LTL) and the scenario modeling in PROMELA. The verification was carried out in three different scenarios. The first is the centralized cooperative control scheme, the second is the same decentralized scheme, and the third considers high-level mission planning. In addition to presenting the results, the author discusses the difficulties encountered in applying MC using SPIN and the strengths and weaknesses of the resources used for modeling and describing properties. One of the limitations cited is scenario modeling. For example, only a certain number of aircraft and base stations are considered to not fall into the state explosion problem. However, the authors did not provide complementary methods to avoid such limitations.

Instead of using MC for verification, in [43], the authors presented the R2U2 framework for runtime monitoring of Unmanned Aerial Systems (UAS) security properties. The framework monitors the code via information sent over the system bus. It uses observers based on properties written in LTL and its extension for deadline with real-time, the Metric Temporal Logic (MTL) [49]. A hybrid architecture based on software and hardware was proposed to implement this verification method using an FPGA. As a result, simulations were performed where several scenarios of system attacks were considered. Although it has presented successful results, this framework is used for a specific UAV using a NASA bus system called NASA CFS/CFE, thus restricting its application to other systems.

The runtime verification (RV) toolchain called *Rmtld3synth* is presented in [50], being the first RV framework for real-time embedded systems that can

cope with explicit time and durations. It is based on the restricted fragment of Metric Temporal Logic with duration (MTL-$\int$) [51], which allows to express properties considering explicit time and temporal order of durations of the system's states and yields a three-valued verdict (true, false, unknown). In addition, this language can specify tasks' behavioral properties, considering their time constraints. The work presents a study related with the monitoring of the *PX4 autopilot system* for an *ArduCopter* UAV with the *Pixhawk* [52] flight controller.

In contrast to previous studies, which suggest a single verification path, we tackle two different formal verification approaches in this work that significantly improve the safety of the system under design. On the one hand, an automated transformation process extracts the timed automata from the AADL model. Based on this model, several properties may be evaluated, including the more general timed characteristics like deadlock absence. On the other hand, using ESBMC supports the evaluation of application code, significantly improving the safety of the generated application code.

7 Conclusions and Future Works

This paper presented a formal verification (FV) process devoted to improving the safety of the software architecture within CPS. Such an FV process is used to build a safe continuous-control architecture for a UAV. The proposed architecture is, on its own, a valuable contribution to this paper, given that we understand that it is generic enough to be used within other continuous-control/CPS applications.

The proposed FV process has shown to be very useful in improving the safety of the proposed continuous-control architecture. On the one hand, ESBMC allowed for the identification of implementation errors in the continuous-control algorithms. Besides the benefits achieved in this specific project, we understand that the control algorithm design team's constant use of this tool can improve programming style so that errors can be detected earlier in the development process. On the other hand, the schedulability analysis infrastructure developed in UPPAAL provides safety guarantees for the designed architecture. Moreover, as previously discussed, the proposed model-checking-based schedulability analysis overcomes existing analytical schedulability analysis techniques' limitations, which occurs by allowing full processor utilization (up to 100%) while mixing preemptable and non-preemptable tasks and by providing counter-examples when the task-set is not schedulable.

The developed Runtime Monitoring library was able to collect events of interest and analyze them. Such events are related to the tasks' timing properties, like periodicity, execution time, and deadline misses. Making the library more generic to cope with any event types should be targeted in our future works.

Additional future work efforts should be divided in two different directions. The first one relates to ESBMC. The intention is to provide means to facilitate the identification of faulty lines in the analyzed code, given that this is currently a very laborious task. Another direction relates to providing runtime verification support so that formal verification is adopted in the proposed process in every possible way, allowing it to reach the maximum possible safety levels. Efforts in this direction have already started, targeting the FreeRTOS operating system [53].

References

[1] Derler, P., Lee, E.A., Vincentelli, A.S.: Modeling cyber–physical systems. Proceedings of the IEEE **100**(1), 13–28 (2012)

[2] Lee, E.A., Seshia, S.A.: Introduction to Embedded Systems: A Cyber-physical Systems Approach. Mit Press, Cambridge, U.S. (2016)

[3] Austin, R.: Unmanned Aircraft Systems: UAVS Design, Development and Deployment. John Wiley & Sons, New York, U.S. (2011)

[4] Knight, J.C.: Safety critical systems: challenges and directions. In: Proceedings of the 24th International Conference on Software Engineering. ICSE 2002, pp. 547–550 (2002)

[5] Stankovic, J.A.: Misconceptions about real-time computing: a serious problem for next-generation systems. Computer **21**(10), 10–19 (1988). https://doi.org/10.1109/2.7053

[6] Gadelha, M.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: ESBMC 5.0: An industrial-strength C model checker. In: 33^{rd} ACM/IEEE Int. Conf. on Automated Software Engineering (ASE'18), pp. 888–891. ACM, New York, NY, USA (2018). https://doi.org/10.1145/3238147.3240481

[7] Gadelha, M.R., Monteiro, F., Cordeiro, L., Nicole, D.: Esbmc v6. 0: Verifying c programs using k-induction and invariant inference: (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems: 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part III 25, pp. 209–213 (2019). Springer

[8] Behrmann, G., David, A., Larsen, K.G.: A Tutorial on Uppaal. Sfm-rt 2004, pp. 200–236. Springer, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30080-9_7

[9] Broering, E.F., Becker, L.B.: Applying runtime verification in real-time systems with freertos. In: 2022 XII Brazilian Symposium on Computing

Systems Engineering (SBESC), pp. 1–6 (2022). https://doi.org/10.1109/SBESC56799.2022.9964952

[10] Gonçalves, F.S., Pereira, D., Tovar, E., Becker, L.B.: Formal verification of aadl models using uppaal. In: 2017 VII Brazilian Symposium on Computing Systems Engineering (SBESC), pp. 117–124 (2017). IEEE

[11] Misson, H.A., Gonçalves, F.S., Becker, L.B.: Applying integrated formal methods on cps design. In: 2019 IX Brazilian Symposium on Computing Systems Engineering (SBESC), pp. 1–8 (2019). https://doi.org/10.1109/SBESC49506.2019.9046084

[12] Passarini, R.F., Farines, J., Fernandes, J.M., Becker, L.B.: Cyber-physical systems design: transition from functional to architectural models. Des Autom Embed Syst (19), 345–366 (2015). https://doi.org/10.1007/s10617-015-9164-y

[13] Gonçalves, F.S., Donadel, R., Raffo, G.V., Becker, L.B.: Assessing the use of Simulink on the development process of an unmanned aerial vehicle. In: 3rd Workshop on Cyber-Physical Systems (CyPhy 2013) (2013)

[14] Zowghi, D., Coulin, C.: Requirements elicitation: A survey of techniques, approaches, and tools. Engineering and managing software requirements (2005)

[15] Lara, A.V., Nascimento, I.B., Arias-Garcia, J., Becker, L.B., Raffo, G.V.: Hardware-in-the-loop simulation environment for testing of tilt-rotor uav's control strategies. In: Congresso Brasileiro de Automática-CBA, vol. 1 (2019)

[16] Cordeiro, L.C., de Lima Filho, E.B., de Bessa, I.V.: Survey on automated symbolic verification and its application for synthesising cyber-physical systems. IET Cyper-Phys. Syst.: Theory & Appl. **5**(1), 1–24 (2020)

[17] Feiler, P.H., Gluch, D.P.: Model-based Engineering with AADL: an Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley, ??? (2012)

[18] Lasnier, G., Zalila, B., Pautet, L., Hugues, J.: Ocarina: An environment for aadl models analysis and automatic code generation for high integrity applications. In: Reliable Software Technologies–Ada-Europe 2009: 14th Ada-Europe International Conference, Brest, France, June 8-12, 2009. Proceedings 14, pp. 237–250 (2009). Springer

[19] Alshmrany, K.M., Aldughaim, M., Bhayat, A., Cordeiro, L.C.: Fusebmc

v4: Smart seed generation for hybrid fuzzing - (competition contribution). In: Johnsen, E.B., Wimmer, M. (eds.) 25th International Conference on Fundamental Approaches to Software Engineering - FASE 2022. LNCS, vol. 13241, pp. 336–340 (2022). https://doi.org/10.1007/978-3-030-99429-7_19

[20] Monteiro, F.R., Garcia, M., Cordeiro, L.C., de Lima Filho, E.B.: Bounded model checking of C++ programs based on the qt cross-platform framework. Softw. Test. Verification Reliab. **27**(3) (2017)

[21] Cordeiro, L.C., de Lima Filho, E.B.: Smt-based context-bounded model checking for embedded systems: Challenges and future trends. ACM SIGSOFT Softw. Eng. Notes **41**(3), 1–6 (2016)

[22] Alhawi, O.M., Rocha, H., Gadelha, M.R., Cordeiro, L.C., de Lima Filho, E.B.: Verification and refutation of C programs based on k-induction and invariant inference. Int. J. Softw. Tools Technol. Transf. **23**(2), 115–135 (2021)

[23] Silva, T., Porto, C., da S. Alves, E.H., Cordeiro, L.C., Rocha, H.: Verifying security vulnerabilities in large software systems using multi-core k-induction. CoRR **abs/2102.02368** (2021)

[24] Gadelha, M.Y.R., Ismail, H.I., Cordeiro, L.C.: Handling loops in bounded model checking of C programs via k-induction. Int. J. Softw. Tools Technol. Transf. **19**(1), 97–114 (2017)

[25] Behrmann, G., David, A., Larsen, K.G.: A tutorial on uppaal 4.0 (2006). URL http://www. it. uu. se/research/group/darts/papers/texts/new-tutorial. pdf (2014)

[26] Clarke, E.M.: The birth of model checking. 25 Years of model checking: history, achievements, perspectives, 1–26 (2008)

[27] Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT press, ??? (2008)

[28] Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Workshop on Logic of Programs, pp. 52–71 (1981). Springer

[29] Donadel, R., Raffo, G.V., Becker, L.B.: Modeling and control of a tiltrotor uav for path tracking. IFAC Proceedings Volumes **47**(3), 3839–3844 (2014). https://doi.org/10.3182/20140824-6-ZA-1003.01735. 19th IFAC World Congress

[30] Singhoff, F., Legrand, J., Marcé, L.N.L.: Cheddar : a flexible real time

scheduling framework. ACM SIGAda Ada Letters **24**(4), 1–8 (2004)

[31] STMicroelectronics: STMicroelectronics STM32F4DISCOVERY Discovery Kit with STM32F407VG MCU Data Brief STmicroelectronics. http://www.st.com/resource/en/data_brief/stm32f4discovery.pdf

[32] ARM: ARM Cortex-M4 Processor. https://www.arm.com/products/silicon-ip-cpu/cortex-m/cortex-m4

[33] Barry, R.: Mastering the freertos real time kernel. A Hands-On Tutorial Guide Real Time Engineers Ltd (2016)

[34] Donadel, R.: Modeling and control of a tiltrotor unmanned aerial vehicle for path tracking. Master's thesis, Federal University of Santa Catarina (2015)

[35] Hooker, J.N.: Solving the incremental satisfiability problem. The Journal of Logic Programming **15**(1), 177–186 (1993)

[36] da S. Alves, E.H., Cordeiro, L.C., de Lima Filho, E.B.: A method to localize faults in concurrent C programs. J. Syst. Softw. **132**, 336–352 (2017)

[37] Silberschatz, A., Galvin, P.B., Gagne, G.: Applied Operating System Concepts. John Wiley & Sons, Inc., New York, U.S. (1999)

[38] Goncalves, F.S., Misson, H.A., Cordeiro, L., Becker, L.B.: Supplementary Material of the Paper "Applying Formal Methods to Build a Safe Continuous-Control Architecture for Cyber-Physical Systems". https://provant.paginas.ufsc.br/formal-verification/

[39] Ferrando, A., Dennis, L.A., Cardoso, R.C., Fisher, M., Ancona, D., Mascardi, V.: Toward a holistic approach to verification and validation of autonomous cognitive systems. ACM Trans. Softw. Eng. Methodol. **30**(4) (2021). https://doi.org/10.1145/3447246

[40] Xu, H., Wang, P.: Real-time reliability verification for uav flight control system supporting airworthiness certification. PloS one **11**(12), 0167168 (2016)

[41] Sirigineedi, G., Tsourdos, A., White, B.A., Zbikowski, R.: Modelling and verification of multiple uav mission using smv. arXiv preprint arXiv:1003.0381 (2010)

[42] Humphrey, L.R.: Model checking for verification in uav cooperative control applications. Recent Advances in Research on Unmanned Aerial Vehicles, 69–117 (2013)

[43] Schumann, J., Moosbrugger, P., Rozier, K.Y.: R2u2: monitoring and diagnosis of security threats for unmanned aerial systems. In: Runtime Verification: 6th International Conference, RV 2015, Vienna, Austria, September 22-25, 2015. Proceedings, pp. 233–249 (2015). Springer

[44] Harrison, J.: Handbook of Practical Logic and Automated Reasoning. Cambridge University Press, Cambridge, U.K. (2009)

[45] Chaves, L.C., et al.: Formal verification applied to attitude control software of unmanned aerial vehicles (2018)

[46] Ismail, H., Bessa, I., Cordeiro, L.C., de Lima Filho, E.B., Filho, J.E.C.: Dsverifier: A bounded model checking tool for digital systems. In: 22nd International Symposium on Model Checking Software (SPIN). LNCS, vol. 9232, pp. 126–131 (2015)

[47] Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: Nusmv 2: An opensource tool for symbolic model checking. In: 14th International Conference Computer Aided Verification (CAV). LNCS, vol. 2404, pp. 359–364 (2002)

[48] Holzmann, G.J.: Parallelizing the spin model checker. In: 19th International Workshop Model Checking Software (SPIN). LNCS, vol. 7385, pp. 155–171 (2012)

[49] Alur, R., Henzinger, T.A.: Logics and models of real time: A survey. In: Proceedings of the Real-Time: Theory in Practice, REX Workshop, pp. 74–106. Springer, Berlin, Heidelberg (1991)

[50] Matos Pedro, A., Pinto, J.S., Pereira, D., Pinho, L.M.: Runtime verification of autopilot systems using a fragment of mtl-$\int$. International Journal on Software Tools for Technology Transfer (STTT) **20**(4), 379–395 (2018)

[51] Lakhneche, Y., Hooman, J.: Metric temporal logic with durations. Theor. Comput. Sci. **138**(1), 169–199 (1995)

[52] Foundation, D.: The Pixhawk Open Source Flight Controller. https://pixhawk.org/

[53] FreeRTOS.org: FreeRTOS: Real-time Operating System for Microcontrollers. https://freertos.org/

Statements and Declarations

Funding

This work was supported by the Brazilian funding agencies CNPq, CAPES, FAPESC and the UK's Royal Academy of Engineering through its Chair in Emerging Technologies scheme.

Competing Interests

The authors have no relevant financial or non-financial interests to disclose.