Specifying Monitors for Autonomous Cyber-Physical Systems

A dissertation submitted towards the degree

Doctor of Engineering (Dr.-Ing.)

of the Faculty of Mathematics and Computer Science

of Saarland University

by

Sebastian Schirmer

Saarbrücken, 2025

Abstract

In this thesis, we investigate and apply specification-based monitoring for autonomous cyber-physical systems, such as unmanned aerial vehicles (UAV). The aim is to support development and ensure safe and correct operation.

In Part I, we show how aviation safety documents map to monitoring and how system behaviors are formalized. In particular, we propose temporal behavior trees (TBT), which build upon the widely used Behavior Tree (BT) framework for robotic task execution by combining it with temporal languages. TBTs provide a modular structure for decomposing complex tasks and enable retrofitting monitoring into applications that use BT.

In Part II, we present offline monitoring algorithms that analyze system log files post-execution. First, we introduce trace segmentation that splits the trace into segments and assigns them portions of the specification. This helps to understand which parts of the specification are violated and require further development. Second, we propose trace repair that minimally modifies a trace that violates its specification so that it satisfies it. Our experiments include an autonomous landing of a UAV on a ship and demonstrate their practical use.

In Part III, we present tools for online monitoring that ease the integration of specified monitors and validate these monitors in real-world flight tests. The results confirm the effectiveness of our specified monitors in safeguarding both machine learning components and UAV operations.

Zusammenfassung

In dieser Arbeit untersuchen und wenden wir spezifikationsbasiertes Monitoring für autonome cyber-physische Systeme wie unbemannte Luftfahrzeuge (UAVs) an, um die Entwicklung zu unterstützen und einen sicheren Betrieb zu gewährleisten.

In Teil I zeigen wir, wie Luftfahrtdokumenten auf Monitoring abgebildet werden und wie Systemverhalten formalisiert werden. Wir führen Temporale Behavior Trees (TBT) ein, die auf dem verbreiteten Behavior-Tree-Framework (BT) für robotische Aufgabensteuerung aufbauen und es um temporale Sprachen erweitern. TBTs bieten eine modulare Struktur zur Zerlegung komplexer Aufgaben und ermöglichen die nachträgliche Integration von Monitoring in BT-Anwendungen.

In Teil II stellen wir Offline-Monitoring-Algorithmen vor, die Logdateien analysieren. Zunächst führen wir Trace-Segmentierung ein, die die Ausführung in Segmente teilt und diesen Teile der Spezifikation zuweist. Dies hilft zu verstehen welche Teile weiteren Entwicklungsbedarf haben. Anschließend stellen wir Trace-Reparatur vor, die eine verletzende Ausführung minimal modifiziert, sodass sie die Spezifikation erfüllt. Experimente, darunter die autonome Landung eines UAV auf einem Schiff, zeigen die praktische Anwendbarkeit.

In Teil III zeigen wir Werkzeuge für das Online-Monitoring, die die Integration spezifizierter Monitore vereinfachen und validieren diese in Flugtests. Die Ergebnisse bestätigen die Wirksamkeit beim Schutz vor maschineller Lernkomponenten und des UAV-Betriebs.

Acknowledgements

This thesis was written during my work as a research scientist at the Institute of Flight Systems in the Unmanned Aircraft Department of the German Aerospace Center (DLR) in Braunschweig.

I would like to express my sincere gratitude to my supervisor, Bernd Finkbeiner, for his guidance and support throughout my research and beyond. I am especially thankful for the opportunity to attend the RV Conference as a student in 2016, which motivated me to begin my research journey at DLR. I am also deeply grateful that you accepted me as your PhD student, even though I am based in Braunschweig.

I want to thank my colleagues at DLR. Especially, I am grateful to Johann C. Dauer for providing me with the opportunity to stay abroad, which allowed me to gain valuable experience and insights that greatly contributed to my research. I would also like to thank Christoph Torens, who shares many of my perspectives and has supported me since I joined DLR. These are just two examples, but I am thankful to all the members of the department of Unmanned Aircraft: their dedication and expertise in flight testing and research have greatly enriched my experience over the years!

Next, I want to thank all my co-authors. In particular, I am grateful to the members and previous members of the Reactive Systems Group: Jan Baumeister, Peter Faymonville, Jesko Hecking-Harbusch, Florian Kohn, Noemi Passing, Frederik Scheerer, Maximilian Schwenger, and Hazem Torfah. Visiting you in Saarbrücken and hosting you for flight testing was always fun. I would also like to thank Sriram Sankaranarayanan for welcoming me to join his group at CU Boulder during my stay abroad. I greatly appreciated the time spent there and the discussions with you and your group.

I want to thank my examination board for their time and feedback. Thank you Bernd Finkbeiner, Martina Maggio, and Sriram Sankaranarayanan.

I am very grateful for the support of my family and close friends. Last but not least, i would like to thank my wife Kathleen for her love, patience with me, and for the little miracle she brought into our lives – Niklas.

Eidesstattliche Versicherung
Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Declaration of original authorship
I hereby declare that this dissertation is my own original work except where otherwise indicated. All data or concepts drawn directly or indirectly from other sources have been correctly acknowledged. This dissertation has not been submitted in its present or similar form to any other academic institution either in Germany or abroad for the award of any other degree.

Saarbrücken, November, 2025

Sebastian Schirmer

Contents

Chapter 1

Introduction

Imagine a drone delivered your eagerly awaited online order right to your doorstep within a few minutes, skipping traffic. While such drones are still being developed and tested for everyday deliveries, they are already saving lives by transporting blood and medical products nowadays [1]. At the core of these drones are computer systems that are reactive and embedded. A *reactive system* is a system that continuously interacts with its environment [2]. The interaction is typically event-driven and time-sensitive, i.e., a corresponding response to an event must be provided within a limited amount of time. An *embedded system* consists of specialized hardware and software integrated into a larger system with the aim of fulfilling a dedicated purpose [3]. Such systems often operate under limited resources, including computational power, memory capacity, and low energy availability. A *cyber-physical system* (CPS) generalizes embedded systems by putting an emphasis on the interaction with the physical world via sensors and actuators in a feedback loop [4]. The term cyber-physical system was coined by Helen Gill at the National Science Foundation, following the realization within the community that the core challenge of embedded systems was no longer limited resources but rather their interaction with the physical world [3].

As an illustrative example of a CPS, consider an unmanned aerial vehicle (UAV), i.e., a drone. Typically, a UAV is equipped with multiple sensors to perceive information about its surroundings. Common on-board sensors include a GPS receiver for positioning and an inertial measuring unit to estimate angular rates and orientation. The gathered data are processed by algorithms for planning and navigation. Given the current position, graph search algorithms such as A* [5] are used to plan a trajectory that guides the UAV to its target location. This trajectory is then followed by a control algorithm such as a PID controller [6], which generates actuator commands to stay on the planned trajectory. A major strength of such traditional approaches is that formal guarantees can be provided as their fundamental algorithms are explicitly defined. For example, control theory provides rigorous proofs of important properties of a controller, such as controllability, reachability, and stability [7].

While traditional algorithms are effective at following predefined flight routes, they lack sufficient semantic understanding of the environment to identify safe delivery spots such as your doorstep or to see and categorize objects to avoid collisions effectively. In recent years, data-driven methods have gained significant momentum to achieve this task [8]. In particular, machine learning approaches based on camera images have surpassed traditional algorithms in perceiving and interpreting their surroundings [9]. Further, reinforcement learning has demonstrated remarkable success by defeating world champions in Go [10, 11] and reaching champion-level performance in drone racing by learning efficient control policies [12]. When integrated into a CPS, such components elevate the system's capabilities beyond the pre-programmed execution typical of automation, enabling an *autonomous cyber-physical system*.

However, machine learning approaches typically offer few, if any, formal guarantees [13]. This limitation adds to the existing challenges of CPS, which already involve time-sensitive computations, limited resources, and close interaction with the physical world. Also, while a robot tasked with tidying a room might occasionally be permitted to damage an item, a UAV delivering cargo cannot tolerate failures. These missions are inherently *safety-critical* as human lives may be at risk.

To address these challenges, this thesis investigates the use of runtime verification for autonomous CPS. →Part I, p. 15 In Part I, we provide background on aviation regulations and standard documents that yield a safe autonomous CPS when followed and link them to monitoring. We also present approaches to specifying monitors, ranging from monitoring sensors to CPS behaviors. In particular, we extend behavior trees, which are a prominent way to program and execute complex robotic behaviors, into a monitoring formalism called *temporal behavior tree*. This allows us to easily retrofit monitoring to existing systems with minimal effort. →Part II, p. 51 In Part II, we propose novel offline monitoring techniques to analyze autonomous CPS post-execution and help improving these systems. This is important as increasing task complexity makes debugging and error correction more difficult. →Part III, p. 89 In Part III, we introduce tools that ease the integration of monitors into CPS, providing monitoring verdicts during execution. This integration addresses the challenges of CPS and flight test results demonstrate the effectiveness of our monitors in safeguarding both machine learning components and UAV operations.

The remainder of this introduction is as follows. In Section 1.1, we give an overview of runtime verification. Then, Section 1.2 states challenges. Next, Section 1.3 lists the publications which this thesis is based on and Section 1.4 outlines the main contributions. Finally, Section 1.5 provides an overview of related work.

1.1. Runtime Verification

There is a spectrum of approaches for showing that a system functions as intended [14]. At one end is testing, which is cost-effective but provides no formal guarantees. At the other end is formal verification, which requires specifying properties φ in a formal language and proves that a model M, a safe abstraction of the system under consideration, satisfies these properties for all runs of the model, i.e., $\forall \sigma \in \mathit{runs}(M) : \sigma \models \varphi$. While formal verification is often costly, it offers strong guarantees. Unfortunately, neither of these extremes is ideal for autonomous CPS. Testing only examines behaviors of preselected scenarios, and achieving a full coverage of all scenarios is infeasible. Formal verification suffers under scalability issues, especially for CPS where continuous dynamics of the system and the physical world are involved.

Runtime verification is a formal method and lies between these extremes [15, 16, 17]. Like formal verification, runtime verification relies on properties expressed in a formal specification language. However, instead of proofing correctness for all runs, the specification is used to generate a monitor that runs alongside the CPS to observe its behavior. For each such concrete CPS execution *trace* σ, the monitor checks if $\sigma \models \varphi$. Unlike testing, runtime verification ensures that specifications are checked against actual executions of the system not only during development or early deployment. It is considered as integral part of the operational system to assess system health, used to activate counter-measures when needed. One could argue that every system should include monitoring, since no system should be considered 100% safe. The monitor that checks the compliance is automatically generated and its correctness is guaranteed by construction as the translation from specification to executable code is formally proven. Furthermore, monitors can produce both qualitative and quantitative verdicts. A qualitative verdict indicates if the property holds, whereas a quantitative verdict provides information about how much the properties is satisfied or violated and may include statistical information. In this thesis, we use the terms runtime monitoring and runtime verification interchangeably.

Figure 1.1 illustrates the integration of a monitor into a CPS. The sensors, compute units, and actuators of a CPS are instrumented to forward events to and receive verdicts from a monitor synthesized from a formal specification. Instrumentation should select the smallest set of signals required by the specification while minimizing overhead and changes to the CPS. Instrumentation may be hardware-based or software-based [18]. Hardware approaches use additional physical lines that mirror analog/mixed signals to the monitor. Software approaches add code in firmware, drivers, or middleware to publish events via existing links, e.g., buses such as CAN or I^2C; serial interfaces such as UART; or networks such as Ethernet. Ideally, the required signals already traverse a shared bus, allowing the monitor to passively observe them without additional instrumentation. The monitor itself may execute *on-system* or *off-system*. Running on-system gives direct access to internal data, such as CPU load or available memory, ensures low latency communication, and provides a straightforward data synchronization due to a shared clock. In contrast, running the monitor off-system improves non-instrusiveness, i.e., failures of the system or the monitor are isolated and do not affect each other. However, this decoupling comes at the cost of reduced observability and greater synchronization efforts. Note that the figure intentionally abstracts from where events are

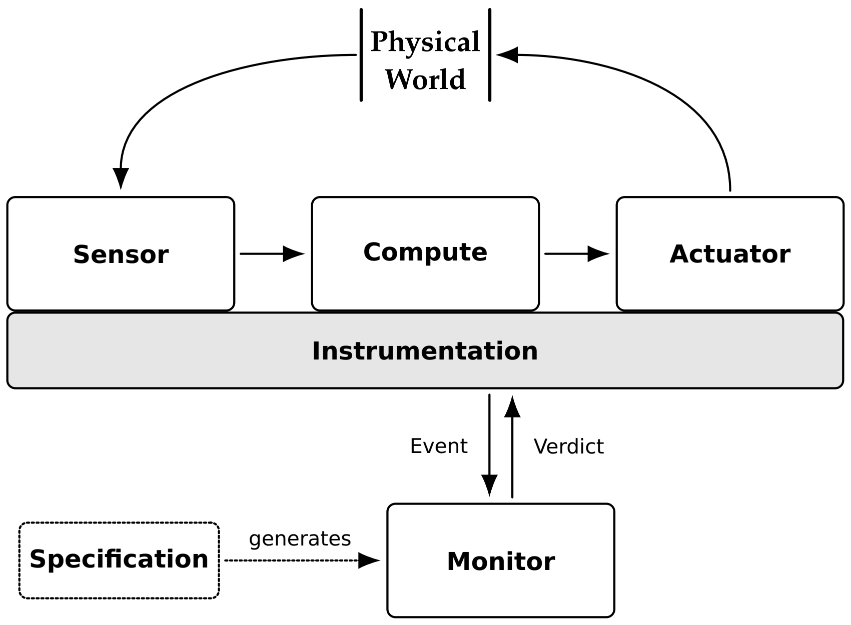


Figure 1.1.: Events from the instrumented CPS are checked by a monitor synthesized from a specification. Verdicts of the monitor are sent back to the CPS to response to violations.

extracted or verdicts are used. For example, relevant sensor data are often processed by a computing unit, so it may be preferable for monitoring to receive the processed data rather than the raw data. Likewise, a monitor verdict may either command an actuator directly, e.g., to a predefined position, or be passed to a compute unit that selects the appropriate actuator command.

Runtime verification has a wide range of applications. It ranges from safety-critical domains, such as aerospace [19, 20, 21], automotive [22, 23, 24], and railways [25] to emerging areas such as robotics [26, 27, 28], healthcare [29, 30, 31], and internet of things [32, 33, 34]. It is also not limited to safety, but it has also been used for networks to ensure security policies [35, 36] and for algorithmic fairness [37, 38]. Monitors have been deployed both during development and operation [15]. During development, they are integrated into software-in-the-loop and hardware-in-the-loop simulation, as well as log file analysis post execution, to provide early feedback and generate verification artifacts. During operation, runtime verification is employed as a safety net, continuously checking that the execution adheres to its specification and triggering countermeasures when necessary. In general, we distinguish two concepts: *online* and *offline* monitoring [39]. In *offline monitoring*, the system execution has finished and the execution trace is fully available to the monitor. A monitor algorithm may perform arbitrary, multi-pass analyses where the trace can be traversed forward or backward with jumps to random positions of the trace. Further, since no time-critical decision is pending and monitors must not be deployed on-board, resource constraints are typically not the limiting factor. By contrast, *online monitoring* observes an execution with new events continuously arriving. A monitor algorithm processes them in arrival order under finite-prefix semantics, i.e., final verdicts must be delayed until sufficient evidence appear at later positions in the trace. Because the monitor is deployed within a CPS, it must operate under strict resource limitations and timing constraints. Ideally, time and memory guarantees can be given.

1.2. Challenges

This thesis addresses several open challenges when using runtime verification for autonomous CPS.

The first challenge is *how to think about complex systems when writing and managing specifications*. Typically, monitoring is applied locally to individual parts of the system rather than the entire system, which makes it difficult to obtain a global, end-to-end understanding of the system's status. Nevertheless, ensuring that monitoring spans the full system stack from low-level sensors to high-level operations behaviors is crucial for reasoning about and ensuring the safety of operating an autonomous CPS. A representative statement that acknowledges this challenge is [40] where a holistic assurance case is presented. The authors state that "Aviation safety is based on the notion of risks and hazards, however rare, will surface, and aviation systems should be designed to be fail-safe. [...] However, innovations that introduce many unknown risks require additional mitigations beyond those incorporated in their design. [...] To manage safety in aviation, it is imperative that each component be understood, and its safety behavior be measured, monitored, and corrected in a predictive manner." [Underlining added]. Design-time assurance alone is indeed insufficient for autonomous CPS and it is also necessary to understand and to monitor not only individual components of the system stack but also the larger systems that emerge from combining components. This thesis contributes by proposing a hierarchy of monitoring properties for autonomous systems that introduces levels that support the integration of a holistic runtime monitoring.

The second challenge is *how to define specifications for online monitoring autonomous CPS that act as "sandbox"*. In particular, the challenge is to formulate properties that constraint the system's behavior such that it can operate safely, while still allowing the flexibility to react autonomously. This challenge dates back to the well-known Simplex architecture [41], which was originally proposed to safely update a controller during operation. In recent years, the aviation community has revisited this architecture in the advent of integrating untrustworthy yet highly capable components such as machine-learning based components into the avionics stack. Most notably is the published ASTM standard F3269-21 (Standard Practice for Methods to Safely Bound Behavior of Aircraft Systems Containing Complex Functions Using Run-Time Assurance), discussed in [42]. In this standard, safety monitors play the central role for safety by identifying unsafe behaviors and switching to safe recovery functions if necessary. This thesis provides empirical evidence that runtime verification matches the requirements for implementing such safety monitors and further strengthens design-time reasoning about these monitors.

The third challenge is *how can verdicts produced by offline monitoring be processed and presented to developers to yield insights that help to refine and improve the system*. Typically in software and systems engineering, testing is a widely used approach for understanding and improving a system based on requirements captured in test cases. Sometimes, test cases are even defined before any system code is written, a practice known as test-driven development [43]. Further, when such tests describe the system's expected behavior using a domain-specific language, the approach is commonly referred to as behavior-driven development [44]. Expected behaviors often involve temporally related events, making them inherently complex to capture as a test, e.g., "execute A only after B" or "wait for five seconds, before executing C". In this thesis, we use a specification language to concisely express such complex behaviors. Further, we will not only check if a trace satisfies or violates the specification, but, more importantly, we will identify precisely which parts of the behavior are responsible for satisfaction or violation and how any violations can be repaired. Therefore, we go beyond the capabilities of traditional testing.

1.3. Publications and Collaborations

This thesis presents work in collaboration with many people. The contributions listed in Section 1.4 are the author's work. This thesis is based on publications in the fields of formal methods and aviation:

[45] Sebastian Schirmer, Philipp Schitz, Johann C. Dauer, Bernd Finkbeiner, Sriram Sankaranarayanan. **Trace Repair for Temporal Behavior Trees**. In Proceedings of the 32nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2026).

[46] Jan Baumeister, Bernd Finkbeiner, Franz Jünger, Florian Kohn, Sebastian Schirmer, Christoph Torens. **A ROS Adapter for RTLola**. In Proceedings of the 25th International Conference on Runtime Verification (RV 2025).

[47] Christoph Torens, Franz Jünger, Sebastian Schirmer, Pranav Nagarajan, Simon Schopferer, Dmytro Zhukov, and Johann Dauer. **Runtime Monitoring of Operational Design Domain to Safeguard Machine Learning Components**. In the Journal of the Council of European Aerospace Societies (CEAS 2025).

[48] Jan Baumeister, Bernd Finkbeiner, Florian Kohn, Florian Löhr, Guido Manfredi, Sebastian Schirmer, and Christoph Torens. **Monitoring Unmanned Aircraft: Specification, Integration, and Lessons-Learned**. In Proceedings of the 36th International Conference on Computer Aided Verification (CAV 2024).

[49] Sebastian Schirmer, Jasdeep Singh, Emily Jensen, Johann C. Dauer, Bernd Finkbeiner, and Sriram Sankaranarayanan. **Temporal Behavior Trees: Robustness and Segmentation**. In Proceedings of the 27th ACM International Conference on Hybrid Systems: Computation and Control (HSCC 2024).

[50] Jan Baumeister, Johann C. Dauer, Bernd Finkbeiner, and Sebastian Schirmer. **Monitoring With Verified Guarantees**. In International Journal on Software Tools for Technology Transfer (STTT 2023).

[51] Sebastian Schirmer, Christoph Torens, Johann C. Dauer, Jan Baumeister, Bernd Finkbeiner, and Kristin Y. Rozier. **A Hierarchy of Monitoring Properties for Autonomous Systems**. At American Institute of Aeronautics and Astronautics SciTech Forum and Exposition (SciTech 2023).

[52] Sebastian Schirmer and Christoph Torens. **Safe Operation Monitoring for Specific Category Unmanned Aircraft**. In the book "Automated Low-Altitude Air Delivery: Towards Autonomous Cargo Transportation with Drones", part of the "Research Topics in Aerospace" book series (RTA 2022).

[21] Jan Baumeister, Bernd Finkbeiner, Sebastian Schirmer, Maximilian Schwenger, and Christoph Torens. **RTLola Cleared for Take-Off: Monitoring Autonomous Aircraft**. In Proceedings of the 32nd International Conference on Computer Aided Verification (CAV 2020).

The joint work presented in [21, 46, 48, 50] is connected through a common project.

The work in [21] predates the project but was an initial starting point. It reports a flight test in which an RTLola specification was executed on an FPGA flying onboard DLR's superARTIS. The author's work are the development of the RTLola geofencing specification and the hardware integration of the FPGA. The flight test validates the geofencing specification and illustrates the motivation for using an FPGA for DLR's superARTIS. The software integration, the RTLola-to-VHDL transcompilation, the specifications other than geofencing, and the VHDL synthesis evaluation were not carried out by the author. This work contributed to a master's thesis [53] co-advised by the author and two PhD theses [54, 55].

The work in [48] was jointly conducted with the project partners and summarizes the steps accomplished during the project, i.e., it reports the challenges when using runtime verification for industrial products. The author's work that ended up in this thesis is the extension of the RTLola geofencing specification by incorporating the aircraft state to compute the remaining time until a potential geofence breach. The abstract integration, concrete integration, and the RTLola specification other than geofencing are not the author's work.

The work in [46] presents an experimental flight test that safeguards an integrated machine-learning based component onboard the DLR's unmanned aircraft. The author's work are the development of the RTLola adapter, which automates the integration of RTLola monitors into ROS systems, and the development of the RTLola specification used during the experimental flight test. The development within the RTLola framework is not the author's work.

The work in [50] highlights the importance of functional correctness of specifications and is the author's work.

1.4. Contributions

The contributions of this thesis lie at the intersection of formal methods and cyber-physical systems, with a particular focus on unmanned aircraft. They address the challenges stated in the previous section. Therefore, the thesis mixes chapters focused on CPS and others centered on formal methods. The formal methods chapters provide the formal foundation, while CPS chapters demonstrate the practical application of formal methods to CPS. Each chapter visually indicates its focus by including an icon in the title: for CPS and for formal methods.

Chapter 2: A Hierarchy of Monitoring Properties for Autonomous Systems
This chapter addresses the first challenge and is based on the work presented in [51]. The contributions can be summarized as:

- Mapped requirements and architectures from aviation regulations and standards to monitoring
- Introduction of a hierarchy of monitoring properties that provides a holistic perspective on monitoring autonomous CPS, i.e., from sensor properties up to mission properties
- Presentation of example properties relevant to aviation for each level of the hierarchy, illustrating different specification languages that help bridge the gap between runtime verification and aviation
- Critical discussion of monitoring specification languages in practice, highlighting concerns and potential when bridging the two domains

Chapter 3: Specification Languages
This chapter addresses the second challenge and is based on the work presented in [45, 49]. The contributions can be summarized as:

- Introduction of Temporal Behavior Trees (TBTs) as a specification language that combines behavior trees with common temporal specification languages
- Formal definition of Boolean and Robust semantics for TBTs

- Proof that TBTs are more expressive than finite-trace Signal Temporal Logic

Chapter 4: Trace Segmentation
This chapter addresses the third challenge and is based on the work presented in [49]. The contributions can be summarized as:

- Formulation of the segmentation problem, which assigns segments of a trace to corresponding nodes in the TBT
- Dynamic programming algorithm to compute TBT robustness
- Definition and computation of (alternative) segmentations
- Introduction of trace subsampling to approximate robustness of a given trace w.r.t. its TBT specification
- Use of lazy evaluation to approximate robustness of a given trace w.r.t. its TBT specification
- Empirical evaluation of segmentation algorithms through the analysis of human behavior and autonomous ship deck landing of a UAV

Chapter 5: Trace Repair
This chapter addresses the third challenge and is based on the work presented in [45]. The contributions can be summarized as:

- Formulation of the trace repair problem for CPS, where a violating trace w.r.t. its TBT specification is minimally modified to achieve satisfaction
- Optimal mixed-integer linear programming formulation for trace repair, including the encoding of TBTs and different cost functions
- Introduction of the incremental repair strategy and the landmark-based repair strategy, that both mitigate the computational limitations of the optimal formulation
- Empirical evaluation of the trace repair algorithms using a robot search task and an autonomous ship deck landing of a UAV

Chapter 6: Safeguarding Machine-learning
This chapter addresses the second challenge and is based on the work presented in [46]. The contributions can be summarized as:

- Development of a ROS adapter for RTLola, simplifying integration of RTLola monitors into ROS applications

- RTLola specification for cross-validating untrustworthy machine-learning-based object detection components against a traditional trustworthy LiDAR-based blob detector, thereby reducing false positives
- Flight test validation demonstrating how such a cross-validation improves safe landing of an unmanned aircraft

Chapter 7: Safe Operation Monitoring
This chapter addresses the second challenge and is based on the work presented in [21, 47, 48, 52]. The contributions can be summarized as:

- Design of ALAADy's Run-Time Assurance architecture to support SORA compliance
- Development of FencyCreator, a tool that generates RTLola specifications for the safe operation of the ALAADy
- RTLola specifications for geofencing, flight path monitoring, and progress monitoring necessary for a safe operation
- Simulation-based evaluation of flight termination triggered by a geofencing monitor, applied to a gyrocopter, a box-wing aircraft, and a twin-boom aircraft
- Flight test validations demonstrating the specifications in practice and showing how such a monitor supports safe operation by improving situational awareness

Chapter 8: Monitoring with Verified Guarantees
This chapter addresses the second challenge and is based on the work presented in [50]. The contributions can be summarized as:

- Introduction to an extension of Lola annotated with Hoare triples
- SMT encoding of Lola with assumptions and assertions
- Development of an offline verification algorithm for proving the functional correctness of an annotated Lola specification
- Implementation of this algorithm in the tool SpecVerifier
- Applying SpecVerifier to existing avionics specifications, uncovering previously unknown critical bugs
- Development of an online monitoring algorithm that exploits assumptions by only monitoring assertions when related assumptions are violated
- Experimental evaluation of the runtime performance of the online monitoring algorithm

1.5. Related Work

We provide an overview of related work, focusing on the key contributions.

Temporal Behavior Trees Behavior trees (BT) first were invented to enable modular AI in computer games [56], but have become increasingly popular in robotics [57, 58, 59, 60, 61, 62] and beyond [63, 64, 65, 66]. There are several properties that make them attractive including composability, reactivity, and human-readability. The purpose of behavior trees is to specify complex behaviors and execute them. Temporal behavior trees are complementary and can be retrofitted to BT. Their purpose is to specify successful or failed behavior properties that can be used to segment or monitor a trace.

TBTs have semantics over finite traces. Classically, however, temporal logics have their semantics defined over infinite traces. When it comes to finite traces, the main concern is how to handle *open* obligations at the end of the trace. One approach is to provide a multi-valued semantics such as the one based on so-called *good*, *bad*, and *ugly* prefixes by Bauer et al. [67]. Our approach is more closely related to the semantics provided by De Giacomo and Vardi [68]. In fact, the regular language operators presented in this work are identical to some of the operators used in TBT. We additionally consider robust semantics.

The notion of robustness of a trace with respect to temporal logic specification for *infinite length* trace has its origins independently in the work of Fainekos and Pappas; Donze and Maler; and Rizk et al. [69, 70, 71]. The problem of defining and computing robustness of finite-length traces has been studied as well. Deshmukh et al. [72] provide a robust interval semantics that maps a finite length trace (seen as a prefix of an infinite trace) and an STL formula to an interval over robustness (l, v) such that for any suffix of the trace, l is the greatest lower bound and v is the lowest upper bound in respect to its robustness value. Our robust semantics of TBT is real-valued, but works over finite traces and introduces robustness for operators that make TBT more expressive when compared to standard temporal logics. The sequence operator for TBTs is similar to the ";" operator in regular linear temporal logic [73] and the "chop operator" in interval temporal logic (ITL) [74]. These ideas go back to the work of Halpern, Manna, and Moszkowski [75] and later further developed by many others, including Harel & Peleg [76] and Rosner & Pnueli [77]. The tool Tempura [1] by Moszkowski and others is an interpreter for executable ITL and can be used similar to a programming language whereas we retrofit temporal logics to executable behaviors trees [78]. Therefore, we go in the reverse direction: we turn a framework used for specifying plans into a temporal logic for specifying properties.

Segmentation Trace segmentation is closely related to timed pattern matching [79]. However, while timed pattern matching identifies all segments that *satisfy* the pattern

[1] `http://www.antonio-cau.co.uk/ITL/itlhomepagesu14.html`

criteria, segmentation finds the optimal trace assignment, which may encompass segments that *violate* its specification. The pattern matching algorithm, as outlined in [79], uses a bottom-up zone construction. In contrast, our segmentation algorithm utilizes dynamic programming in a top-down fashion.

Trace Repair Trace repair was first introduced as correction for regular languages by Wagner [80], and was later referred to as regular repair of specification [81]. Both works investigate the repair in terms of sequences of a finite alphabet, i.e., words. They showed that the problem is decidable using distance automata, but becomes PSPACE when the target language is specified in LTL. To the best of our knowledge, we are the first to extend the notion of repair to traces of CPS, where traces consist of sequences of numerical events, including sensor readings and actuator commands, instead of words. Because CPS traces involve numerical values and must respect the system's dynamics, simple symbolic edits such as flipping a single symbol are insufficient. This makes automata-based approaches not applicable. Instead, we formulate the repair problem as mixed-integer linear program (MILP), which naturally handles numerical constraints.

STL is widely used as a specification language for controlling system behaviors. In [82], STL formulas are transformed into a MILP for model-predictive control. However, the NP-HARD complexity of MILP limits this approach, especially with nested formulas or longer trajectories, making it unsuitable for long-horizon trajectory planning. [83] addresses this by structurally decomposing STL formulas to then incrementally solving them. However, their decomposition does not handle disjunctions, whereas our repair does.

Landmarks have been studied for strategy solving [84] and planning [85] as key features that must be true on any solution. In this work, we adopt a similar conceptual role but with a distinct technical use. Here, a landmark serves as a sufficient feature that guarantees the satisfaction of a TBT specification. This enables a more efficient linear program encoding, significantly extending the capability to handle longer traces.

Falsification [86] tries to find traces that "falsify" a given specification by using stochastic optimization to minimize its robustness. Trace synthesis [87], on the other hand, generates traces that satisfy the specification. In contrast, this work addresses the problem of trace repair: given a violating trace, we minimally modify it so that the resulting trace satisfies the specification. Our approach avoids stochastic optimization and uses a MILP formulation that ensures specification satisfaction while minimizing changes to the violating trace.

Runtime Verification for Unmanned Aircraft NASA uses the concept of runtime monitoring across several projects. For instance, the NASA Safeguard technology [88, 89] provides an assured safety net for UAVs, implemented as an independent system component. Safeguard can trigger mitigation tactics, e.g., flight termination, when geo-limitations are violated. Its algorithms, code, and mathematical models have been formally verified. However, as trigger conditions grow more complex, verifying hand-written code becomes increasingly difficult. In contrast, our approach relies on a formal specification language, offering a higher level of abstraction for defining monitoring properties. This enables automatic code generation that is correct-by-construction, improving both flexibility and maintainability through clear and concise property specifications.

A related formal methodology is the R2U2 framework [90]. R2U2 employs mission-time linear temporal logic in combination with a Bayesian Network for monitoring and reasoning about system health. By contrast, our specification language RTLola supports numerical computations, allowing both tasks to be expressed within a single formalism. Furthermore, R2U2 can be synthesized on an FPGA [90].

Another comprehensive framework for runtime monitoring is the NASA EVAA RTA network architecture [91]. This approach utilizes a moral compass to select the best behavior when conflicting violations occur at the same time. The moral compass can be seen as an implementation of the pilot's moral. Finally, the NASA System-wide Safety Initiative uses monitoring as one component to guarantee safety of the future airspace in combination with airspace services [92].

For the EASA's specific drone category, the operation conditions being monitored are described in the Concept of Operations (ConOps) document. One particularly critical monitoring property there is geofencing, which is increasingly important with the upcoming use of unmanned aircraft. Systems undergoing the SORA safety assessment must demonstrate compliance with technical containment requirements to remain within the operational volume defined in ConOps. The general mathematical problem of determining whether a point lies inside a polygon has been extensively studied, and comparative analyses have been conducted in the geofencing context [93, 94]. The use of RTLola for geofencing has been first proposed in [95] and later applied on a FPGA [21]. Moreover, there is growing interest in runtime monitoring implemented on FPGAs [96, 97, 98].

Integration of RV into ROS Systems In recent years, the integration of RV into ROS systems has received growing attention. ROS is widely used not only for developing robotic applications but also for supporting various simulation frameworks such as Gazebo [99] or Carla [100], as well as visualization tools like PlotJuggler. Several RV frameworks have been developed to support ROS. For example, RTAMT [101] integrates monitors based on Signal Temporal Logic into ROS applications. The Ogma [102] tool runs monitors specified in languages such as CoPilot, FRET, and Lustre as ROS nodes. Similarily, [103] uses the stream-based specification language TeSSLa to integrate monitors into ROS systems. Similar to them, we introduce the specification language RTLola to ROS. However, unlike prior tools, our tool supports not only topic-based communication, i.e., publishing and subscribing, but also introduces an RTLola service interface. This service enables request-response communication, thereby reducing the need for manual handling of monitoring responses using topic subscriptions. Although ROSMonitoring 2.0 [104] supports services by wrapping monitors in Python, we chose Rust for our adapter due to its advantages in safety and performance. Moreover, we refer to our tool as an “adapter” rather than a “bridge” because it incorporates logic that automatically derives the mappings between ROS and RTLola, offering more intelligent and robust integration.

Assume and Guarantee for Lola Most work on the verification of monitors focuses on the correct transformation into a general programming language. For example, CoPilot [20] specifications can be compiled into C code with constant time and memory requirements. Similarly, there is a translation validation toolkit for RTLola monitors implemented in Rust [105], which is based on the Viper verification tool [106]. Translation validation of this type is orthogonal to the verification approach of this thesis. Instead of verifying the correctness of a transformation, our focus is to verify the specification itself. Both activities complement each other and facilitate safer future CPS.

Our verification approach is based on classic ideas of inductive program verification [107, 108], and is closely related to the techniques used in static program verifiers like KeY [109], Why3 [110], and Dafny [111]. In a verification approach like Dafny, we are interested in functional properties of procedures, specified as post-conditions that relate the values upon the termination of the procedure with those at the time of entry to the procedure, e.g., *ensure y = old(y)*. By contrast, a stream-based language like Lola allows arbitrary access to past and future stream values. This makes it necessary to *unfold* the Lola specification in order to properly relate the assumptions and assertions in time.

Most closely related to stream-based monitoring languages are synchronous programming languages like LUSTRE [112], ESTEREL [113], and SIGNAL [114]. For these languages, the compiler is typically used for verification – a program representing the negation of desired properties is compiled with the target program and a check for emptiness decides whether the properties are satisfied. Furthermore, a translation from past linear-time temporal logic to ESTEREL was proposed to simplify the specification of more complex temporal properties [115]. Other verification techniques also exist like SMT-based k-Induction for LUSTRE [116] or a term rewriting system on synced effects [117]. A key difference in our approach is that we do not rely on compilation. Our verification works at the level of an intermediate representation. Furthermore, synchronous programming languages are limited to past references, while the stream unfolding for the inductive correctness proof of the Lola specification includes both past and future temporal operators. Similar to k-Induction, our approach is sound but not complete.

Part I

Foundations

Hot Take:
Autonomous CPS require specification languages.

Chapter 2

A Hierarchy of Monitoring Properties for Autonomous Systems

This chapter provides an introduction to readers unfamiliar with CPS, offering a foundational understanding of how monitoring capabilities are developed and applied. For readers already familiar with CPS, we present a hierarchy of monitoring properties that extends existing standardization and regulatory frameworks to address the emerging challenges posed by autonomous systems.

To support readers who may not be familiar with specification languages, we include a brief survey of commonly used languages for defining monitoring properties. This survey introduces key ideas and expressive language features, providing the necessary background for discussion. We then analyze the advantages and limitations of these languages.

2.1. Aviation Regulations & Standards

Aviation is a regulated domain with a special emphasis on safety and reliability. The key regulatory bodies such as the Federal Aviation Administration (FAA) in the United Stated, the European Union Aviation Safety Agency (EASA), and the International Civil Aviation Organization (ICAO) establish regulations and endorse standards that guide industry compliance for certification.

Unmanned aircraft systems are regulated by EASA Regulation (EU) 2019/947, which introduces three categories: “open”, “specific”, and “certified”. The “open” category addresses lower-risk civil drone operations, e.g., hobbyist drones. The “specific” category covers riskier civil drone operations, where a risk assessment provides sufficient evidence that the drone operator ensures the safe operation, e.g., inspection of infrastructure. Last, the “certified” category considers high-risk civil operations and is considered similar to manned aviation, e.g., passenger transportation.

For manned aviation, the regulatory framework is well defined, with standards developed by organizations such as SAE International or the European Organization for Civil Aviation Equipment (EUROCAE), which are formally endorsed by authorities as acceptable means of compliance. A brief overview of some standards is provided in Figure 2.1. These standards cover both the development phase and the operational phase, ensuring that intended functions are properly implemented and that their safe functionality is maintained during operation. For example, Aerospace Recommended Practice (ARP) 4761 offers guidance on conducting safety assessments for airborne systems and equipment, while ARP 4754 focuses on the overall development. This development is typically divided into hardware and software, each covered by dedicated standards. RTCA DO-254 provides guidance for the development of electronic hardware, while DO-178C addresses software development. These standards introduce a hierarchy of abstraction levels used to decompose the aircraft and its functions. The three main levels are *item*, *system*, and *aircraft*. An *item* refers to a unit composed of one or more hardware and/or software elements and represents the lowest level of abstraction. A *system* consists of inter-related items working together to perform a specific function, representing a higher level of abstraction. The *aircraft* is composed of all systems and items, it forms the highest level. A critical aspect of this hierarchy is that bidirectional traceability between all abstraction levels must be established. Requirements must be traced down from aircraft level to system and item level, and upward through verification steps. This ensures that all intended functionalities are implemented, while also guaranteeing that no unintended features exist and no dead code remains. For instance, if the aircraft must support automatic landing, this may involve a system for automatic landing, and at the item-level, it may require the flight controller to receive position data and generate actuator commands for landing.

As the “certified” category of drones closely aligns with manned aviation as the risk to harm people is similar, the mentioned standards are applicable. Drones in the “specific” category could also follow these standards, but because development according to such rigorous requirements is costly, this approach may not be practical for many business cases. Instead, the “specific” category relies on a holistic risk assessment that incorporates details about the specific operation, i.e, the aircraft *and* the operation. The Specific Operation Safety Assessment (SORA) is such an assessment developed by the Joint Authority for Rulemaking on Unmanned Systems (JARUS). For example, if a tethered drone, one not developed according to standards, is used to inspect a building in a way that ensures no people or critical infrastructure would be harmed in the event of a crash, the risk assessment may determine that the operation poses a sufficiently low risk, even tough a crash could still be costly or inconvenient for the drone operator. This line of reasoning may be a key enabler for autonomy using state-of-the-art machine learning (ML) algorithms, which is an area where existing standards and regulations are still under development. EASA’s roadmap outlines a target of 2035 for the first

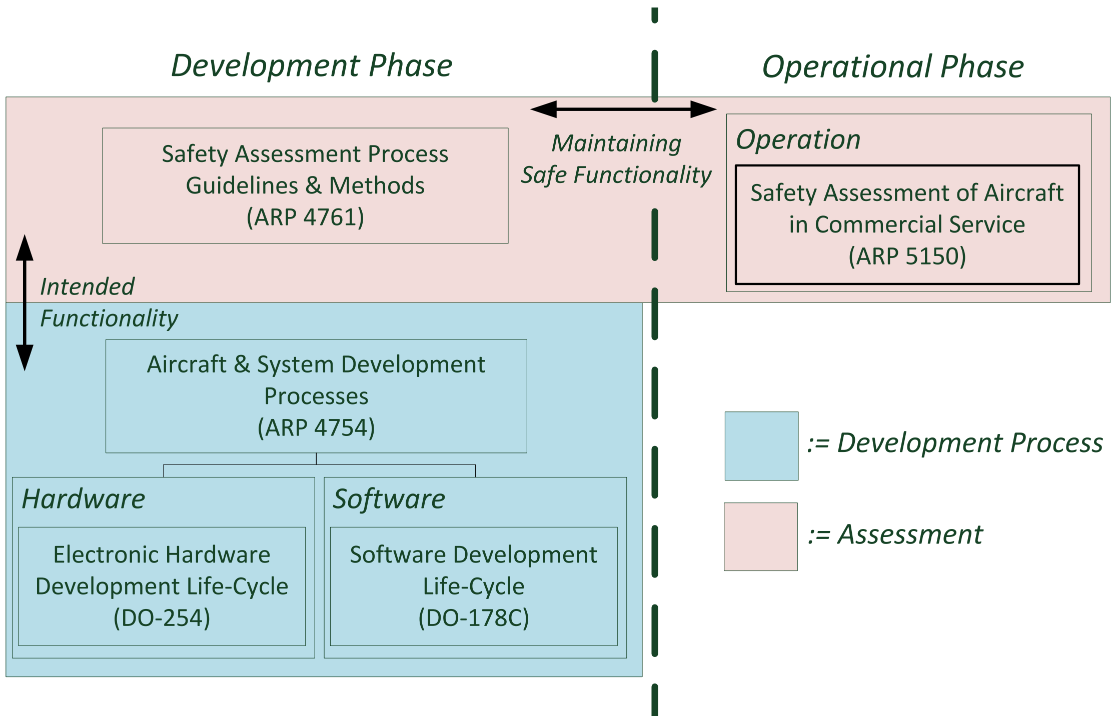

Development Phase
Operational Phase
Safety Assessment Process Guidelines & Methods (ARP 4761)
Maintaining Safe Functionality
Operation
Safety Assessment of Aircraft in Commercial Service (ARP 5150)
Intended Functionality
Aircraft & System Development Processes (ARP 4754)
Hardware
Electronic Hardware Development Life-Cycle (DO-254)
Software
Software Development Life-Cycle (DO-178C)
:= Development Process
:= Assessment

Figure 2.1.: Overview of aircraft development standards [118].

approved ML application in roles beyond assistance function [1]. At the same time, this development presents an opportunity to use runtime verification as a technique for providing safety layers for certification.

In addition to standards for development and assessment, there are also architectural standards that guide the integration of complex functions into unmanned aircraft. A prominent example is ASTM International's standard F3269-21 (Standard Practice for Methods to Safely Bound Flight Behavior of Unmanned Aircraft Systems Containing Complex Functions), which introduces a certification strategy for ensuring safe operation of systems that incorporate complex functions. A complex function can be a function that relies on a deep neural networks and is therefore difficult to certify using traditional methods. F3269-21 is based on a Run-Time Assurance (RTA) architecture, where the behavior of a complex function is bounded by a safety monitor [42], similar to the well-known Simplex architecture [41]. A simplified version of the architecture[2] of ASTM F3269-21 is shown in Figure 2.2. At the core of the architecture is a safety monitor that observes both the inputs and outputs of the complex function and determines whether it is behaving within expected, safe bounds. If a deviation is detected, the mon-

[1] `https://www.easa.europa.eu/en/document-library/general-publications/easa-artificial-intelligence-roadmap-20`

[2] In its original version, data is separated into assured and unassured data and data preparation components are added.

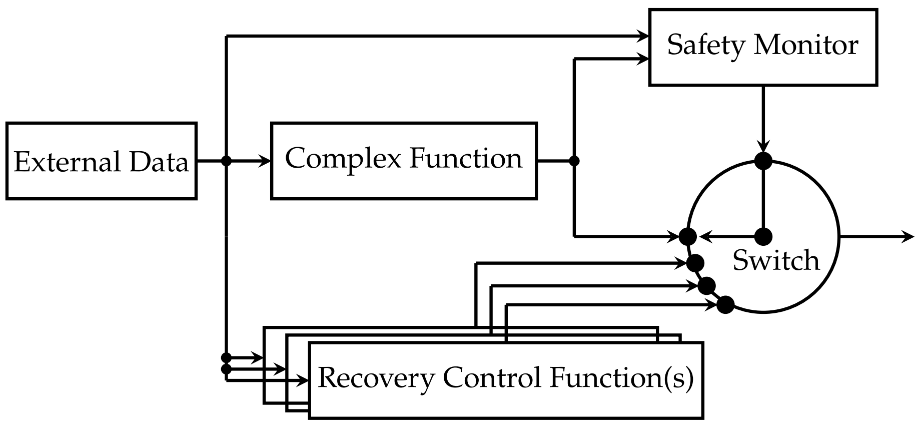


Figure 2.2.: A simplified version of the run-time assurance architecture proposed by ASTM F3269-21 to safely bound a complex function using a safety monitor.

itor switches the control from the complex function to a matching recovery function. For example, the monitor might continuously track the drone's current position and, upon detected a breach of the authorized flight area, could initiate a recovery response such as activating a parachute for a safe flight termination.

Remark 2.1 (Online and Offline Monitoring)

Online and offline monitoring differ primarily in when they are applied. Online monitoring is performed during operation, processing events as they occur in real time. Since the execution is ongoing, online monitors must react incrementally. In contrast, offline monitoring is conducted after the execution has completed, allowing the entire event trace to be analyzed, e.g., provided as a log file. This allows the use of more powerful and computationally intensive algorithms that are not suitable for real-time in general. In the context of aviation, online monitoring aligns with standards such as ASTM F3269-21, which focus on safety assurance during operation. Offline monitoring is valuable during function development, where it helps to uncover issues and generates verification artifacts that can support certification processes.

2.2. A Hierarchy for Monitoring Autonomous Systems

The importance of monitoring as highlighted in ASTM F3269-21 supports the view that the abstraction level traditionally used in manned aviation may be insufficient for monitoring autonomous systems, as their operational and mission boundaries become increasingly critical for the overall safety argument. To address this, we introduce a hierarchy of monitoring properties for autonomous aircraft systems, as depicted in Figure 2.3. This hierarchy is organized into multiple abstraction levels, each corresponding

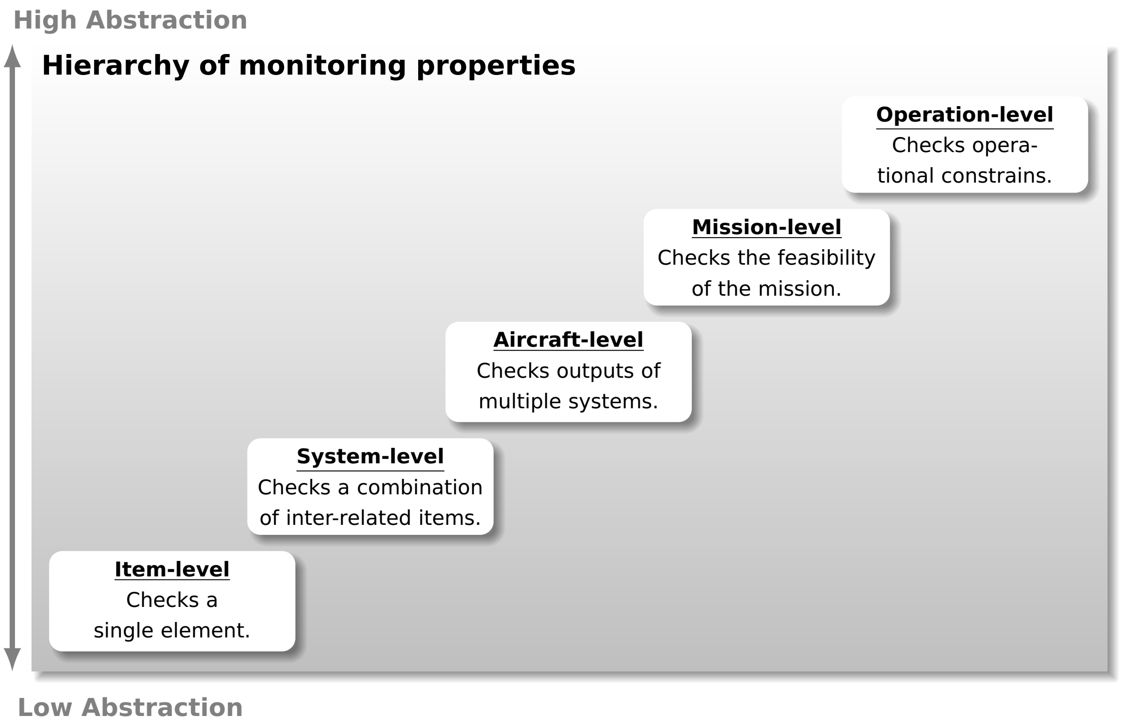

High Abstraction
Hierarchy of monitoring properties
Operation-level
Checks operational constrains.
Mission-level
Checks the feasibility of the mission.
Aircraft-level
Checks outputs of multiple systems.
System-level
Checks a combination of inter-related items.
Item-level
Checks a single element.
Low Abstraction

Figure 2.3.: A hierarchy of monitoring properties for autonomous systems.

to different monitoring levels of the autonomous software stack where monitoring plays a critical role in ensuring safe operation and enable system health management.

Our proposed abstraction levels extend the previous abstraction levels presented in SAE ARP4761. The lowest abstraction level is the item-level, which includes units that interact directly with the physical world. At this level, monitoring typically involves checking sensor or actuator outputs against pre-defined thresholds, expected frequencies, or signal characteristics. Above item-level is the system-level, which are monitoring properties on specific functions of the aircraft. For example, position estimation systems often integrate data from multiple, potentially dissimilar sensors. Monitoring at this level involves cross-validating these sources to detect inconsistencies or faults. Next is the aircraft-level, concerned with assessing the current status of the aircraft. In addition to validating consistency between redundant systems, this level includes monitoring the data-flow between system, i.e., inter-system communication with response times. For instance, consider two systems of the aircraft: one that controls the altitude of the aircraft and another system that assesses the current position. In this case, monitoring ensures that control commands are effective, e.g., "When an altitude increase is commanded, an altitude increase must be observed within three seconds." A new level that is above the aircraft-level is the mission-level. A mission defines a specific objective of an aircraft, e.g., "transport goods from A to B while maintaining a distance of 200 meters to C". Monitoring at this level evaluates whether mission properties are being upheld, using data from lower levels to assess if the overall mission remains feasible, e.g., is the state of charge of the battery sufficient to reach the target location. Finally, at the top of the hierarchy is the operation-level, which governs the validity of entire missions. This includes operational limits and safety rules. A typical property at this level is geofencing. Geofencing defines a virtual barrier beyond which aircraft are not allowed to operate. For instance, a geofence can capture the boundaries of a field within which an agricultural drone is free to operate. Violating operation-level properties often involves the activation of contingency or emergency procedures.

Remark 2.2 (Hierarchy used by FAA and NASA)

The hierarchy was used by FAA and NASA for flight testing their system architecture that integrates Automatic Dependent Surveillance-Broadcast (ADS-B) and Global Positioning System (GPS) Localizer Performance Vertical (LPV) guidance technology onto NASA's high-altitude research airplane [119].

2.3. Types of Monitoring Properties

Besides the level at which monitoring is applied, it is important to consider the types of monitoring properties that can be expressed. For each type, we provide illustrative examples corresponding to different levels within the hierarchy. Some of these examples are then formalized using different specification languages. While we do not present the full syntax and semantics of these languages, we will show their general structure and provide references for further reading.

Note that by "types" of monitoring properties, we do not refer to classical classifications such as liveness and safety [120, 121], nor to the hierarchy of temporal properties introduced by Manna and Pnueli [122]. Instead, we use "types" to describe practical categories of properties that arise in monitoring scenarios.

Remark 2.3 (Practical Trade-off: Choosing Between Expressiveness and Simplicity)

As an engineer, there is a trade-off between the complexity of learning a specification language and the expressiveness it offers. For example, a language that supports asynchronous communication typically introduces additional operators to reason about out-of-sync data, but it also requires the engineer to understand subtle semantic nuances and corner cases.

2.3.1. Temporal Properties

Properties that reason about temporal behavior are among the most common language features in Runtime Verification. Runtime Verification is a research field concerned with

Hierarchy	**Example property**
Item-level	The LiDAR sensor shall produce at least one valid measurement every second.
System-level	The integral over one second of acceleration readings shall reflect the change in velocity within the last second.
Aircraft-level	Whenever a command to increase altitude is given, a corresponding behavior shall be observed by an altimeter reading within one second.
Mission-level	The aircraft shall fulfil its mission to fly to Waypoint G, reaching first Waypoint A and then Waypoint B in between.
Operation-level	Whenever an emergency helicopter enters the airspace, the aircraft shall trigger a safe landing within 30 s.

Table 2.4.: Example temporal properties for each level in the monitoring hierarchy.

checking whether a system's execution *trace*, i.e., a sequence of observed events, satisfies a given specification during or after execution. Temporal properties are typically expressed using the operators $\Diamond$, $\Box$, and $\mathcal{U}$, which build upon propositional logic, i.e., set of atomic propositions combined using connectives like $\wedge, \neg$. For propositions a and b, the formula $\Diamond\, a$ states that *eventually* a must become *true*, $\Box\, a$ requires that a is *always true*, and $a\ \mathcal{U}\ b$ specifies that a must hold *until* b becomes *true*.

Table 2.4 depicts some example properties for each abstraction level. Temporal properties range from properties that specify real-time properties like "The LiDAR sensor shall produce at least one valid measurement every second" to properties on the order of events like "The aircraft shall fulfil its mission to fly to Waypoint G, reaching first Waypoint A and then Waypoint B in between". Using atA, atB, and atG as propositions, which become *true* when the aircraft is near waypoints A, B, and G, respectively, the property can be expressed in Linear Temporal Logic (LTL) as: $((\neg atB)\ \mathcal{U}\ atA) \wedge ((\neg atG)\ \mathcal{U}\ atB) \wedge \Diamond\ atG$. By using $\mathcal{U}$, the formula enforces the order of events, e.g., $((\neg atB)\ \mathcal{U}\ atA)$ ensures that the aircraft reaches B only after it has reached A.

A variant of LTL is Mission-Time Linear Temporal Logic (MLTL) which syntactically extends LTL by introducing bounded, closed, and discrete intervals for the temporal operators [123]. The bounded and closed intervals related to the finite length of a mission, while the discrete nature assumes time steps hat may correspond to clock ticks with a frequency of one second. For example, the formula $\Diamond_{[0..10]} land$ specifies that the aircraft must land within ten seconds. Both LTL and MLTL assume a synchronous model for incoming data, meaning that all input events in the trace arrive simultaneously at each time step. However, there are specification languages designed to handle asynchronously arriving input data. One example is the stream-based language RTLola; an example specification is provided in Section 2.3.2.

Remark 2.4 (Monitorability)

A fundamental limitation shared by all monitoring languages is that not all properties are monitorable. For example, the property $\square\Diamond a$, which asserts that proposition a must always eventually occur, is not monitorable. Intuitively, given only a finite trace up to the current point in time, a monitor cannot determine whether another occurrence of a will arrive in future. As a result, the monitor is forced to wait possibly indefinitely, unable to emit a definite Boolean verdict. Formally, monitorability is defined in [124, 67] using the notions of *good*, *bad*, and *ugly* prefixes. A good prefix is a finite trace for which all possible continuations satisfy the property; a bad prefix is one for which all continuations violate it. An ugly prefix is a finite trace for which no continuation can be classified as definitely good or bad prefix, i.e., satisfaction or violation remain possible, regardless of future events. For instance, any finite trace that contains an occurrence of a is a good prefix for the formula $\Diamond a$. The formula $\square\Diamond a$ from before only has ugly prefixes. In fact, a formula is considered monitorable, if it does not have any ugly prefixes. To provide more informative feedback in the case of an ugly prefix, the verdict domain can be extended to four values by adding *presumably true*, and *presumably false*. For example consider a request-response property $\square(r \rightarrow \Diamond g)$, which states that every request r must eventually be granted g. If, at a given point in the trace, there is an outstanding request that has not yet been granted, the monitor will output *presumably false*. Conversely, if there are no pending requests, the monitor will emit *presumably true*, acknowledging that while the verdict is not final, the trace will satisfy the property if no further requests arrive.

2.3.2. Statistical Properties

The previous section described properties using atomic propositions, i.e., values that can be either *true* or *false*. To apply this approach, sensor or control values must be preprocessed to translate them to propositions. This works well for properties like the request-response property presented in Remark 2.4. However, to gain deeper insights and effectively assess the health of a sensor, the monitor should operate directly on the actual numerical values rather than on Booleans. Consider the item-level property in Table 2.5 as an example. It can be expressed by the logical formula $\square$ *altitude_varies_at_most_five_meters*. Yet, the proposition abstracts away much valuable information that ideally should be part of the monitor description, e.g., threshold values for sensor readings.

To address this, stream-based specification languages provide a more expressive alternative. One such language is RTLola [125], where each sequence of sensor or control

Hierarchy	Example property
Item-level	The average altitude over one second should only vary each second by at most five meters.
System-level	The perception module shall receive values by the LiDAR and camera sensor by a maximum offset of 100 ms.
Aircraft-level	The maximal path deviation of the aircraft within the last 30 s shall be at most ten meters.
Mission-level	During flight, a progress percentage towards reaching Waypoint A shall be observed.
Operation-level	At least two contingencies must still be possible with the current battery consumption.

Table 2.5.: Example statistical properties for each level in the monitoring hierarchy.

values can be represented as a dedicated `input` stream. `Output` streams aggregate, combine, and transform these input streams (or other output streams) to compute valuable statistics about the system. Similar to output streams, `trigger` streams can be defined to produce Boolean verdicts that reflect the system's health based on the computed statistics. Using RTLola, the item-level property in Table 2.5 is expressed as:

```
input altitude: Float64                                                          1
output avg_alt @1Hz := altitude.aggregate(over: 1s, using: avg).defaults(to: 0.0)  2
trigger |avg_alt - avg_alt.offset(by: -1, or: avg_alt)| > 5.0 "Deviation to high!"  3
```

In this specification, the monitor receives directly actual sensor values, as illustrated by the input stream `altitude`. In Line 2, it computes the average of the `altitude` values over a one second window with a frequency of one Hertz. If no value is received within the window, the average computation defaults to `0.0`. Line 3 then defines a trigger stream that compares the current value of `avg_alt` with its previous value, accessed using the `offset` operator. If no such value exists, the operator defaults to the current `avg_alt` value. In this example, this is the case at the beginning of the trace. If the computed difference exceeds `5.0` meters, a message is sent to the user to report the threshold violation. A more detailed introduction to RTLola is given in Section 3.2.

→Sec. 3.2, p. 38

Remark 2.5 (Aggregation Functions)

Note that the `aggregate` operator requires its aggregation function to be a list homomorphism to ensure an efficient evaluation of the real-time windows at runtime requiring only finite memory [126]. A list homomorphism is a function on lists, which basically allows to split the list any way, apply the function to the parts independently, and then combine the results. Fortunately, many common statistical functions, such as the average, satisfy this condition. However, some statistical functions such as the median are not list homomorphisms, as they require access to the entire list to compute the result. These functions may demand unbounded memory, making them unsuitable for real-time monitoring. In practice, online monitors that run during flight should be limited to list-homomorphic aggregation functions. Yet, this restriction does not apply to offline log analysis, where full traces are available and memory is not necessarily a constraint. All properties given in Table 2.5 can be expressed using list-homomorphic aggregation functions.

2.3.3. Parameterized Properties

Parameterized properties define a family of properties that are distinguished by one or more parameters. They often appear in first-order temporal logics or stream-based languages and require the dynamic creation of multiple property instances based on the values of observed parameters. For example, while the number of output streams was bounded in Section 2.3.2, parameterized properties use stream templates that are instantiated when a `spawn` condition holds and closed when the instance's `close` condition is satisfied. Table 2.6 provides example properties for each level in the hierarchy.

Consider the given mission-level property that checks “Whenever a new waypoint is added to the waypoint list, this waypoint shall be reached within the specified time bound”. Here, the number of waypoints, their values, and their time bound are not known beforehand and are only provided at runtime. Using RTLola, the property is formalized in Listing 2.1. The specification uses the input stream `pos` to receive the current position of the vehicle, the input `new_wp` to receive new target waypoints, and `timebound_new_wp_in_secs` to specify the time bound within which the received target waypoint is expected to be reached (Lines 1 to 3). For each new waypoint, an instance of three stream templates is spawned (Lines 5, 7, and 12) To determine whether a waypoint has been reached, the monitor computes the distance between the current position and the corresponding target waypoint (Line 5). Next, the specification defines a stream template that counts the number of seconds elapsed, based on its sampling frequency of one Hertz (Lines 7 to 10). Last, this timer is used in the `violation` template, which checks if specified `duration` has been exceeded, i.e., the time bound (Lines 12 to 15). If the time bound is violated, the trigger is activated and the user is notified (Lines 17 to 18).

Remark 2.6 (Unbounded Number of Instances)

Whenever a parameterized property may lead to an unbounded number of instances, the monitor would require an unbounded amount of memory. To ensure efficient runtime monitoring, this challenge must be addressed, e.g., by placing restrictions on the number of active instances.

Hierarchy	**Example property**
Item-level	Whenever a specific actuator position is commanded, the corresponding actuator feedback should be received within two seconds.
System-level	Whenever a candidate landing site has been tracked by a perception algorithm, the tracked candidate must be persistently tracked the next three seconds before it is classified as an alternative landing site.
Aircraft-level	Whenever an intruder aircraft is detected, the intruder must be characterized as threat when entering the remain-well-clear volume.
Mission-level	Whenever a new waypoint is added to the waypoint list, this waypoint shall be reached within the specified time bound.
Operation-level	A new polygon that represents a *stay-out* region, where the aircraft is not allowed to fly, is added.

Table 2.6.: Example parameterized properties for each level in the monitoring hierarchy.

```
input pos: (Float32, Float32, Float32) // (x,y,z)-Position
input new_wp: (Float32, Float32, Float32) // (x,y,z)-Position
input timebound_new_wp_in_secs: UInt32 // time bound in seconds
// Computes when a waypoint is reached
output reached(wp) spawn with new_wp eval with |pos - wp| < ϵ close reached(wp)
// Represents the elapsed time since the new waypoint was provided
output wp_timer (wp)
  spawn with new_wp
  eval @1Hz with wp_timer(wp).offset(by: -1, or: 0) + 1
  close reached(wp)
// Boolean template that checks whether the timer is exceeded
output violation (wp, duration)
  spawn with (new_wp, timebound_new_wp_in_secs)
  eval with wp_timer(wp) > duration
  close reached(wp)
// Notifies the user if any waypoint was not reached in time.
trigger violation.aggregate(over_instances: all, using: ∃)
        "WARNING: Waypoint not reached in time."
```

Listing 2.1: An RTLola specification that uses parametrization to monitor whether a new waypoint is reached within a specified time bound.

2.3.4. Spatial Properties

Spatial properties allow to specify changes in space, assuming a spatial model. A common spatial model is a weighted graph, as used by *Spatio-Temporal Reach and Escape Logic (STREL)* [127, 128], where nodes and edges have physical and logical attributes that may evolve over time. A spatial representation is essential, as spatial operators explore the possibility of an event in space. For example, to evaluate the property "there must always be an emergency landing site within one kilometer" the current position of the aircraft and the locations of all landing sites must be known. To formalize such a requirement, STREL extends temporal logic with spatial operators that are augmented by elements d of a distance domain and a distance function f that maps paths of the weighted graph to the distance domain. Given a spatio-temporal property ϕ, where temporal operators include the standard one such as $\square$ and $\diamond$, the following spatial operators are additionally available:

- ϕ_1 **reach** $^{f}_{[d_1,d_2]}$ ϕ_2 is satisfied if, from a location where ϕ_1 holds, there exists a path to a location where ϕ_2 holds, such that the distance along the path, as compute by the function f , lies within the interval $[d_1, d_2]$,
- **escape**$^{f}_{[d_1,d_2]}$ ϕ is satisfied if, from the current location, there exists a path whose total distance lies within $[d_1, d_2]$, and ϕ holds at every location along the path,
- **somewhere**$^{f}_{\leq d}$ ϕ is satisfied if there exists at least one location within distance d from the current location where ϕ holds,
- **everywhere**$^{f}_{\leq d}$ ϕ is satisfied if, at all locations within distance d from the current location, the property ϕ holds.

Given a spatial model where nodes represent the aircraft and alternative landing sites, the weights of the edges represent the Euclidean distance in meters, and the distance function *dist* computes the sum of weights along a path, the requirement from above can be formalized as $\square$(**somewhere**$^{dist}_{[0,1000]}$(`node = emergency_landing_site`)). Additional spatial properties are listed in Table 2.7. Note that the examples for item-level and system-level introduce spatial properties using a spatial model that represents a network topology where weights on edges indicate the latency. For example, given the *hop* distance function that counts the number of edges along a path, the requirement "the mission manager (mm) must be directly connected to the ground control station (gcs)" can be specified as (`node = mm`) **reach**$^{hop}_{[0,1]}$ (`node = gcs`).

Hierarchy	**Example property**
Item-level	The output of the sensor must be broadcasted to all subscribers with a latency of at most one second.
System-level	The mission manager must be directly connected to the ground control station.
Aircraft-level	The average noise level within one kilometer of houses should always be less than 50 dB.
Mission-level	There must always be an alternative trajectory leading to an alternative landing site within 200 m.
Operation-level	There shall always be an emergency landing site within one kilometer.

Table 2.7.: Example spatial properties for each level in the monitoring hierarchy.

Remark 2.7 (Relationship Between Types)

There is a strong relationship between the different types of properties, and in practice, they are often combined to capture more expressive properties. For example, statistical properties are frequently also temporal, as they involve aggregating or analyzing sequences of values over time rather than evaluating isolated inputs. Similarly, parameterized properties often incorporate temporal or spatial aspects, depending on the nature of the monitored system and the property's intent.

2.4. Discussion on Monitoring Specification Languages in Practice

Next, we discuss the advantages and disadvantages of using a formal language to specify monitoring properties.

Disadvantages

There is no language to rule them all. We have presented three specification languages that complement each other: MLTL, RTLola, and STREL. Currently, there is no language that covers all property types. Further, to provide guarantees such as memory bounds, some specification languages restrict their expressiveness to efficient fragments [126].

Mostly academic tools. There is ongoing active research on specification languages, with new operators being introduced and existing operators being changed, making it challenging to stay up to date. While there have been initial steps towards the commer-

cialization of runtime monitoring tools based on such specification languages, no fully industrial-grade tool has yet been established that can be used for most of the cases.
Yet another language to learn. Formal methods and specification languages in particular are not part of standard engineering education, which makes it sometimes difficult for engineers to apply such methods. The engineers require clear benefits before investing in learning.
Never change a running system. Often there is legacy code that already contains monitor code. It is an investment to extract this code and replace it, especially when the code was already approved.
There are properties that cannot be expressed. Specification languages have a clear syntax that has benefits like an automated analysis of the written specification, but also downsides when the syntax of the language limits expressiveness.

Advantages
Formalization process helps to think about monitoring properties. When formalizing a requirement, the language already guides towards specifying "good" monitoring properties and highlights aspects that might be missing. For instance, properties can be inefficient to monitor or default values may need to be provided for accesses non-existing past or future events.
Specification languages are concise and precise. When familiar with the language, specification are easy to write since the syntax is concise and precise. Especially, during development, when properties change frequently, this helps improve maintenance and also reduces errors when writing code by hand. This is especially important when considering future autonomous aircraft systems that will include more and more black-box components that require monitoring capabilities.
Analysis of specification language. Formal specification languages are designed with a strong focus on analyzability. For instance, RTLola allows for static analysis to determine upper bounds on the memory consumption of the generated monitor implementation. Further, specification analysis also enables consistency checking and to provide verified guarantees on the behavior of the monitor [50] that help argue ASTM F3269-21 monitor coverage.
Automatic generation of monitor implementation. Monitors can be automatically synthesized from a specification. The previously mentioned analyses also produce artifacts that support the safe integration of these monitors into systems. Also, the implementation whether in software or hardware is decoupled from the specification. This decoupling allows flexibility in choosing or switching between implementation languages such as C or VHDL, without modifying the specification. In addition, optimizations made at the specification level automatically propagate to all generated implementations.
Closes gap between natural language and implementation. Operators like *always* ($\square$), *eventually*($\diamondsuit$), or *aggregate(over: 1s, using: avg)* are close to natural language requirements. This alignment provides a clear advantage for traceability during certification.

```
input alt_cmd, alt :Float64, Float64                                                 1
output inc : Bool := alt > alt.offset(by:-1,or:alt) > alt.offset(by:-2,or:alt)       2
trigger spawn when alt_cmd > 0.0 eval @1Hz when true with "Violation" close when inc  3
```

Listing 2.2: An RTLola specification that represents "when the command to increase altitude is given, then within the next second the altitude should be increasing for at least three consecutive altimeter readings"

Monitoring is a common capability of modern everyday software systems. However, specification-based monitoring has not yet seen widespread adaptation. The disadvantages show that there are challenges ahead. Most importantly, specification languages need to converge to a stable version, maybe even with an industrial product, before they can be widely used in industry. Nevertheless, the advantages also show that monitoring specification languages offer significant benefits, particularly for future autonomous aircraft systems, which often incorporate black-box components. In such systems, monitoring complex properties is essential for ensuring safe integration and operation.

To illustrate the benefits of using a monitoring specification language over handwritten code, we consider the following property: "when the command to increase altitude is given (i.e., `alt_cmd` > 0), then within the next second the altitude should be increasing for at least three consecutive altimeter readings". This property can be formalized in MLTL as $\Box(\texttt{alt\_cmd} > 0 \rightarrow \Diamond_{[0,1]}\ \texttt{alt\_increased\_more\_than\_3})$ where `alt_increased_more_than_3` is an atomic proposition that is computed outside the specification. The same property can be expressed in RTLola, as shown in Listing 2.2 where altitude increase is computed directly within the specification. Line 2 checks if the last three altitude values are strictly increasing by accessing the current and the two preceding ones. In Line 3, if the spawn condition holds, i.e., an altitude command is given, the trigger is evaluated and will notify a violation unless the instance is closed beforehand, which happens if three consecutive increasing altitude readings are observed.

A corresponding handwritten monitor written in pseudo code is depicted in Listing 2.3 which for simplicity assumes that both inputs arrive synchronously. Here, inputs are read in Line 20 and previous input values are updated correspondingly from Line 22 to Line 33. To track the expiration of the time limit of one second due to a positive altitude command (Line 35), the *obligation* variable is set to *false* and a separate thread is spawned (Lines 36 to 37) that sleeps for one second and checks whether the *obligation* was satisfied in the meantime (Lines 5 to 12). The *obligation* represents an increase in altitude over three consecutive altitude readings (Line 40). Given this explanation, it is clear how it matches the monitor property. Yet, without such an explanation, it is quite hard to understand the task of the monitor. In contrast, the specification language offers a more compact representation. Even if one is not familiar with the details of the specification language, the property is more intuitive to understand since it is more compact and directly supports temporal notions. Further, memory management and race conditions of concurrent computations are prone to errors and can be avoided if a dedicated monitoring language is used.

2.5. Summary

This chapter introduced a hierarchy of monitoring properties for autonomous systems, particularly in the context for (unmanned) aircraft. The chapter defined five level of abstractions: item, system, aircraft, mission, and operation. It further presented four types of monitoring properties: temporal, statistical, parameterized, and spatial. These types were illustrated with examples across all levels. Formal specification languages such as MLTL, RTLola, and STREL were used to formalize several of these properties, showing their advantages in clarity, analyzability, and traceability. While challenges remain, specification-based monitoring seem promising, particular for systems that integrate black-box or highly complex components and require strong assurance guarantees. These developments align with emerging trends in certification standards,e.g., ASTM F3269-21, which emphasize the need.

```
1   // Global variables
2   double alt_cmd, alt, alt_past_1, alt_past_2;
3   bool watchdog_active, obligation;
4
5   void timer(){ // Represents the time limit of the watchdog
6      watchdog_active = true;
7      sleep(1);
8      if(!obligation){
9         print("Violation");
10     }
11     watchdog_active = false;
12  }
13
14  void monitor() {
15     int default_count = 2;
16      watchdog_active = false;
17     while(1) {
18        //read inputs
19        double tmp_alt = alt;
20        alt_cmd, alt = read_blocking() // Reads inputs and blocks if no inputs are present
21        // updates past values by respecting default values if value does not exist yet.
22        if (default_count == 2){
23           alt_past_1 = alt; // no previous value does exist
24           alt_past_2 = alt; // no previous value does exist
25           default_count--;
26        } else if (default_count == 1){
27           alt_past_2 = alt; // no previous value does exist
28           default_count--;
29        }
30        else {
31           alt_past_2 = alt_past_1;
32           alt_past_1 = tmp_alt;
33        }
34        // activates watchdog if positive altitude command was given.
35        if(alt_cmd > 0 && !watchdog_active){
36           obligation = false;
37           thread watchdog(timer); // Spawn a thread that executes the function timer(),
                 see Line 5
38        }
39        // updates obligation if altitude was increasing for three consecutive altitude
             readings.
40        obligation = alt > alt_past_1 && alt_past_1 > alt_past_2;
41     }
42  }
```

Listing 2.3: A handwritten monitor that represents "when the command to increase altitude is given, then within the next second the altitude should be increasing for at least three consecutive altimeter readings"

Chapter 3

Specification Languages

In this chapter, we introduce three specification languages. We begin with the Signal Temporal Logic (STL) [129], then present the stream-based language RTLola [130, 35, 126], and finally introduce Temporal Behavior Trees (TBTs). Using illustrative examples, we demonstrate how logics and stream-based languages can be used.

3.1. Signal Temporal Logic

A trace is a finite sequence of observable states of a system: $\sigma(1), \ldots, \sigma(N)$, wherein $\sigma(i)$ denotes the state at time $i \in [1, N]$. We write the length of the trace as $|\sigma|$. A trace with $|\sigma| = 0$ is the *empty trace*. Given trace σ, we denote $\sigma[l : u]$ as a slice of the trace from "offsets" l to u (inclusive), as follows:

$$\sigma[l : u] = \begin{cases} \sigma(l+1), \cdots, \sigma(\min(u+1, N)), & \text{if } l \leq u \text{ and } l < N \\ \text{empty trace} & \text{otherwise} \end{cases}$$

For convenience, let $\sigma[l :] = \sigma[l : N-1]$ and $\sigma[: u] = \sigma[0 : u]$. Note that offsets begin with 0 and end at $|\sigma| - 1$. For instance, $\sigma[0 : 3] = \sigma(1), \ldots, \sigma(4)$ and $\sigma[5 :] = \sigma(6), \ldots, \sigma(N)$.

Example 3.1.1. The state $\sigma(i) : (x_{uav}, y_{uav}, z_{uav}, x_s, y_s, z_s, h_s)$ for $1 \leq i \leq N$ describes the positions of a UAV and a ship, wherein the subscript s denotes the state of the ship, *uav* to those of the UAV, and h_s is the heading angle of the ship. △

Remark 3.1 (Assumption on Time)

Many real-time logics treat time as a continuous variable and each state has a corresponding time stamp. Here, we will simply assume that the trace states occur at regular time intervals and thus $\sigma(i)$ is associated with some time $t = i\tau$ for a time period $\tau > 0$. The precise value of τ is not required.

An STL formula extends propositional logic by temporal operators that access future trace events. Let us define a set of *atomic propositions* $\mathrm{AP} = \{p_1, \ldots, p_m\}$ where (for the sake of convenience), each p_i is associated with a function f_i that maps states to a real number such that $\sigma \models p_i$ for a state σ iff $f_i(\sigma(1)) \geq 0$.

Definition 3.1 (STL Syntax)
STL formulas over AP have the following syntax:

$$\begin{array}{lll}
\varphi & := \mathrm{AP} & \leftarrow \text{Atomic propositions} \\
& \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi & \leftarrow \text{Boolean combinations} \\
& \mid \Diamond_{[l,u]}(\varphi) & \leftarrow \text{Eventually with interval} \\
& \mid \Box_{[l,u]}(\varphi) & \leftarrow \text{Globally over interval} \\
& \mid \varphi\, \mathcal{U}_{[l,u]}\, \varphi & \leftarrow \text{Until with interval}
\end{array}$$

Note that some of the formulas are indexed by an interval $[l, u]$ wherein $0 \leq l \leq u \leq \infty$. We write $\Diamond(\varphi)$ as a shorthand for $\Diamond_{[0,\infty)}(\varphi)$, and similar conventions apply for $\Box(\varphi)$ and $\varphi_1 \mathcal{U} \varphi_2$. The "next-state" operator $\bigcirc(\varphi)$ is syntactic sugar for $\Diamond_{[1,1]}(\varphi)$.

Intuitively, the formula $\Diamond_{[l,u]}(\varphi)$ means that φ must *eventually* hold at some point within the time interval $[l, u]$. The formula $\Box_{[l,u]}(\varphi)$ expresses that φ must hold *always* throughout the time interval $[l, u]$. The formula $\varphi_1\, \mathcal{U}_{[l,u]}\, \varphi_2$ requires that φ_1 holds *until* φ_2 holds.

In general, STL formulas are defined over infinite traces. However, we consider slices of the form $\sigma[i : j]$, which represent finite traces as typically used in monitoring. There have been many approaches to provide finite trace semantics for temporal logics [124, 131]. They differ on how they treat the truth of a formula at the end of the trace. We will follow a simple approach along the lines of De Giacomo and Vardi [68]. We write $\sigma \models \varphi$ to denote that the finite trace σ satisfies the STL formula φ.

Definition 3.2 (STL Boolean Semantics)
The Boolean semantics of STL is defined as follows:

$$\begin{array}{lll}
\sigma \models p_i & \iff & f_i(\sigma(1)) \geq 0 \text{ if } |\sigma| > 0 \text{ else } \mathit{false} \\
\sigma \models \neg\varphi & \iff & \sigma \not\models \varphi \\
\sigma \models \varphi_1 \wedge \varphi_2 & \iff & \sigma \models \varphi_1 \wedge \sigma \models \varphi_2 \\
\sigma \models \varphi_1 \vee \varphi_2 & \iff & \sigma \models \varphi_1 \vee \sigma \models \varphi_2 \\
\sigma \models \Diamond_{[l,u]}(\varphi) & \iff & \exists\, i \in [l, u],\ \sigma[i:] \models \varphi \\
\sigma \models \Box_{[l,u]}(\varphi) & \iff & \forall\, i \in [l, u],\ \sigma[i:] \models \varphi \\
\sigma \models \varphi_1\, \mathcal{U}_{[l,u]}\, \varphi_2 & \iff & \exists i \in [l, u],\ \ (\forall j \in [0, i-1],\ \sigma[j:] \models \varphi_1) \wedge \sigma[i:] \models \varphi_2
\end{array}$$

Note that, unlike the semantics for infinite traces, the semantics of STL for finite traces can be non-intuitive especially when the truth of formulas involves reasoning beyond the end of the trace. For instance, the formula $\bigcirc p$ is always false for an empty trace, whereas, $\neg \bigcirc \neg p$ is always true. $\Diamond\varphi$ in a finite trace semantics specifies that φ becomes true before the trace ends. $\Box\varphi$ specifies that φ remains true until the trace ends.

Example 3.1.2. Recalling the definition of a state from Example 3.1.1, the STL formula $\Diamond\Box_{[0,5]}\ p_{behind}$ represents that the position of the UAV should eventually be 20 meters behind the ship, and remain in that position for 5 time steps. The function f_{behind} allows two meters deviation from a point that is 20 meters behind the ship based on its current heading angle: $f_{behind} = 2 - \sqrt{\begin{array}{c}((x_s + 20 \cdot \cos(180 + h_s)) - x_{uav})^2 + \\ ((y_s + 20 \cdot \sin(180 + h_s)) - y_{uav})^2 + ((z_s + 20) - z_{uav})^2\end{array}}$. △

We define the robustness of an STL formula on a trace as a numerical value that we provide to a trace with respect to an STL formula. Positive values of robustness denote that a trace satisfies a property whereas negative values indicate the reverse. Finally, the magnitude of the robustness provides us with a measure of how “far away” a trace that satisfies the formula is from violating it, or vice-versa [70, 69]. Robustness measures are useful for a variety of applications, including runtime monitoring and falsification [17].

Definition 3.3 (STL Robust Semantics)

Robustness of an STL formula φ over a trace σ, denoted $\rho(\varphi, \sigma)$, is defined as follows:

$$\begin{array}{lcl}
\rho(p_i,\ \sigma) & = & f_i(\sigma(1))\ ,\text{if } |\sigma| > 0 \text{ else } -\infty \\
\rho(\neg\varphi,\ \sigma) & = & -\rho(\varphi,\sigma) \\
\rho(\varphi_1 \wedge \varphi_2,\ \sigma) & = & \min(\rho(\varphi_1,\sigma), \rho(\varphi_2,\sigma)) \\
\rho(\varphi_1 \vee \varphi_2,\ \sigma) & = & \max(\rho(\varphi_1,\sigma), \rho(\varphi_2,\sigma)) \\
\rho(\Diamond_{[l,u]}(\varphi),\ \sigma) & = & \max\limits_{i\in[l,u]}\ \rho(\varphi, \sigma[i:]) \\
\rho(\Box_{[l,u]}(\varphi),\ \sigma) & = & \min\limits_{i\in[l,u]}\ \rho(\varphi, \sigma[i:]) \\
\sigma \models \varphi_1\ \mathcal{U}_{[l,u]}\ \varphi_2 & = & \max\limits_{i\in[l,u]} \min(\rho(\varphi_2, \sigma[i:]), \min\limits_{j\in[0,i-1]} \rho(\varphi_1, \sigma[j:]))
\end{array}$$

Example 3.1.3. To compute the robustness of the STL formula $\Diamond\Box_{[0,5]}\ p_{behind}$ given in Example 3.1.2, a naive algorithm could iterate over all time steps, calculate the minimal value of the six consecutive p_{behind} values, and then take the maximum of theses minimal values. If this final value is negative, the formula is violated. Otherwise, it is satisfied. △

Theorem 1. *For trace σ and STL formula φ, $\sigma \models \varphi$ iff $\rho(\varphi, \sigma) \geq 0$.*

Proof. We prove the theorem by structural induction on φ. The full proof is given in Section A.2.1. □

➔Sec. A.2.1, p. 163

3.2. RTLola

While many specification languages rely on logic, RTLOLA is fundamentally stream-based. An RTLOLA specification treats trace events as *input streams* and derives verdicts through the computation of *output streams*. We provide a brief overview of RTLOLA next, focusing on the operators relevant to this work. For a comprehensive introduction, see [125]. We begin by introducing RTLOLA without its real-time features, referred to as LOLA [130].

Definition 3.4 (LOLA Syntax [130])

LOLA is a stream language describing the translation from input to output streams:

$$\begin{aligned}
&\textbf{input } t_1 : T_1 \\
&\quad\vdots \\
&\textbf{input } t_m : T_m \\
&\textbf{output } s_1 : T_{m+1} := expr_1(t_1, \ldots, t_m, s_1, \ldots, s_n) \\
&\quad\vdots \\
&\textbf{output } s_n : T_{m+n} := expr_n(t_1, \ldots, t_m, s_1, \ldots, s_n) \\
&\textbf{trigger } \varphi \; message
\end{aligned}$$

where input streams $t_1, \ldots, t_m$ and output streams $s_1, \ldots, s_n$ are called *independent* and *dependent variables*, respectively. Each variable is typed: independent variables t_i are typed T_i and dependent variables s_i are typed T_{m+i}. A trigger generates a *message* when its corresponding condition φ becomes *true*.

Intuitively, input streams carry synchronously arriving data from the system under scrutiny, output streams represent calculations, and triggers generate notification *message*s at instants where their condition φ becomes *true*. Dependent variables are computed based on *stream expressions* $expr_1, \ldots, expr_n$ over dependent and independent stream variables. A (stream) expression is one of the following:

- an atomic expression c of type T if c is a constant of type T;
- an atomic expression s of type T if s is a stream variable of type T;
- an expression $ite(b, expr_1, expr_2)$ of type T if b is a Boolean expression and $expr_1, expr_2$ are expressions of type T. Note that *ite* abbreviates the control construct *if-then-else*;

- an expression $f(expr_1, \ldots, expr_k)$ of type T if $f : T_1 \times \cdots \times T_k \mapsto T$ is a k-ary operator and $expr_1, \ldots, expr_k$ are expressions of type $T_1, \ldots, T_k$;

- an expression $o.offset(by : i, or : d)$ of type T if o is a stream variable of type T, i is an Integer, and d is of type T.

We provide some practical examples next before providing Lola's semantics.

Example 3.2.1. We consider a Lola specification for an altitude bound check:

```
input altitude: Float32 // in m
output altitude_bound := altitude > 200.0
trigger altitude_bound "Warning: Decrease altitude!"
```

This Lola specification defines a single input stream `altitude` (Line 1) for an unmanned aircraft flying. Based on this input, it computes the output stream `altitude_bound`, which checks whether the `altitude` exceeds its operating limit of `200.0` meters (Line 2). If a violation occurs, the trigger emits a warning (Line 3). Note that the stream type `bool` is not explicitly declared for `altitude_bound`, but instead inferred automatically. $\triangle$

Lola uses temporal operators that allow output streams to access its and others previous and future stream values. The stream

```
output alt_count := if altitude ≤ 200.0 then 0 else alt_count.offset(by: -1, or: 0) + 1
```

represents a count of consecutive altitude violations by accessing its own previous value, i.e., `offset(by: x, or: d)` where a negative and positive integer `x` represents past and future stream accesses, respectively. Since temporal accesses are not always guaranteed to exist, `d` defines a value which is used instead. Here, at the first position of `alt_count` the default value zero is taken. As abbreviations for the temporal operators, `alt_count[x, d]` is used. Further, `s[x..y, d, ∘]` for `x` $<$ `y` abbreviates `s[x,d]` $\circ$ `s[x+1,d]` $\circ$ `...` $\circ$ `s[y,d]` where $\circ$ is a binary operator. Considering Example 3.2.1, a trigger based on the count, e.g., `trigger alt_count > 10`, is preferable if only persistent violations should be reported.

In general, a Lola specification describes an equation system, whose semantics is defined in terms of finding a corresponding *evaluation model*.

Definition 3.5 (Lola Semantics [130])

Let φ be a Lola specification over independent stream variables $t_1, \ldots, t_m$ and dependent stream variables $s_1, \ldots, s_n$. Given a trace σ with $|\sigma| = N$, the sequence of tuples $\langle \tau_1, \ldots, \tau_n \rangle$ of streams of length N is called an *evaluation model*, if each equation $s := expr_i(t_1, \ldots, t_m, s_1, \ldots, s_n)$, $\langle \tau_1, \ldots, \tau_n \rangle$ in φ satisfies the following conditions:

$\tau_i(j) = val(e_i)(j)$ for $0 < j \leq N+1$ where $val(e)(j)$ is defined inductively as follow:

Base cases:	Inductive cases:
$val(c)(j) = \mathtt{c}$	$val(f(e_1,\dots,e_n)(j)) = \mathtt{f}(val(e_1)(j),\dots,val(e_n)(j))$
$val(t_i)(j) = \sigma_i(j)$	$val(ite(e_b,e_1,e_2)(j)) = \mathtt{ite}(val(e_b)(j), val(e_1)(j), val(e_2)(j))$
$val(s_i)(j) = \tau_i(j)$	$val(e[k,c])(j) = \begin{cases} val(e)(j+k) & \text{if } 0 \leq j+k \leq N+1, \\ c & \text{otherwise} \end{cases}$

In general, Lola enables the specification of complex temporal properties in a way that is both precise and concise, reducing the likelihood of errors compared to hand-crafting monitoring code. The focus is on *what* properties should be monitored, rather than *how* the monitor should be executed. Following this principle, Lola provides automatic synthesis of monitors that infer and optimize implementation details such as evaluation order of streams and memory management. The evaluation order of Lola streams is derived by analyzing the specification's *dependency graph* [105]. A dependency graph captures the temporal relations among stream. For instance, `altitude_bound` depends on the current value of `altitude`, resulting in an edge with weight zero from `altitude_bound` to `altitude`. In contrast, `alt_count` has a self-loop with weight -1 as it depends on its previous value. Automatically deriving the evaluation order allows users to exploit the advantage of Lola's modular structure of output streams, e.g.,:

```
output alt_avg := alt_count / position
output alt_count := if altitude ≤ 200.0 then 0 else alt_count[-1, 0] + 1
output pos := pos.offset(by: -1, or: 0) + 1
```

where `pos` and `alt_count` are used before their definition. Further, the dependency graph is used to guarantee *well-formed* specifications by detecting invalid cyclic stream dependencies that do not have an unique evaluation model, e.g., `output a: Int32 := a` has more than one evaluation model while `output a :bool := ¬ a` has none.

So far, streams are considered synchronous. We now define event-based and periodic output streams, thereby introducing RTLola [125].

Definition 3.6 (RTLola)

The basic concept of translating input to output streams is extended by distinguishing each output stream $s \in s_1,\dots,s_n$ into one of two categories: event-based stream s_e and periodic stream s_p. These are defined as follows:

$$\begin{array}{ll} \textbf{output } s_e \ @\ bool(t_1,\dots,t_m) & : T_e := expr_e(t_1,\dots,t_m,s_1,\dots,s_n) \\ \textbf{output } s_p \ @\ \mathcal{P} & : T_p := expr_p(t_1,\dots,t_m,s_1,\dots,s_n) \end{array}$$

where @ indicates that an event-based stream s_e updates whenever the Boolean activation expression *bool* that follows becomes *true*, whereas a periodic stream s_p updates at regular time intervals defined by its period $\mathcal{P}$.

Intuitively, RTLola supports asynchronously arriving inputs, which can be synchronized by periodic streams. These synchronized values can then be used by event-based streams, similarly to how it was done in Lola. To handle asynchronicity, we restrict ourselves to the basic expressions:

- an expression $v.aggregate(over : \delta,\ using : \gamma).defaults(to : d)$ of type T, where v is an input or an periodic output stream of type T, δ is of type $\mathcal{P}$, γ is an aggregation function [126], and d is a value of type T;
- an expression $v.hold(or : d)$ of type T, where v is an input or output stream of type T and d is a default value of type T.

As before, this syntactic definition alone is insufficient, as it ensures well-typedness but not well-formedness. RTLola addresses this by introducing a type system that not only checks stream dependencies but also verifies compatible pacing between streams [132]. The specification

```
input a: Float32
input b: Float32
output c @(a & b) := a + b
output d @(a | b) := a + b
```

is *not* well-formed but well-typed. While the activation expression of stream `c` guarantees that both `a` and `b` have new values (`@(a & b)`), this is not the case for stream `d`, which may be activated by either one of them individually (`@(a | b)`). In such a case, this becomes problematic as it is unclear which value to take for the missing input. Here, the type system can be satisfied by using the *hold*-expression: `output d @(a | b):= a.hold(or: 0.0)+ b.hold(or:0.0)`, which acts in a sample-and-hold fashion, providing either the last value or the default value. The following example provides a well-formed RTLola specification that uses a periodic and an event-based stream.

Example 3.2.2. We consider an RTLola specification that extends the previous altitude bound check from Example 3.2.1:

```
input altitude: Float32 // in m                                                  1
output altitude_bound @altitude := altitude > 200.0                              2
                                   or altitude_avg.hold(or: 0.0) > 180.0         3
output altitude_avg @1Hz := altitude.aggregate(over: 2s, using: avg).defaults(to: 0.0)   4
trigger altitude_bound "Warning: Decrease altitude!"                             5
```

Lines 1, 2, and 5 reproduce the specification in Example 3.2.1, triggering a warning whenever the altitude exceeds 200 meters. Line 3 extends this check by ensuring that the most recently computed average altitude stays below 180 meters, thereby providing early detection of a potential 200 meters threshold violation. The average altitude is calculated in Line 4 using RTLola's aggregation expression, which computes, once per second (`1Hz`), the average of all altitude values received over the previous two seconds. △

Remark 3.2 (Robustness of Monitoring Functions)

ARP6539 [133] provides guidance to validate and verify the need for, and the robustness of monitoring functions in complex flight control and related systems. The standard emphasizes the importance of developing monitors that are sufficiently *robust* to handle operational nuisances, such as signals temporarily exceeding their thresholds without compromising safety. This approach ensures that the monitoring system remains effective under various operating conditions, thereby enhancing the reliability and safety. An example is GPS jumps, i.e., sudden, erroneous position changes due to sensor glitches. A robust monitor should tolerate such transient anomalies, if they are short-lived and do not represent a safety hazard, instead of immediately triggering a contingency or emergency action.

One reason for GPS jumps is a poor satellite constellation where only a few satellites are in range. Using its real-time features, RTLola is well-suited to express such properties:

```
input num_sat: UInt8 // Number of satellites
trigger @1Hz (num_sat.aggregate(over:1s,using:avg).defaults(to:0))<7 "Few satellites"
```

where a trigger checks each second if the one-second average of `num_sat` is less than seven, indicating that too few satellites are in range.

Next, we define *parameterized* RTLola which allows to carry input data along the stream. A parameterized output stream can be viewed as a template that is instantiated when its *spawn* condition is met, evaluated when its *eval* condition holds, and closed once its *close*-condition is satisfied. After being spawned, each template instance starts a *local clock* that it can access.

Definition 3.7 (Parametrized RTLola)

Given output streams $s_1, \dots, s_n$ and input streams $t_1, \dots, t_m$, parameterized output streams are defined as follows:

$$\textbf{output}\; s(n_1 : T_1, \dots, n_k : T_k) : T$$
$$\textbf{spawn when}\; bool(t_1, \dots, t_m, s_1, \dots, s_n)\; \textbf{with}\; p_1, \dots, p_n$$
$$\textbf{eval when}\; @cnd\; \textbf{with}\; expr(t_1, \dots, t_m, s_1, \dots, s_n, n_1, \dots, n_k) \quad \textbf{close}\; @cnd$$

where '*cnd*' is either a Boolean expression or a period $\mathcal{P}$ relative to the local clock *Local* or the global clock *Global*. Each spawned instance is identified by parameter values $n_1, \dots, n_k$, which are assigned values from the existing streams $p_1, \dots, p_n$ at the time of spawning. These parameters are used to access individual spawned instances as $s(n_1, \ldots, n_k)$ and can be used for the *expr* of type T. The entire family of spawned instances can be aggregated using $s.aggregate(over\_instances : all, using : \gamma)$ where γ is an aggregation function.

While we omit details on ensuring well-formed parameterized RTLOLA specifications here, an example helps illustrate the use of parameterized output streams. Details on the extended type system are provided in [132]. Consider the specification

```
input user_id: UInt32                                                           1
input time_online: Float32                                                      2
output total_time_online (id)                                                   3
  spawn when user_id > 10 with user_id                                          4
  eval when user_id = id with total_time_online(id)[-1, 0.0] + time_online      5
  close @Local(1h)                                                              6
```

where the parameterized stream `total_time_online` tracks the cumulative online time for each user with an `user_id > 10`. A new instance is spawned for each such user when the condition holds (Line 4). The instance is updated by the current input `time_online` whenever the current `user_id` matches its `id` (Line 5), i.e., the same user is active. Each instance is closed after one hour has passed relative to its creation.

3.3. Temporal Behavior Tree

Temporal Behavior Trees retrofit monitoring capabilities to Behavior Trees. Behavior trees are increasingly popular in robotic applications. They were originally used in computer games to animate complex sequences of actions for characters. A behavior tree is a programmatic specification of a plan or sequence of possible actions by an autonomous agent. It includes operators that enable us to specify a sequence of sub-plans executed one after the other; falling back to a different sub-plan if the current sub-plan encounters an unexpected failure; repeating a sub-plan multiple times; and conducting many sub-plans in parallel until the number of the sub-plans that have succeeded exceeds a specified lower limit [134]. Figure 3.1 depicts such a behavior tree for an autonomous landing of a UAV on a ship deck. It specifies a sequence of operations: (1) move the UAV into position relative to the ship; (2) maintain this position for sometime; (3) move above the touchdown point; and (4) descend onto the ship.

A TBT borrows the structural operators of BTs, but augments leaf nodes with monitoring properties, here restricted to STL formulas for clarity. We will now define the notion of a TBT.

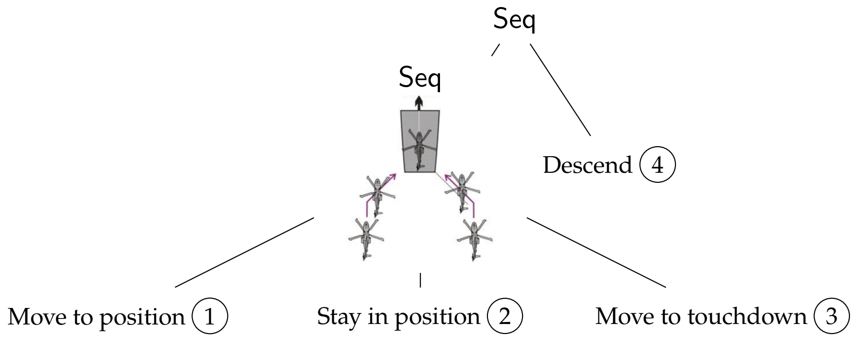


Figure 3.1.: A BT that executes a landing maneuver on a ship deck. It uses two *Sequence* operators to execute its children from left to right. The figure under the second Seq does not belong to the BT, it only illustrates the role of the child nodes, which is to guide the UAV above the touchdown point on the ship.

Definition 3.8 (Syntax of Temporal Behavior Trees)

→ Def. 3.1, p. 36 Let φ be an STL formula as described by the grammar in Definition 3.1. We construct a temporal behavior tree using the following syntax:

$$
\begin{array}{llll}
\mathcal{T} & := \mathsf{Leaf}(\varphi), & & \leftarrow \text{Leaf node} \\
 & \mid \mathsf{Fback}([\mathcal{T},\ldots,\mathcal{T}]), & & \leftarrow \text{Fallback node} \\
 & \mid \mathsf{Par}_M([\mathcal{T},\ldots,\mathcal{T}]), & M \in \mathbb{N} & \leftarrow \text{Parallel node} \\
 & \mid \mathsf{Seq}([\mathcal{T},\cdots,\mathcal{T}]), & & \leftarrow \text{Sequence node} \\
 & \mid \mathsf{Tout}_t(\mathcal{T}), & t \in \mathbb{N} & \leftarrow \text{Timeout node} \\
 & \mid \bigstar_n(\mathcal{T}), & n \in \mathbb{N} \cup \{\infty\} & \leftarrow \text{Repeat node}
\end{array}
$$

Informally, an STL formula at a leaf node specifies that the trace must satisfy the formula. Likewise, $\mathsf{Fback}([\mathcal{T}_1,\ldots,\mathcal{T}_k])$ mimics the semantics of a "fallback" node in a behavior tree: at least one of the subtrees $\mathcal{T}_1,\ldots,\mathcal{T}_k$ must eventually be satisfied by the trace. $\mathsf{Par}_M([\mathcal{T}_1,\ldots,\mathcal{T}_k])$ denotes a parallel operator that specifies that at least M distinct subtrees must be satisfied simultaneously by the trace σ. $\mathsf{Seq}([\mathcal{T}_1,\cdots,\mathcal{T}_k])$ is a sequential node that denotes that σ must be partitioned into k parts (some of which may be empty) $\sigma_1;\sigma_2;\sigma_3;\cdots;\sigma_k$ such that each part $\sigma_i \models \mathcal{T}_i$. $\mathsf{Tout}_t(\mathcal{T})$ is a timeout node denoting the subtree $\mathcal{T}$ must be satisfied by a prefix of the trace of size t. $\bigstar_n(\mathcal{T})$ is a repeat operator node denoting σ must be partitioned into $k \leq n$ parts $\sigma_1;\sigma_2;\sigma_3;\cdots;\sigma_k$ such that $\sigma_i \models \mathcal{T}$.

The presence of the Seq and $\bigstar$ operators makes temporal behavior trees strictly more expressive than STL (see Theorem 3). We write $\sigma \models \mathcal{T}$ to denote that the finite trace σ satisfies the TBT specification $\mathcal{T}$.

Definition 3.9 (Boolean Semantics of TBTs)

The Boolean semantics of TBTs is defined as follows:

$$
\begin{aligned}
\sigma &\models \mathsf{Leaf}(\varphi) && \iff && \sigma \models \varphi \text{ (see Definition 3.2)}\\
\sigma &\models \mathsf{Fback}([\mathcal{T}_1, \ldots, \mathcal{T}_n]) && \iff && \exists j \in [1,n], \exists i \in [0, |\sigma|-1], \sigma[i:] \models \mathcal{T}_j\\
\sigma &\models \mathsf{Par}_M([\mathcal{T}_1, \ldots, \mathcal{T}_n]) && \iff && \exists I \subseteq [1,n], |I| \geq M \wedge \forall i \in I, \sigma \models \mathcal{T}_i\\
\sigma &\models \mathsf{Seq}([\mathcal{T}_1, \mathcal{T}_2]) && \iff && \exists i \in [0, |\sigma|-1], \sigma[:i] \models T_1 \wedge \sigma[i+1:] \models T_2\\
\sigma &\models \mathsf{Tout}_t(\mathcal{T}) && \iff && |\sigma| \leq t \ \wedge \ \exists t'. \, 0 \leq t' < |\sigma| \ \wedge \ \sigma[:t'] \models \mathcal{T}\\
\sigma &\models \bigstar_n(\mathcal{T}) && \iff && \exists\, k < n \text{ indices } i_1 \leq i_2 \leq \cdots \leq i_k \text{ with } \sigma[:i_1] \models \mathcal{T},\\
&&&&& \sigma[i_1+1:i_2] \models \mathcal{T}, \cdots, \sigma[i_k+1:] \models \mathcal{T}
\end{aligned}
$$

Note that the formula $\mathsf{Leaf}(\neg \bigcirc\, true)$ is *true* only in the last state of a finite trace. We abbreviate this formula by $\bullet$. Also, for brevity, we occasionally omit $\mathsf{Leaf}(\varphi)$ and write φ. We now provide some simple examples of TBTs to show how the modularity of TBT helps in practice.

Example 3.3.1. Let $\mathcal{T}$ be a TBT that adds formulas to the leaf nodes of the BT depicted in Figure 3.1. We can add requirements "on top of" a tree, e.g., $\mathsf{Par}_2([\mathsf{Leaf}(\Box \neg obstacle), \mathcal{T}]$ to specify that obstacles are avoided during landing or $\mathsf{Fback}([\mathsf{Tout}_t(\mathcal{T}), \mathsf{Leaf}(contingency)])$ to state that we need to land within t time units or otherwise a contingency needs to be activated. Further, we can add new nodes within $\mathcal{T}$, e.g., we can replace the leaf node 4, referred to as L, with $\mathsf{Seq}([\mathsf{Leaf}(\Box_{[0,t]} aboveTP), L])$ to make sure that we also remain above the touchdown point for a period of time t before descending. △

Remark 3.3 (Original Tout_t Semantics)

In [49], an alternative semantics for timeout was given:

$$\sigma \models \mathsf{Tout}_t(\mathcal{T}) \iff \sigma[:t-1] \models \mathcal{T} \text{ if } t \leq |\sigma| \text{ or } \sigma \models \mathcal{T} \text{ if } t > |\sigma.$$

Consider the specification $\mathsf{Seq}([\mathsf{Tout}_{10}(\Diamond\, a), \mathsf{Tout}_{10}(\Diamond\, b)])$, which requires that a holds within 10 steps, and afterwards b also holds on its segment within 10 steps. Now take a trace σ of length 30, where a holds at Position 9 and b holds at Position 25. Intuitively, this trace should *not* satisfy the specification because in the worst case b must hold within 20 steps, which it does not.

However, given the original semantics, the trace satisfies the specification. The issue arises because Seq may select the "chop" point at Index 20, assigning a right segment that satisfies $\mathsf{Tout}_{10}(\Diamond\, b)$. The left segment is also "incorrectly" satisfied, since the original semantics does not restrict the suffix length of σ, which allows part of the subsequent trace to be "hidden". In this example, the left segment has length 20, so $t \leq |\sigma|$, and within its first 10 steps $\Diamond\, a$ holds.

By contrast, the revised semantics rejects this trace. Although it still allows hiding a portion of the suffix trace when t is sufficiently large, it prevents the problematic case from before. The reason is that no valid "chop" can divide the trace of length 30 into two segments where both time limits of length 10 can be met, i.e., $|\sigma| \leq 10$.

Let us denote $max_M([i_1, \ldots, i_n])$ as the function that outputs the M^{th} largest number in the list $[i_1, \ldots, i_n]$ if $M \leq n$, and $-\infty$ if $M > n$. Also, for simplicity, in the following we rewrite $\mathsf{Seq}([\mathcal{T}_1, \ldots, \mathcal{T}_k])$ as a sequence $\mathsf{Seq}([\mathcal{T}_1, (\mathsf{Seq}([\mathcal{T}_2, \cdots, \mathsf{Seq}([\mathcal{T}_{k-1}, \mathcal{T}_k]))))$ for $k \geq 2$. Note that $\mathsf{Seq}([\mathcal{T}])$ is the same as $\mathcal{T}$.

Definition 3.10 (Robustness Semantics of TBTs)

The robustness of a temporal behavior tree $\mathcal{T}$ on a finite execution trace σ, denoted $\rho(\mathcal{T}, \sigma)$, is defined as follows:

$$\begin{aligned}
\rho(\mathsf{Leaf}(\varphi),\ \sigma) &= \rho(\varphi, \sigma) \\
\rho(\mathsf{Fback}([\mathcal{T}_1, \ldots, \mathcal{T}_k])),\ \sigma) &= \max_{j \in [1,k]} \max_{i \in [0,|\sigma|-1]} \rho(\mathcal{T}_j, \sigma[i:]) \\
\rho(\mathsf{Par}_M([\mathcal{T}_1, \ldots, \mathcal{T}_k]), \sigma) &= max_M\Big(\ \rho(\mathcal{T}_1, \sigma),\ \cdots,\ \rho(\mathcal{T}_k, \sigma)\ \Big) \\
\rho(\mathsf{Seq}([\mathcal{T}_1, \mathcal{T}_2]), \sigma) &= \max_{u \in [0,|\sigma|-1]} \min(\rho(\mathcal{T}_1, \sigma[:u]), \rho(\mathcal{T}_2, \sigma[u+1:])) \\
\rho(\mathsf{Tout}_t(\mathcal{T}), \sigma) &= \begin{cases} \max_{t' \in [0,|\sigma|-1]} \rho(\mathcal{T}, \sigma[:t']) & \text{, if } |\sigma| \leq t \\ -\infty & \text{, otherwise.} \end{cases} \\
\rho(\bigstar_n(\mathcal{T}), \sigma) &= \begin{cases} \rho(\mathsf{Seq}([\mathcal{T}, \bigstar_{n-1}(\mathcal{T})]), \sigma) & \text{, if } |\sigma| > 0 \wedge n > 0 \\ \infty & \text{, otherwise.} \end{cases}
\end{aligned}$$

→ Def. 3.3, p. 37

As in the case of STL formulas (see Definition 3.3), Definition 3.10 evaluates a given TBT for a trace into a real number, which denotes the "distance to satisfaction" of the specification by the trace. We now prove that the robustness ρ for a TBT corresponds to the notion of satisfaction defined above in the following manner.

Theorem 2. *For any trace σ and TBT $\mathcal{T}$, $\sigma \models \mathcal{T}$ iff $\rho(\mathcal{T}, \sigma) \geq 0$.*

Proof. The full proof of the theorem by structural induction on $\mathcal{T}$ is shown in Section A.2.2. □

→ Sec. A.2.2, p. 165

STL and TBT are not equivalent in terms of expressiveness. Whereas any property in finite trace STL can be trivially expressed as a TBT with a single leaf node, the converse does not hold.

Theorem 3. *There exists properties of finite traces specified using TBT that cannot be written as a finite trace STL formula.*

Proof. It is well-known that temporal logics based on the $\Diamond$,$\Box$,$\mathcal{U}$, and $\bigcirc$ temporal operators cannot express the property "p is *true* in all odd indexed states of a sequence", but TBT can. Pierre Wolper [135] proves this for LTL over infinite traces. Wolper's proof carries over to the version of STL used in this thesis. Furthermore, the STL formulas subscripted by finite intervals such as $\Box_{[l,u]}\varphi$ can be written systematically using finitely many nestings of the next operator $\bigcirc$ and the unbounded $\Box$ operator. Similar considerations apply to formulas of the form $\Diamond_{[l,u]}(\varphi)$ and $\varphi_1\mathcal{U}_{[l,u]}\varphi_2$. Further, the proof extends to finite trace semantics as it only requires a finite prefix p^i that is sufficiently large, i.e., number next operators $l < i$.

However, the *even* property can be expressed as

$$\bigstar_\infty(\mathsf{Seq}([\mathsf{Tout}_1(\mathsf{Leaf}(p)), \mathsf{Tout}_1(\mathsf{Leaf}(\mathit{true}))]))\,.$$

The $\bigstar$-operator repeats the chopping of each Tout operator until of the end of the trace. The first Tout handles odd positions and the second Tout checks that p holds at even positions. $\Box$

Remark 3.4 (Extending TBTs to Other Specification Languages in Leaf Nodes)

For clarity, we introduced TBTs using STL formulas in the leaf nodes. To support other logics, an arbitrary logic φ must provide a Boolean semantics ($\sigma \models \varphi$) as well as a robust semantics ($\rho(\varphi, \sigma)$).

Extending TBTs to stream-based languages is different, since these produce multiple output values that can be propagated up the tree. We handle this in a complementary way: a stream language "pre-processes" a trace, and its output values extend the set of atomic propositions available in the leaf nodes, either in combination with or independently from a logic language. Note that this approach can introduce redundancy. For example, a stream computing `output o := o[-1,true]` $\wedge$ `i` for Boolean input `i`, when combined with $\Box$ `o` in a leaf node, is equivalent to using stream `o` directly in that leaf node. To handle asynchronicity in streams, we use a sample-and-hold approach, i.e., using the most recent available value, or a default value if none has been produced yet.

3.4. Summary

This chapter introduced three specification languages for describing and monitoring temporal properties of autonomous systems.

Signal Temporal Logic provides a logic to express temporal requirements over traces with both Boolean and Robust semantics. The latter semantic allows to quantify the degree to which a property is satisfied or violated. STL abstracts away the underlying computations of atomic propositions, focusing purely on their logical and temporal relationships.

In contrast, RTLOLA uses a stream-based approach that allows these APs to be defined directly through synchronous and asynchronous stream computations that aggregate input streams from the system or other output streams in the language. This enables a more modular and data-driven monitoring.

Finally, Temporal Behavior Trees extend classical behavior trees by embedding monitoring properties in their leaf nodes while retaining standard control operators such as sequence, fallback, parallel, timeout, and repeat. Within TBTs, logics like STL can be used to specify properties at the leaves, while stream-based languages like RTLOLA enrich the set of APs available within the tree. This combination unifies both approaches, resulting in a modular and expressive specification language that can be seamlessly retrofitted to existing behavior trees.

Remark 3.5 (Monitoring Temporal Behavior Trees)

→ Part II, p. 51

In Part II, we present novel offline algorithms, while leaving the online monitoring problem open for TBTs. The main challenge in online monitoring TBTs is deciding when a monitor should "chop" the trace and proceed to the next node. For example, consider the TBT $\mathsf{Seq}([\Diamond\, a, \Box\, b])$. It is unclear when the monitor should transition from the leaf node $\Diamond\, a$ to $\Box\, b$ during execution. After observing a, there is no immediate need to move on. Even if the monitor advances to $\Box\, b$ when b initially holds, it may later need to backtrack if b becomes *false*, since remaining in $\Diamond\, a$ would still be consistent with the tree. This possibility of backtracking implies that all potential "chops" must be considered in the worst-case. In general, it is not obvious what constitutes a *good* monitorable TBT fragment.

→ Rem. 2.4, p. 24

For qualitative monitoring using Boolean or multi-valued semantics (see Remark 2.4), monitoring TBTs can be performed by translating the leaf nodes into corresponding automata [124] and connecting these automata by nondeterministic transition, similar to the McNaughton-Yamada-Thompson construction for converting regular expressions into nondeterministic finite automata [136].

For quantitative monitoring using TBT's robust semantics, monitoring is harder as relevant numerical values for possible chops most be stored. We identify two simplifying assumptions for online monitoring of TBTs. First, the controller executing the behavior tree provides the set of currently active nodes. Then, monitoring reduces to monitoring a set of monitors for different phases that can be activated and deactivated. For instances, the following specification uses RTLOLA's parameterized streams:

```
input leaf_active: Set<Leaf>
output time(leaf)
   spawn when Leaf(<>total_time_online) ∈ leaf_active
      with (leaf)
      eval when @1Hz with time(leaf)[-1, 0.0] + 1.0
      close when Leaf(<>total_time_online) ∉ leaf_active
```

The specification spawns a new instance whenever a new leaf node becomes active and closes it once the leaf is deactivated. While this examples simply computes the active duration of a leaf, it can be extended to more general leaf node computations.

Second, when sufficient for the application, a greedy monitoring algorithm can be employed where the monitor simply advances in the tree once a leaf node is satisfied. Returning to the TBT $\mathsf{Seq}([\Diamond\, a, \Box\, b])$, a greedy monitor moves from $\Diamond\, a$ to $\Box\, b$ upon the first occurrence of a. This approach can be realized via an automaton construction for each leaf [124]. Although such monitors are inflexible, as they do not support backtracking, they achieve good results in reinforcement learning [137].

Part II

Offline Monitoring: Algorithms & Applications

Hot Take:
Offline monitoring currently impacts autonomous CPS more than online monitoring.
(2025)

Chapter 4

Trace Segmentation

In this chapter, we solve the *segmentation problem*. A *segmentation* assigns portions of a trace, i.e., $\sigma[i : j]$, to nodes of a TBT specification. The segmentation problem aims to find the segmentation that maximizes the TBT's robustness. In Figure 4.1, a BT for a landing maneuver of a UAV on a ship is given and the corresponding segmentation is depicted in Figure 4.2. Assume that the robustness of segment (1) is positive, whereas that for segment (2) is negative. Therefore, we conclude that the UAV moved to the target position as required, but then failed to stay in position. Such a segmentation is very useful in our analysis of the trace: for a trace that violates a specification, it shows the specific portions of the trace that correspond to violations of sub-parts of the overall specification, potentially leading us to diagnostic explanations. It also allows us to rank these failures in terms of the robustness from worst to best. Similarly, for a satisfying trace, we can examine which parts of the trace came closest to violating the

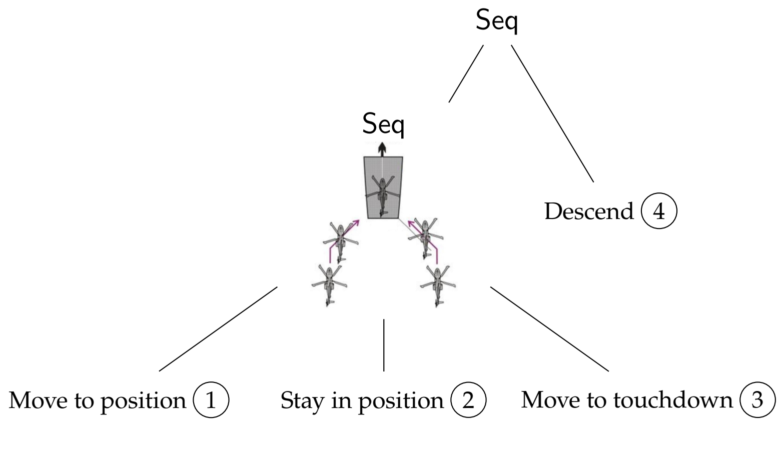


Figure 4.1.: A behavior tree that implements a maneuver for landing on a ship deck. It uses two *Sequence* operators to *execute* its children from left to right.

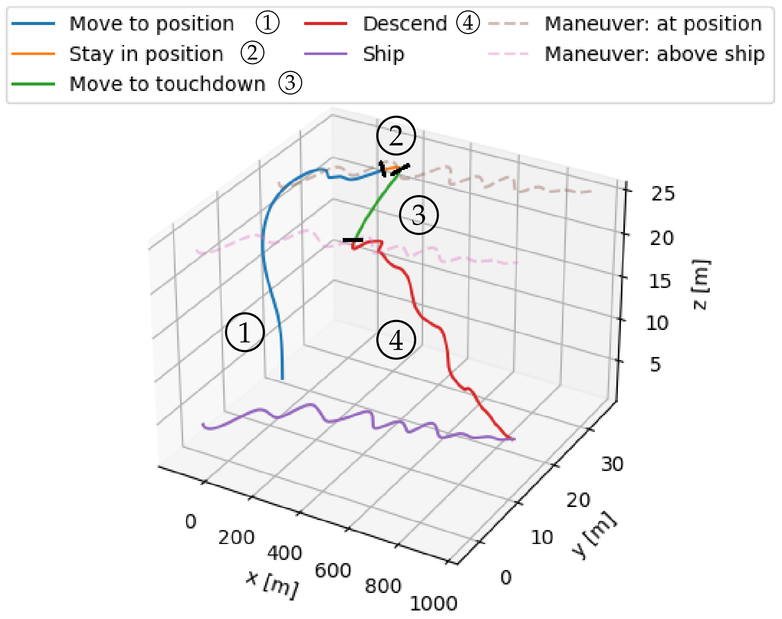


Figure 4.2.: Segmentation result of a UAV 45-Degree ship deck landing maneuver. In (1) the UAV starts with moving in position, then (2) it stays for the required amount of time there, before (3) moving above the touchdown, and (4) finally descending. The trajectory of the ship is the purple curve at the bottom with $z = 0$. Dotted lines represents best positions relative to the ship, i.e., left diagonally behind the ship and above touchdown. The various segments are shown in different colors with numbers alongside. The segments are automatically computed by the algorithm described in this chapter.

corresponding parts of the specifications. Although the exact algorithm for computing robustness and segmentation is at most cubic time in the size of the trace and linear in the size of the tree, we find that in practice this is forbiddingly expensive for long traces with hundreds or thousands of samples per second. We present two systematic approaches to speeding up the computations by orders of magnitude: (a) sub-sampling of the trace using a carefully calculated stride length that preserves the Boolean semantics but approximates the robustness; and (b) a lazy evaluation scheme to rapidly compute an approximate answer.

We show that segmentation is very useful in CPS applications through two empirical case studies based on data from realistic applications. In one study, we examine a detailed TBT specification of a UAV landing on a ship through one of four possible maneuvers, each involving multiple stages to ensure a safe and robust landing even in the presence of disturbances. We also present the use of TBTs and segmentation to analyze human operator performance in flying a UAV inside a desktop-based simulation environment. By specifying the task as a TBT, we demonstrate how segmentation reveals surprising patterns in the data that were not apparent through a whole trace analysis.

The remainder of this chapter is structured as follows. Section 4.1 introduces segmentation and presents alternative segmentation and approximation techniques. Finally, Section 4.3 demonstrates experimental results. →Sec. 4.3, p. 63

4.1. Computing Robustness

We will now provide a dynamic programming approach to compute the robustness efficiently. This will be used to define the notion of trace segmentation in the next section. For convenience, let σ be the given original trace.

Definition 4.1 (Dynamic Programming for TBT Robustness)

The robustness of a TBT $\mathcal{T}$ on a finite execution trace σ can be implemented in a dynamic programming fashion by computing $\rho_\sigma(\mathcal{T}, i, j) = \rho(\mathcal{T}, \sigma[i:j])$ as follows:

$$\begin{aligned}
\rho_\sigma(\mathsf{Leaf}(\varphi), i, j) &= \rho(\varphi, \sigma[i:j]) \text{ (Cf. Definition 3.3)},\\
\rho_\sigma(\mathsf{Fback}([\mathcal{T}_1, \dots, \mathcal{T}_k]), i, j) &= \max_{l\in[1,k]} \max_{i'\in[i,j]} \left(\rho_\sigma(\mathcal{T}_l, i', j)\right),\\
\rho_\sigma(\mathsf{Par}_M([\mathcal{T}_1, \dots, \mathcal{T}_k]), i, j) &= \max{}_M(\rho_\sigma(\mathcal{T}_1, i, j), \dots, \rho_\sigma(\mathcal{T}_k, i, j)),\\
\rho_\sigma(\mathsf{Seq}([\mathcal{T}_1, \mathcal{T}_2]), i, j) &= \max_{u\in[i,j]} \min(\rho_\sigma(\mathcal{T}_1, i, u), \rho_\sigma(\mathcal{T}_2, u+1, j))\\
\rho_\sigma(\mathsf{Tout}_t(\mathcal{T}), i, j) &= \begin{cases} \max_{t'\in[i,j]} \rho_\sigma(\mathcal{T}, i, t') & \text{, if } j < i + t\\ -\infty & \text{, otherwise.}\end{cases}\\
\rho_\sigma(\bigstar_n(\mathcal{T}), i, j) &= \begin{cases} \rho_\sigma(\mathsf{Seq}([\mathcal{T}, \bigstar_{n-1}(\mathcal{T})]), i, j) & \text{, if } n > 0 \text{ and } i \leq j\\ \infty & \text{, if } n = 0 \text{ or } i > j\end{cases}
\end{aligned}$$

The computed robustness values that are stored in a memo table T can be accessed by $\mathrm{T}(\mathcal{T}, i, j)$ where $\mathcal{T}$ is a (sub-)TBT, i is the beginning of the segment, and j its end. The size of the table is $O(N^2|\mathcal{T}|)$ given a trace of size $|\sigma| = N$. However, the time taken to fill out each entry in the memo table can be $O(N)$ in the worst case (see the definition of the Seq node). It also depends on the time taken to compute the robustness of the temporal logic formulas at the leaf, which we assume will take time $O(N^2|\varphi|)$, for a leaf node $\mathsf{Leaf}(\varphi)$, wherein $|\varphi|$ denotes the size of the formula. Assuming that $|\mathcal{T}|$ includes the size of the temporal logic formulas at the leaves, as well, we can bound the execution time as $O(N^3|\mathcal{T}|)$.

Lemma 4. *For any trace σ and TBT $\mathcal{T}$, the value of $\rho_\sigma(\mathcal{T}, 0, |\sigma| - 1)$ from Definition 4.1 is the same as $\rho(\mathcal{T}, \sigma)$ from Definition 3.10.* → Def. 3.10, p. 46

Proof. We prove by induction on the structure of the tree, that for any two indices i, j, we have $\rho_\sigma(\mathcal{T}, i, j) = \rho(\mathcal{T}, \sigma[i : j])$. Base case is for leaf formulas and formulas of the form $\bigstar_n(\mathcal{T})$ for $n = 0$ or $i > j$. In all cases, a comparison of the cases in Definition 3.10 (→ Def. 3.10, p. 46) and Definition 4.1 shows that they yield the same values. The proof compares each node type to show that if ρ_σ and ρ agree on the children of a node, they also agree on the node itself. Comparing Definition 3.10 (→ Def. 3.10, p. 46) and Definition 4.1 shows that they yield the same values. □

4.2. Segmenting Traces

In this section, we define the problem of segmenting traces with respect to a TBT specification. We show the connection between segmentation and the robustness semantics as in Definition 3.10 (→ Def. 3.10, p. 46). The dynamic programming formulation that computes robustness also computes the segmentation of a trace. We briefly describe how the segmentation can be useful, with further demonstrations of usefulness provided in Section 4.3 (→ Sec. 4.3, p. 63).

We begin by defining the segmentation of a trace with respect to a TBT $\mathcal{T}$. Informally, given a TBT $\mathcal{T}$, the segmentation of a trace σ splits it into multiple subtraces of the form $\sigma[i : j]$ such that (a) every subtree of $\mathcal{T}$ is associated with a subtrace; and (b) the satisfaction and robustness of the specification $\mathcal{T}$ by the trace σ can be linked to the satisfaction and robustness of each of the subtraces $\sigma[i : j]$ associated with the corresponding subtree $\hat{\mathcal{T}}$. We recall the definition of robustness $\rho(\mathcal{T}, \sigma)$ from Definition 4.1.

Definition 4.2 (Segmentation of a TBT)

The segmentation of a trace σ with respect to a TBT $\mathcal{T}$ is a graph $G = (V, E)$ whose vertex set V consists of triples of the form

$$V = \{(\hat{\mathcal{T}}, i, j) \mid \hat{\mathcal{T}} \text{ is a subtree of } \mathcal{T},\ 0 \leq i, j \leq |\sigma| - 1\},$$

and edges $E \subseteq V \times V$ such that the following conditions hold:

1. $(\mathcal{T}, 0, |\sigma| - 1) \in V$ corresponding to the entire tree $\mathcal{T}$ and the entire trace from indices 1 to $|\sigma|$.

2. If a node v is of the form $(\mathsf{Fback}([\mathcal{T}_1, \ldots, \mathcal{T}_k]), i, j) \in V$, and $i \leq j$ then there is precisely one subtree index $l \in [1, k]$ and a single trace index $i' \in [i, j]$ such that the edge $v \to (\mathcal{T}_l, i', j) \in E$.

3. If a node v of the form $(\mathsf{Seq}([\mathcal{T}_1, \mathcal{T}_2]), i, j) \in V$, there exists a unique index u such that $(\mathcal{T}_1, i, u) \in V$, $(\mathcal{T}_2, u + 1, j) \in V$ and the edges $v \to (\mathcal{T}_1, i, u)$ and $v \to (\mathcal{T}_2, u + 1, j)$ belong to E.

4. If a node v of the form $(\mathsf{Par}_M([\mathcal{T}_1, \ldots, \mathcal{T}_k]), i, j) \in V$, we have precisely M distinct indices $l_1, \ldots, l_M \in [1, k]$ such that the set $S = \{(\mathcal{T}_{l_1}, i, j), \cdots, (\mathcal{T}_{l_M}, i, j)\} \subseteq V$ and edges from v to each of the nodes in S belong to E.

5. If a node v of the form $(\mathsf{Tout}_t(\mathcal{T}), i, j) \in V$ with $j + 1 - i \leq t$, then there exists an index $t' \in [i, j]$ such that $v' = (\mathcal{T}, i, t') \in V$ and $(v, v') \in E$.

6. If v is of the form $(\bigstar_n(\mathcal{T}), i, j) \in S$ and $n \geq 1$ and $i \leq j$, then either (a) the node $v' = (\mathcal{T}, i, j) \in V$ with an edge from v to v'; or (b) there exists indices $u_1 \leq \ldots \leq u_k$ for some $1 \leq k < n$ such that the set of nodes $S = \{(\mathcal{T}, i, u_1), (\mathcal{T}, u_1 + 1, u_2), \ldots, (\mathcal{T}, u_k + 1, j)\} \subseteq V$ have edges in E from v to each node in S.

7. The set of vertices V and edges E are minimal: i.e, no proper subsets of V, E satisfy the conditions stated above.

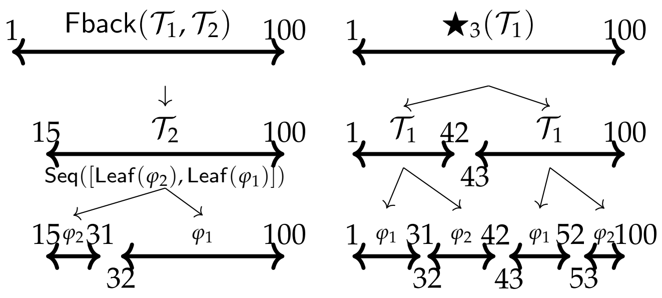
Note that the last condition of minimality is important to ensure that we do not add unneeded vertices and edges to a segmentation.

Lemma 5. *Any segmentation of a trace σ and TBT $\mathcal{T}$ is a directed acyclic graph. Furthermore, a node has no outgoing edges if and only if it is of the form (a) $(\mathsf{Leaf}(\varphi), i, j)$, (b) $(\mathsf{Tout}_t(\mathcal{T}), i, j)$ with $j + 1 - i > t$, (c) $(\bigstar_0(\mathcal{T}), i, j)$, or (d) $(\bigstar_n(\mathcal{T}), i, j)$ with $i > j$.*

Proof. Each edge $v : (\mathcal{T}, i, j) \to v' : (\mathcal{T}', i', j')$ in the set E from Definition 4.2 goes from a TBT $\mathcal{T}$ to its subtree $\mathcal{T}'$. Therefore, no cycles can exist in the graph G. The second part also follows from Definition 4.2. □

Example 4.2.1. Consider a trace σ of length 100 and a TBT specification $\mathcal{T}$ of the form $\mathsf{Fback}([\mathcal{T}_1, \mathcal{T}_2])$ where $\mathcal{T}_1$ is $\mathsf{Seq}([\mathsf{Leaf}(\varphi_1), \mathsf{Leaf}(\varphi_2)])$ and $\mathcal{T}_2$ is $\mathsf{Seq}([\mathsf{Leaf}(\varphi_2), \mathsf{Leaf}(\varphi_1)])$. The TBT expresses the fact that eventually φ_1 must be satisfied followed by φ_2, or the other way around. One possible segmentation (graph) is provided in Figure 4.3 (left).

Consider another TBT of the form $\bigstar_3(\mathsf{Seq}([\mathsf{Leaf}(\varphi_1), \mathsf{Leaf}(\varphi_2)]))$. An example segmentation is shown in Figure 4.3 (right). △



1 Fback($\mathcal{T}_1$, $\mathcal{T}_2$) 100
15 $\mathcal{T}_2$ 100
Seq([Leaf(φ_2), Leaf(φ_1)])
15 φ_2 31 32 φ_1 100
1 $\star_3(\mathcal{T}_1)$ 100
1 $\mathcal{T}_1$ 42 43 $\mathcal{T}_1$ 100
1 φ_1 31 32 φ_2 42 43 φ_1 52 53 φ_2 100

Figure 4.3.: Example segmentations for the two properties described in Example 4.2.1.

Given a segmentation graph $G = (V, E)$ for a TBT $\mathcal{T}$ and trace σ, it induces a value for the robustness of the specification and trace *given the segmentation*. Formally, we will define $\pi_G(\hat{\mathcal{T}}, i, j)$ for each $(\hat{\mathcal{T}}, i, j) \in V$ as follows:

→ Def. 3.3, p. 37

1. If $\hat{\mathcal{T}} = \mathsf{Leaf}(\varphi)$ then $\pi_G(\hat{\mathcal{T}}, i, j) = \rho(\varphi, \sigma[i : j])$ using Definition 3.3.
2. If $\hat{\mathcal{T}} = \star_0(\mathcal{T})$ or the node is of the form $(\star_n, i, j)$ with $i > j$, then $\pi_G(\hat{\mathcal{T}}, i, j) = \infty$.
3. Otherwise, $\pi_G(\hat{\mathcal{T}}, i, j) = \min(\{\pi_G(\mathcal{T}_1, i', j') \mid (\hat{\mathcal{T}}, i, j) \to (\mathcal{T}_1, i', j') \in E\})$, i.e., to the minimum robustness of its given existing successors in G.

Let $\mathsf{segs}(\mathcal{T}, \sigma)$ denote all possible segmentations of the trace σ against the TBT specification $\mathcal{T}$. We say that a segmentation G is optimal iff $\pi_G(\mathcal{T}, 0, |\sigma| - 1) = \max_{G' \in \mathsf{segs}(\mathcal{T},\sigma)} \pi_{G'}(\mathcal{T}, 0, |\sigma| - 1)$. In other words, the robustness calculated by the segmentation is maximal among all possible segmentations of the trace with respect to the specification. Informally, this means that the trace is segmented in the "best possible light" in an attempt to match it against the specification.

Theorem 6. *Let G be an optimal segmentation of trace σ w.r.t. TBT $\mathcal{T}$. It follows that $\pi_G(\mathcal{T}, 0, |\sigma| - 1) = \rho(\mathcal{T}, \sigma)$.*

→ Ex. 3.1.2, p. 37

Example 4.2.2. Consider once again the STL formula used in Example 3.1.2. The target position p is defined by the euclidean distance. Let $\mathsf{Seq}([\mathsf{Leaf}(\Diamond p), \mathsf{Leaf}(\Box_{[0,5]}\, p)])$ be the corresponding TBT for the formula. In this way, a segmentation allows us to identify which child was successful or not. For instance, given a segmentation that contains $(\mathsf{Leaf}(\Diamond p), 0, 99)$ and $(\mathsf{Leaf}(\Box_{[0,5]}, 100, 199)$ for a trace σ, where $\rho(\mathsf{Leaf}(\Diamond p), \sigma[: 99]))$ is positive and $\rho(\mathsf{Leaf}(\Box_{[0,5]}, \sigma[100 :]))$ is negative, we can conclude that the UAV reached position p but did not hold that position for as long as necessary. △

Computing an Optimal Segmentation We will now turn to the problem of computing an optimal segmentation given a trace σ and TBT $\mathcal{T}$. As it turns out, an optimal segmentation can be recovered through the dynamic programming table $\mathrm{T}(\hat{\mathcal{T}}, i, j)$ used to compute $\rho_\sigma(\mathcal{T}, 0, |\sigma| - 1)$. We will outline the steps of this computation below. To begin with, we will use a worklist of unprocessed vertices. Whenever the worklist is empty, we have discovered all the nodes and edges of the optimal segmentation G. We initialize the worklist to contain the vertex $(\mathcal{T}, 0, |\sigma| - 1)$. At each iteration, we pop a node $v = (\hat{\mathcal{T}}, i, j)$ from the worklist:

1. If $\hat{\mathcal{T}} = \mathsf{Leaf}(\varphi)$, $\hat{\mathcal{T}} = \mathsf{Tout}_t(\mathcal{T}')$ with $j + 1 - i > t$, $\hat{\mathcal{T}} = \bigstar_0(\mathcal{T}')$, or $\hat{\mathcal{T}} = \bigstar_n(\mathcal{T}')$ with $i > j$, we add no outgoing edges.
2. If $\hat{\mathcal{T}} = \mathsf{Fback}([\mathcal{T}_1, \ldots, \mathcal{T}_k])$, let $(l, i') = \arg\max_{(l,i') \in [1,k] \times [i,j]} \mathrm{T}(\mathcal{T}_l, i', j)$. We add the vertex $v' = (\mathcal{T}_l, i', j)$ to the worklist if it does not exist previously and the edge $v \to v'$.
3. If $\hat{\mathcal{T}} = \mathsf{Par}_M([\mathcal{T}_1, \ldots, \mathcal{T}_n])$, the list $[\mathrm{T}(\mathcal{T}_1, i, j), \ldots, \mathrm{T}(\mathcal{T}_n, i, j)]$ is sorted in descending order, and take the first M entries in the sorted list. We add the vertices $(\mathcal{T}_{i_1}, i, j), \ldots, (\mathcal{T}_{i_M}, i, j)$ corresponding to the first M sorted entries to the worklist (if they did not exist previously) and outgoing edges form v to all these vertices.
4. If $\hat{\mathcal{T}} = \mathsf{Seq}([\mathcal{T}_1, \mathcal{T}_2])$, let $u = \arg\max_{u \in [i,j]}(\min(\mathrm{T}(\mathcal{T}_1, i, u), \mathrm{T}(\mathcal{T}_2, u + 1, j)))$. We add the vertices $v_1 = (\mathcal{T}_1, i, u)$, $v_2 = (\mathcal{T}_2, u + 1, j)$ to the worklist (if they did not exist previously) and the edges $v \to v_1$ and $v \to v_2$.
5. If $\hat{\mathcal{T}} = \mathsf{Tout}_t(\mathcal{T}_1)$, let $t' = \arg\max_{t \in [i,j]}(\mathrm{T}(\mathcal{T}_1, i, t'))$. If it did not exist before, we add the vertex $v' = (\mathcal{T}_1, i, t')$ to the worklist and the edge $v \to v'$.
6. If $\hat{\mathcal{T}} = \bigstar_n(\mathcal{T}_1)$ and $n > 0$, $i \leq j$, we take the arguments of the maximum of $\forall k \in [1, n]. u_1, \ldots, u_k = \arg \max_{u_1 \in [i,j], \ldots, u_k \in [u_{k-1}, j]} \min(\ \mathrm{T}(\mathcal{T}_1, i, u_1), \ldots,\ \mathrm{T}(\mathcal{T}_1, u_k, j)\ )$.
   We add the vertices $(\mathcal{T}_1, i, u_1 + 1), \ldots, (\mathcal{T}_1, u_k + 1, j)$ to the worklist (if new) and create edges from v to these vertices.

Let G represent the segmentation thus obtained.

Theorem 7. *The segmentation obtained using the worklist algorithm described above is optimal, i.e., $\pi_G(\mathcal{T}, \sigma) = \rho(\mathcal{T}, \sigma)$.*

Proof. The proof of the theorem follows from the construction itself since it simply "reads off" a segmentation from the dynamic programming table T. □

Dynamic programming also allows us to rapidly explore alternative segmentations. Alternative segmentations allow to go beyond the optimal segmentation and help to better understand the executed behavior, e.g., to identify if there are "other" successful segmentations. We control finding such alternatives using two parameters: τ_t and τ_ρ. $\tau_t \in \mathbb{R}^+$ represents how much the segment boundary needs to differ in respect to already identified segmentations. $\tau_\rho \in \mathbb{R}^+$ represents a lower bound of the robustness of alternatives in respect to given segmentations.

Definition 4.3 (Alternative Segmentation)
Let σ be a trace and $\mathcal{T}$ be a temporal behavior tree, a segmentation $G' = (V', E')$ is an alternative to a segmentation $G = (V, E)$ iff $\pi_{G'}(\mathcal{T}, \sigma) > \tau_\rho$ and either $V \neq V'$ or there exists a sink vertex $(\mathcal{T}_k, i, j) \in V$ with no sink vertex $(\mathcal{T}'_k, i', j') \in V'$ for which $\mathcal{T}_k = \mathcal{T}'_k$ and $|i - i'| + |j - j'| \leq \tau_t$.

→ Fig. 4.2, p. 54 **Example 4.2.3.** Consider the segmentation provided in Figure 4.2. As can be seen, *Stay at Position* was assigned to the subtrace of σ, annotated by (2), just before moving towards the ship (3). Using alternative segmentation, we can analyze whether (2) and (3) have an earlier or later assignment that also satisfies the specification, i.e., was it possible to execute the next leaf node earlier or later. To obtain only satisfying alternatives, we choose $\tau_\rho = 0$. We also choose τ_t as the duration of staying in position as specified to make sure that our alternative is significantly different to the one obtained before. Note that by choosing $\tau_t = |\sigma|$, we are guaranteed to find an alternative that uses a different landing maneuver, if existing. $\triangle$

Computing an alternative segmentation uses the same worklist algorithm presented above, but imposes the additional constraints defined in Definition 4.3. Note that the same memo table T is used to compute both the optimal and the alternative segmentations.

Next, we address the most limiting factor of our presented dynamic programming approach – it scales quadratically in the length of the trace. We present approximation techniques that reduce the number of computations. The approximations provide valid segmentations with bounds on their robustness while avoiding false positives, i.e., whenever our approximation returns a successful segmentation, then the segmentation based on dynamic programming also returns a successful segmentation that has a robustness that is within provided bounds.

Subsampling traces for approximating robustness. Traces are often obtained by sampling a continuous signal at regular time intervals. Often, if the sampling is done rapidly compared to how the signal varies, we observe the phenomenon of *stuttering*, wherein the same truth values of atomic propositions repeat over multiple time instances. We now show how the truth/robustness on a carefully subsampled trace relates to the original trace.

Definition 4.4 (δ-Stuttering)
A trace σ is δ-stuttering iff (a) δ divides $|\sigma|$ and (b) for all i such that $i\delta < |\sigma|$, the states $\sigma(i\delta + 1), \ldots, \sigma((i+1)\delta)$ are identical in terms of the atomic proposition truth valuations: $\sigma(i\delta + 1) \models p$ iff $\sigma(i\delta + j) \models p$ for $j = \{1, \ldots, \delta\}$.

We say that a trace σ' is a subsampling of σ with *stride length* $\delta \geq 1$ iff $\sigma'(i + 1) = \sigma(i\delta + 1)$ for $i \in \{0, \ldots, \frac{|\sigma|}{\delta} - 1\}$.

Definition 4.5 (δ-Preserving TBT)

A TBT $\mathcal{T}$ is δ-preserving if (a) all occurrences of $\mathcal{U}_{[l,u]}$ in $\mathcal{T}$ has $l = 0$ and u is divisible by δ; (b) all occurrences of $\Diamond_{[l,u]}, \Box_{[l,u]}$ have l, u divisible by δ; and (c) all occurrences of Tout_t has t divisible by δ.

For δ-preserving TBT $\mathcal{T}$, define TBT $\mathcal{T}'$ with all occurrences of Tout_t replaced by $\mathsf{Tout}_{t/\delta}$ and all occurrences of temporal logic operators $\bowtie_{[l,u]}$ wherein $\bowtie \in \{\Diamond, \Box, \mathcal{U}\}$ replaced with $\bowtie_{[l/\delta, u/\delta]}$.

Example 4.2.4. Let the trace σ be $\begin{smallmatrix} a: \\ b: \end{smallmatrix}\binom{true}{false}, \binom{true}{false}, \binom{false}{true}, \binom{false}{true}$ and let the temporal behavior tree $\mathcal{T}$ be $\mathsf{Seq}([\Box(a \wedge \neg b), \Diamond b])$. Then $\delta = 2$, $\mathcal{T}'$ remains $\mathcal{T}$, and σ' is the sequence $\binom{true}{false}, \binom{false}{true}$, and both $\rho(\mathcal{T}, \sigma)$ and $\rho(\mathcal{T}', \sigma')$ are positive. △

Theorem 8. *For a δ-preserving TBT $\mathcal{T}$ and a δ-stuttering trace σ, if $\sigma' \models \mathcal{T}'$ then $\sigma \models \mathcal{T}$.*

Proof. We prove the theorem by induction on the formula. The full proof is given in Section A.2.3. □ →Sec. A.2.3, p. 166

The theorem shows that by evaluating a TBT on a subsampled trace for a δ-stuttering trace σ, if the subsampled trace satisfies the TBT, then so does the original trace. However, violations on a subsampled trace need not necessarily be violations on the original trace. The Seq operator is the reason for the failure of the converse.

Example 4.2.5. Consider a trace σ with atomic proposition p: $\neg p, \neg p, p, p, \neg p, \neg p$, and TBT $\mathcal{T}$: $\mathsf{Seq}(\mathsf{Leaf}(\Diamond p), \mathsf{Leaf}(\Diamond p))$. Choosing $\delta = 2$, we obtain $\sigma' : \neg p, p, \neg p$. The transformed TBT $\mathcal{T}' = \mathcal{T}$. Note that $\sigma \models \mathcal{T}$ whereas $\sigma' \not\models \mathcal{T}'$.

Also, it is necessary for $\mathcal{U}_{[l,u]}$ occurrences to have $l = 0$. Consider the trace σ : p, p, p, q, q, q and the formula $\varphi : \Box((p\, \mathcal{U}_{[3,3]} q) \vee q)$. Clearly $\sigma \not\models \varphi$. However, we have $\delta = 3$ and thus $\sigma' : p, q$ with $\varphi' : \Box((p\, \mathcal{U}_{[1,1]} q) \vee q)$. It follows that $\sigma' \models \varphi'$. △

Note that, for simplicity, we computed the stride length δ based on the fact that $p \in AP$ remains unchanged in every subtrace $\sigma[i\delta : (i+1) \cdot \delta - 1]$. This is too stringent in practice. For instance, the formula $\Box(p_1 \vee p_2 \wedge p_3) \vee p_4$ with $AP = \{p_1, p_2, p_3, p_4\}$ will be replaced by AP' that consists of p_4 and q where $f_q = \max(f_{p_1}, \min(f_{p_2}, f_{p_3}))$. For our experiments in Section 4.3, we implemented this improvement to increase δ but also to reduce the size of the TBT $\mathcal{T}$. →Sec. 4.3, p. 63

Approximate Robustness Using Lazy Evaluation Next, we introduce a lazy evaluation for computing the robustness of a TBT given a trace. For the lazy evaluation, we use operators common in functional and dynamic programming that rely on iterators and coroutines [138, 139]. Using these concepts, we define $\min_{lazy}$ as follows:

```
define min_lazy(list_of_exprs):
   min_so_far = ∞
   # initialize lazy evaluation of all expressions
   list_of_gens = [lazy_eval(e) for e in list_of_exprs]
   for g in list_of_gens:
      if has_next(g):
         l = next(g) # take the next value yielded
         if l < 0 and l < min_so_far:
             yield
         min_so_far = min(min_so_far, l)
   return min_so_far
```

We define $\max_{lazy}$ in an analogous manner. The key idea here is that a call to $\min_{lazy}$ will yield as soon as it finds a negative value or a value smaller than what it yielded previously. Many programming language implement these features, e.g., Python [1]. We execute the dynamic programming starting until the overall root expression yields its first value and stops. Even if this value is not the exact robustness, we can relate this value to the exact result obtained upon a full computation.

Definition 4.6 (Lazy-Evaluation)

Given a TBT $\mathcal{T}$ and a trace σ, we can evaluate the robustness in a lazy manner, denoted as $\rho_{lazy}(\mathcal{T}, \sigma)$, by replacing all instances of max by $\max_{lazy}$ and min by $\min_{lazy}$ in $\rho(\mathcal{T}, \sigma)$. In addition, we will also use memorization to cache previously evaluated expressions in order to avoid re-evaluation.

Example 4.2.6. Given the same TBT and trace as in Example 4.2.4. The lazy evaluation starts by sequencing the trace right after the first position into two segments. While computing the robustness of the left segment using $\Box$ no lazy return happens since there is only one position. When computing the right side using $\Diamond$ the first lazy return occurs right for the second value provided by the iterator. The evaluation returns to the sequence node, where both segments have a positive robustness. Hence, lazy return will directly return the minimum of these segments. If only one of the segments would have been negative, the evaluation would backtrack to the right segment and continues to call the iterator at the position where it returned before. Note that in this case, the computed robustness is optimal. This is not necessarily the case. $\triangle$

Theorem 9. *Given a TBT $\mathcal{T}$ and a trace σ, $\rho_{lazy}(\mathcal{T}, \sigma) \geq 0$ iff $\rho(\mathcal{T}, \sigma) \geq 0$.*

Proof. The same algorithm as $\rho(\mathcal{T}, \sigma)$ is used and coroutines maintain an internal state that allows them to continue an evaluation of $\max_{lazy}$ and $\min_{lazy}$ when returning to them. $\Box$

[1] https://realpython.com/introduction-to-python-generators/

4.3. Empirical Evaluation and Case-Studies

In this section, we will present different case studies that show how the segmentation of TBTprovides useful insights.

4.3.1. Analysis of Human Behavior

We use segmentation to analyze human operator performance. We obtained data from the study conducted by Byeon et al, wherein thirteen subjects repeatedly attempted to fly a drone in a simulated environment using a joystick setup to control the drone's altitude and attitude while avoiding the obstacle [140]. Each subject attempted the same task of navigating through the waypoints 25 times. The goal of the simulation was to see if the operator through these repeated trials will learn how to navigate the drone to take off and fly through six different waypoints, ending up at a pre-specified position. We used the TBT shown in Figure 4.4 to specify the overall task. Note that the entire task is a sequence of moves from one waypoint to the next reaching the terminus.

We use segmentation of each of the traces of (x, y, z) trajectories of the human operator to understand how their performance evolved over the trials. Note that segmentation is challenging since human operators often behave in ways we could not predict a priori. Figure 4.5 shows the segmentations obtained by our approach for two different operators and trials (there are a total of 13×25 such traces). The segmentations are obtained automatically given the traces and the specification. They split the entire trajectory into seven parts, ascribing each to a subtree which in this case corresponds to reaching a waypoint. Note from Figure 4.5 that it is often hard to perform this segmentation manually especially when the specifications are violated (they are violated in both cases due to missed waypoints and collision with the obstacle in Trial #2). The segmentation allows us to analyze how the subjects are learning or failing to learn the performance of the overall task over each trial. For each trial, we collect the robustness of each segment with respect to its corresponding node in the tree. The plots in Figure 4.6 reveal consistent trends that were observed across all the subjects: (a) Most subjects could carry out the first phase successfully and the overall spread of robustness is relatively small. (b) The second phase ends up being the most challenging. Very few subjects over few trials could navigate this successfully. We suspect that the presence of the obstacle right next to the waypoint for this phase plays a role in this. (c) Subjects are able to navigate phases 3-6 successfully on average but their performance varies across trials. (d) Subjects are consistently unable to navigate the terminal phase. Our preliminary analysis clearly demonstrates the usefulness of a systematic approach to segmentation. The full analysis of how humans learn to teleoperate successfully over the course of multiple trials will be described in an extended version.

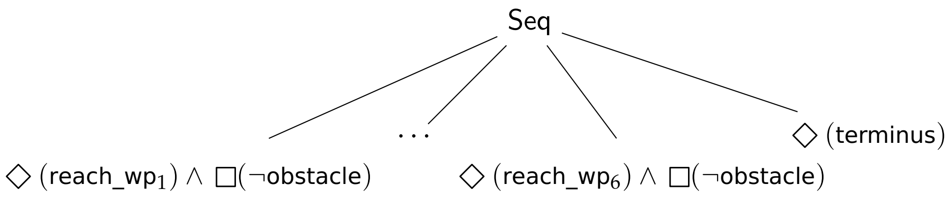

Seq
◇ (reach_wp$_1$) ∧ □(¬obstacle)
. . .
◇ (reach_wp$_6$) ∧ □(¬obstacle)
◇ (terminus)

Figure 4.4.: Temporal behavior tree that was used to specify the task for each participant in the drone flying task. The task consists of reaching a sequence of six waypoints while avoiding two obstacles in turn and ending up at a terminus.

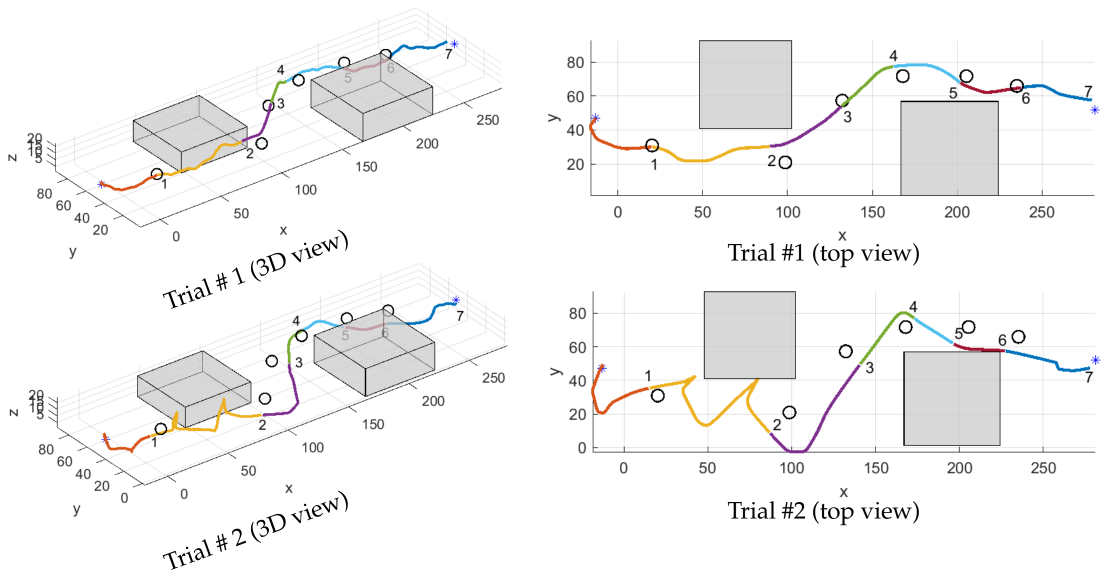


Figure 4.5.: Segmented trajectories from two simulator flights. Each segment is color-coded and numbers at the endpoints indicate the respective waypoint. Waypoints are shown as circles and obstacles are shaded gray.

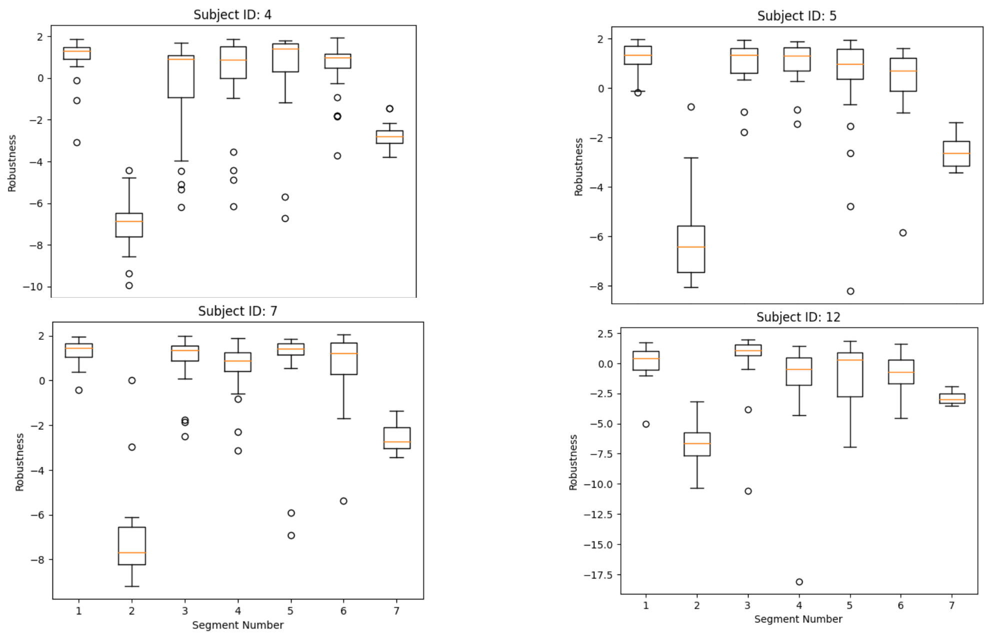


Figure 4.6.: Box plots showing the spread of robustness values corresponding to each of the seven task segments for four different subjects.

4.3.2. Autonomous Ship Deck Landing

Landing on a ship deck is well-known to be a challenging task, wherein various landing aids and maneuvers need to be carefully selected [141, 142]. The TBT presented in Figure 4.7 formally specifies four different landing maneuvers: *Straight-in*, *Lateral*, *45-Degree*, and *Oblique*. At the top of the TBT, there is a sequence node that executes its children from left to right. The first child is a fallback node and the second is a leaf node that represents the descent to the touchdown point. Each maneuver is structured as a sequence with different atomic propositions. The maneuvers differ in their starting position and their heading. Whereas, the 45-Degree and the Oblique landing maneuvers specify that the UAV must be diagonally behind the ship, they differ in the relative heading of the UAV to the ship, i.e., aligned with versus oblique to the ship heading, respectively.

→ Fig. 4.7, p. 68

We use our TBT on simulation data provided by the authors of [141, 142] for landing with a UAV on a ship deck under wind conditions from the side (WS) and from the front (WF). All experiments were run on a single 16-core machine with a 2.50 GHz 11th Gen Intel(R) Core(TM) i7-11850H processor with 32 GB RAM. The algorithms are implemented as a single-threaded program using Rust[2]. Experimental results are given in Table 4.8. The first column indicates the expected behavior for a landing. Each of the log files have mission times between 115 and 127 seconds and contain between 22,046 and 25,313 entries. The second column reports the stride length δ. Next, *Time* represents the execution time required to find a segmentation where ρ_δ refers to the presented subsampling and $\rho_\delta \circ \rho_{lazy}$ refers to first subsampling and then running a lazy evaluation (Definition 4.6). Finally, we report on the *Chosen Maneuver* and the *Worst Segment* given the computed segmentation. For the case where only subsampling was used, at least 2,000,000 and up to 500,000,000 evaluations could be resolved by the memo table. The results show that subsampling is very efficient and helps to find segmentations within seconds. Yet, the dependence on the length of the trace becomes clear when we compare the required time required for $\delta = 25$ and $\delta = 200$. Our segmentations show that oblique maneuvers potentially failed. This also caused the lazy evaluation to take as long as ρ_δ because early returns of $\min_{lazy}$ and $\max_{lazy}$ are not possible in this case. Further, we can see that the most challenging part of the maneuver is *Descend*, which makes sense given the disturbances due to wind and waves when landing on the ship. In fact, segmentation of the *Straight-in-WF* log file potentially failed due to its descend. Note that the best robustness for a *Descend* is one since we allow a tolerable deviation from the target position by one meter, see Figure 4.7. Results indicate that wind did not play a major role during the maneuvers.

→ Tbl. 4.8, p. 69

→ Def. 4.6, p. 62

We now examine the potential failing segmentations for oblique maneuvers. Note that by Theorem 8, given a failing segmentation, we cannot draw conclusions about the segmentation of the original trace. We also see that the segmentation for such a

→ Thm. 8, p. 61

[2] https://github.com/DLR-FT/TBT-Segmentation

maneuver is always a 45-Degree maneuver, depicted in Figure 4.2. In Figure 4.9, we show an alternative segmentation that is worse but uses an oblique landing approach. We received this alternative by using τ_t that was set greater than $|\sigma|$. Choosing such a τ_t ensures that the alternative segmentation uses different nodes for its segments. Figure 4.9 shows that the *stay in position* is poorly assigned. We further examined the robustness values and concluded that the oblique heading relative to the ship expected by the TBT was not sufficiently implemented by the controller. Hence, it behaves "closer" to a 45-degree maneuver than an oblique maneuver. The results of both use-cases show that segmentation provides key insights into complex behaviors. Further, our experiments show that our current segmentation techniques can be used for large traces through a combination of subsampling the trace and approximation through lazy evaluation. – segmentations are computed within seconds for practical use-cases.

➔Fig. 4.2, p. 54

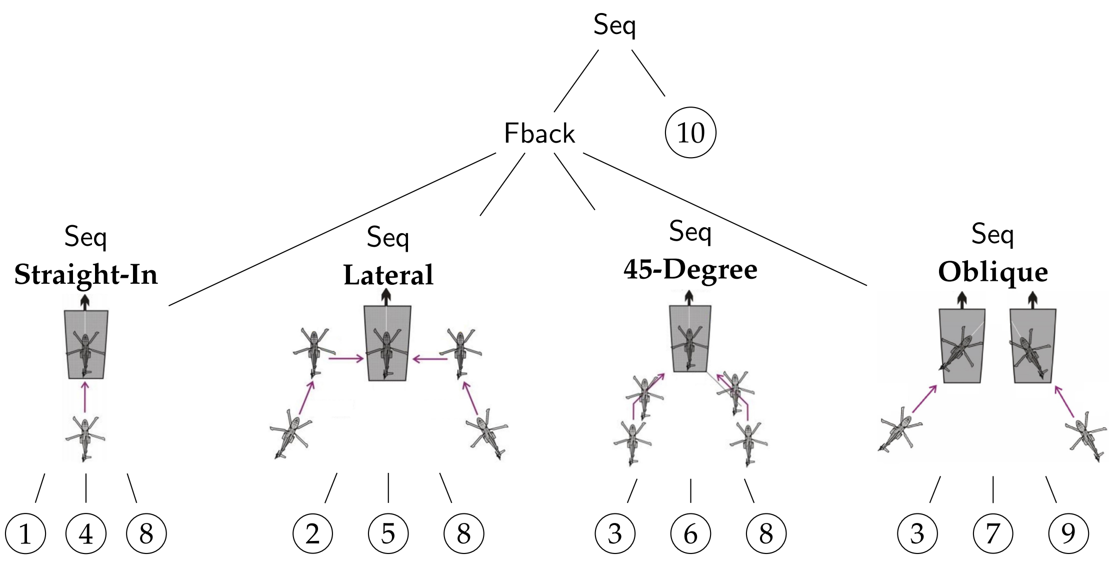
4.4. Summary

This chapter introduced the problem of trace segmentation. A segmentation systematically assigns nodes of the TBT to subtraces using TBT's robust semantics. We have introduced an algorithm to compute optimal and alternative segmentations efficiently. The algorithm is based on dynamic programming where its scalability is further improved through subsampling and lazy evaluation approximation techniques.

We showed on two use-cases the practical value of segmentation when analyzing or debugging complex behaviors. First, we analyzed the performance of 25 human-operators flying a UAV waypoint mission, revealing how segmentation captures learning patterns and failure points. Second, we report on using segmentation for developing an autonomous landing of a UAV on a ship deck, showing its ability to diagnose challenges in complex maneuvers. Both use-cases highlight the relevance of segmentation in practice and confirm that our algorithms can efficiently compute meaningful segmentations that provide deeper insights into which parts of a mission succeeded or failed.

Remark 4.1 (Controlling Trace Segmentation Through TBT Design)

The structure of the TBT directly impacts how the trace is segmented. For instance, the TBT in Figure 4.7 produces a segmentation composed of four segments. If, instead, the TBT that merges leaves (1) and (2), i.e., $\Diamond\Box_{[0,1000]behind(p_s,p_{uav})}$, the resulting segmentation would consist of only three segments. Changing the TBT allows a user to extract the information that is most relevant to their analysis.



Leaf	Description	Temporal Formula
1	Behind ship	$\Diamond\ behind(p_s, p_{uav})$
2	Next to ship	$\Diamond\ nextTo(p_s, p_{uav})$
3	Diagonal behind ship	$\Diamond\ diagonalBehind(p_s, p_{uav})$
4	Stay in behind	$\Box_{[0,1000]}\ (behind(p_s, p_{uav})\ \wedge\ heading(aligned, h_s, h_{uav})\ \wedge\ velocity(aligned, v_s, v_{uav}))$
5	Stay next to	$\Box_{[0,1000]}\ (nextTo(p_s, p_{uav})\ \wedge\ heading(aligned, h_s, h_{uav})\ \wedge\ velocity(aligned, v_s, v_{uav}))$
6	Stay diagonal behind	$\Box_{[0,1000]}\ (diagonalBehind(p_s, p_{uav})\ \wedge\ heading(aligned, h_s, h_{uav})\ \wedge\ velocity(aligned, v_s, v_{uav}))$
7	Stay diagonal behind	$\Box_{[0,1000]}\ (diagonalBehind(p_s, p_{uav})\ \wedge\ heading(oblique, h_s, h_{uav})\ \wedge\ velocity(aligned, v_s, v_{uav}))$
8	Move to touchdown	$\Diamond(move\_to\_touchdown(above, p_s, p_{uav})\ \wedge\ heading(aligned, h_s, h_{uav}))$
9	Move to touchdown	$\Diamond(move\_to\_touchdown(above, p_s, p_{uav})\ \wedge\ heading(oblique, h_s, h_{uav}))$
10	Descending	$\Diamond\ (descended(p_{touchdown}, p_{uav}))$

Figure 4.7.: TBT specifying landing maneuvers on a ship deck: straight-in, lateral, 45-degree, and oblique. They differ in their starting position and their heading relative to the ship. Computations of atomic propositions are omitted. Note that the figures labeled by the maneuver, e.g., Lateral, are just for illustrating the corresponding maneuver and are not part of the TBT.

Log file	δ	Time $(\rho_\delta, \rho_\delta \circ \rho_{lazy})$ [s]	Chosen Maneuver $(\rho_\delta, \rho_\delta \circ \rho_{lazy})$	Worst Segment $(\rho_\delta, \rho_\delta \circ \rho_{lazy})$
45-Degree-WF	50	$(4, < 0)$	(45-Degree, 45-Degree)	(Descend 0.49, Descend 0.05)
45-Degree-WS	200	$(< 0, < 0)$	(45-Degree, 45-Degree)	(Descend 0.45, Descend 0.20)
Lateral-WF	100	$(< 0, < 0)$	(Lateral, Lateral)	(Descend 0.39, Descend 0.06)
Lateral-WS	100	$(< 0, < 0)$	(Lateral, Lateral)	(Descend 0.30, Descend 0.01)
Oblique-WF	25	(29, 29)	(45-Degree, 45-Degree)	(Move-to-pos -0.58, $-\infty$)
Oblique-WS	25	(28, 28)	(45-Degree, 45-Degree)	(Move-to-pos -0.34, $-\infty$)
Straight-in-WF	200	$(< 0, < 0)$	(Straight-in, Straight-in)	(Descend -0.60, $-\infty$)
Straight-in-WS	200	$(< 0, < 0)$	(Straight-in, Straight-in)	(Descend 0.22, Descend 0.22)

Table 4.8.: Results of segmentation using the TBT in Figure 4.7. The segmentation uses subsampling (Definition 4.4) alone and in combination with lazy evaluation (Definition 4.6). All log files represent realistic missions that took approximately two minutes ($> 20,000$ entries).

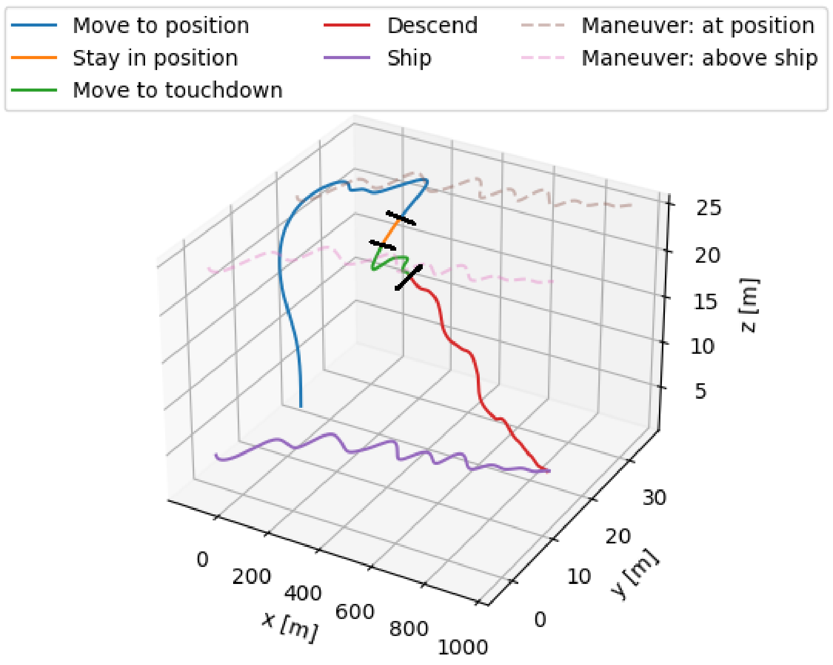


Figure 4.9.: Segmentation of an oblique maneuver where the UAV deviates from the expected behavior. Ideally, the “Stay in position” segment should align with the “Maneuver: at position” line.

Chapter 5

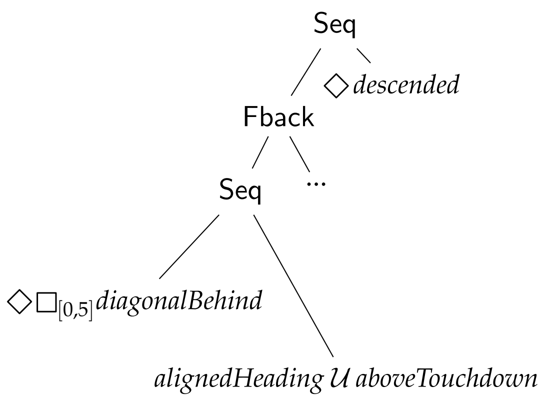
Trace Repair

In this chapter, we study the trace repair problem: giving a trace that does *not* satisfy its specification, the goal is to compute a *repaired* trace that satisfies the specification while minimally modifying the original trace. The repaired trace can be used as a modified plan that corrects the output of an untrusted planner based on a correctness specification. Furthermore, it provides valuable information about what went wrong in the violating trace. For systems that involve a machine-learned component (or a human operator), they are also useful as training examples to avoid the problem in the future. Here, we use TBTs as specification formalism.

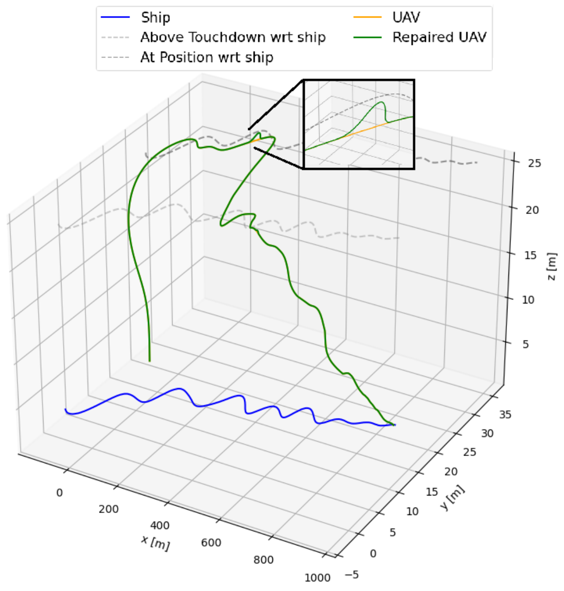
Figure 5.1a shows an excerpt of a TBT that specifies an automated landing maneuver of a UAV on a ship. The TBT decomposes the maneuver into two parts before descending: first, the UAV moves to a position diagonally behind the ship and holds there for five seconds; second, the UAV transitions above the touchdown point while aligning its heading with the heading of the ship. Typically, the TBT would include other fallback maneuvers, such as approaching the ship from behind or from the side, which we omit here for brevity (see Figure 4.7). The formulas at the TBT's leaf nodes specify the successful execution of the various actions. For instance, the UAV reaching and holding a diagonal position behind the ship is expressed by the STL formula $\Diamond\Box_{[0,5s]}\mathit{diagonalBehind}$. Figure 5.1b shows a plot of an executed landing maneuver (yellow line). Although the UAV seems to reach the dotted line representing the correct diagonal position, it is actually slightly off and therefore violates the specification. The minimal trace repair (green line) corrects this by adjusting the UAV's position closer to the dotted line while keeping the rest of the trace unchanged.

→Fig. 4.7, p. 68

In theory, the repair problem could be solved using mixed-integer linear programming (MILP), where the system model, the trace, and the relevant parts of the TBT specification are encoded as constraints. To compute an "optimal repair" we impose a cost function, such as the L_1 norm, that penalizes deviation of the repaired trace states from the original trace states. Since the size of the MILP grows with the length of the


Seq
$\Diamond$ *descended*
Fback
Seq
...
$\Diamond\Box_{[0,5]}$*diagonalBehind*
alignedHeading $\mathcal{U}$ *aboveTouchdown*

(a) Excerpt from a TBT specifying a UAV landing maneuver on a ship.



(b) Trace repair of an execution of the landing maneuver against the TBT in Figure 5.1a.

Figure 5.1.: Repair of a UAV ship landing maneuver [49] against a TBT specification.

trace, the overall complexity of solving the MILP is exponential in the size of the trace. As a result, the MILP encoding approach does not scale.

We address this problem with two complementary repair strategies. The first strategy, *incremental repair*, uses the segmentation provided by the TBT monitor (see Chapter 4) to avoid encoding the entire trace. Instead, it locally repairs segments, starting with leaf nodes and incrementally moving up the TBT only if necessary. Note that the segmentation serves as a heuristic, which may be off for repair, e.g., due to constraints imposed by the system model. In our experiments, for a landing depicted in Figure 5.1b with a trace length of 1014, this approach successfully repairs within 30 seconds, whereas the straightforward MILP encoding takes over 2000 seconds. The second strategy, *landmark-based repair*, is an iterative approach that identifies candidate landmarks within the trace for the repair. If a repaired trace is found that satisfies the landmark, then its satisfaction guarantees the satisfaction of the TBT specification. Given the abundance of candidates, we use the robust semantics of TBTs as a heuristic to find good candidates. This approach entirely avoids the integer and binary variables introduced by the TBT encoding, enabling the problem to be solved as a linear program instead. As a result, landmark-based repair successfully finds a solution for a trace with over 25, 000 entries in under ten minutes, where approaches that fully encode the TBT run out of memory.

→Chapter 4, p. 53

The remainder of this chapter is structured as follows: Section 5.1 provides background on system models, then Section 5.2 introduces the repair strategies, finally Section 5.4 demonstrates experimental results. →Sec. 5.4, p. 83

5.1. System Model

We consider discrete-time systems with linear dynamics, represented by

$$X_{t+1} = A \cdot X_t + B \cdot U_t, \tag{5.1}$$

where $X_t \in \mathbb{X} \subseteq \mathbb{R}^n$ is the state at time step t with n variables, $u_t \in \mathbb{U} \subseteq \mathbb{R}^m$ is the control input at time t. A is an $n \times n$ while B is a $n \times m$ matrix [143].

Example 5.1.1. Consider a one-dimensional integrator system with position p and velocity v as states, i.e., $X = [p, v]^T$, and an acceleration command as the input U. For a sampling time t_s, the system is described as

$$\begin{bmatrix} p_{t+1} \\ v_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & t_s \\ 0 & 1 \end{bmatrix} \begin{bmatrix} p_t \\ v_t \end{bmatrix} + \begin{bmatrix} t_s^2/2 \\ t_s \end{bmatrix} U_t.$$

△

The execution of a system provides a *trace* σ that is a finite sequence of states $\sigma(1), \ldots, \sigma(N)$, where $\sigma(i)$ represents the state of the model and the control input at timesteps $i \in \{1, \ldots, N\}$, i.e., $[X_{i-1}, U_{i-1}]^T$. To access the state of the model X_{i-1} at timestep i, we define a projection operation π_X that extracts this state from $\sigma(i)$: $\pi_X(\sigma(i)) = X_{i-1}$.

The length of a trace is denoted as $|\sigma|$, with $|\sigma| = 0$ indicating an *empty trace*. Given a trace σ, the expression $\sigma[l : u]$ retrieves a slice of the trace from offsets l to u with $l, u \in \mathbb{N}_0$:

$$\sigma[l : u] ::= \begin{cases} \sigma(l+1), \cdots, \sigma(\min(u+1, N)), & \text{if } l \leq u \text{ and } l < N \\ \text{empty trace} & \text{otherwise} \end{cases}$$

For convenience, let $\sigma[l :] = \sigma[l : N-1]$ and $\sigma[: u] = \sigma[0 : u]$.

Example 5.1.2. Consider the system described in Example 5.1.1. Let $t_s = 1\text{s}$, $X_0 = [0, 1]^T$ and $U_t = 0$ for all timesteps t. Executing the system for three timesteps yields $X_1 = [1, 1]^T$, $X_2 = [2, 1]^T$, and $X_3 = [3, 1]^T$. The corresponding trace has a length of four where $\sigma(1) = [0, 1, 0]^T$, $\sigma(2) = [1, 1, 0]^T$, $\sigma(3) = [2, 1, 0]^T$, and $\sigma(4) = [3, 1, 0]^T$. To exclude the first trace entry, slicing can be applied: $\sigma[1 :] = \sigma(2), \sigma(3), \sigma(4)$. Similarly, $\sigma[3 : 3] = \sigma(4)$ only accesses the last entry. △

5.2. Repairing Traces

Given a system model M, a TBT $\mathcal{T}$, and a trace σ with $\sigma \not\models \mathcal{T}$, we construct a repaired trace σ_R that satisfies $\mathcal{T}$ with $|\sigma_R| = |\sigma|$, denoted by $\mathsf{repair}_{\sigma_R}(\mathcal{T}, M, \sigma, \xi)$, as follows:

$$\left.\begin{aligned} \arg\min\nolimits_{\sigma_R} \quad & J(\sigma, \sigma_R) && \leftarrow \text{Minimize cost function} \\ s.t. \quad & \sigma_R \models \mathsf{encode}_{\sigma_R}(M) && \leftarrow \text{Trace follows system model} \\ & \sigma_R \models \mathsf{encode}_{\sigma_R}(\mathcal{T}) && \leftarrow \text{Trace satisfies TBT} \\ & \sigma_R \models \xi && \leftarrow \text{Additional constraints, cf. Section 5.3} \end{aligned}\right\} \quad (5.2)$$

Note that $\mathsf{repair}_{\sigma_R}$ does not change the length of the trace: each state of the repaired trace is a one to one correspondent to a state of the original trace. A repair problem can be infeasible. For instance, one may need a strictly longer trace σ_R to satisfy $\mathcal{T}$. In this case, $\mathsf{repair}_{\sigma_R}$ returns none. In the following, we introduce $\mathsf{encode}_{\sigma_R}(M)$, $\mathsf{encode}_{\sigma_R}(\mathcal{T})$, and present different cost functions J. We then present complementary repair strategies. We encode σ and σ_R using $2 \cdot |\sigma|$ continuous variables with $|\sigma|$ constraints for the original trace.

Encoding of the System Model: Since we consider discrete-time systems with linear dynamics, Equation (5.1) can be directly encoded as a MILP with X_t being the states of the repaired trace σ_R. In this way we ensure that σ_R follows the dynamics. We furthermore introduce lower and upper bounds for each control input in U_t. We denote this encoding by $\mathsf{encode}_{\sigma_R}(M)$. The encoding $\mathsf{encode}_{\sigma_R}(M)$ adds $3 \cdot |\sigma_R|$ constraints. These constraints include the system model itself, a lower bound on the inputs, and an upper bound on the inputs. Since A and B are constants, the encoding adds $|\sigma|$ continuous variables for U.

Encoding of the Temporal Behavior Tree: Since the length of the trace is known, the satisfaction of a trace with respect to a TBT specification, as defined in Definition 3.9, can be encoded in a MILP by extending the MILP encoding for STL to TBT. For a detailed explanation for STL, we refer to [82]. Next, we provide a summary of the STL encoding before extending it to TBTs.

→ Def. 3.9, p. 45

For each STL (sub)formula ψ, and for a trace σ_R with $|\sigma_R| = N$, we introduce a binary variable z_t^{ψ} for each trace position $0 \leq t < N$. The variable $z_t^{\psi} = 1$ if and only if ψ is satisfied at position t in the trace. Since the length of the trace is known, existential quantifiers $\exists j \in [1, \ldots, N]$ and universal quantifiers $\forall j \in [1, \ldots, N]$ are encoded by $\bigvee_{j=1}^{N}$ and $\bigwedge_{j=1}^{N}$, respectively. Thus, the remaining task is to encode $\vee$ and $\wedge$, which can be done as in [82]:

$$\psi = \bigwedge_{i=1}^{m} \varphi_i : \begin{cases} z_t^{\psi} \leq z_t^{\varphi_i}, i \in [1, \ldots, m] \\ z_t^{\psi} \geq 1 - m + \sum_{i=1}^{m} z_t^{\varphi_i} \end{cases} \qquad \psi = \bigvee_{i=1}^{m} \varphi_i : \begin{cases} z_t^{\psi} \geq z_t^{\varphi_i}, i \in [1, \ldots, m] \\ z_t^{\psi} \leq \sum_{i=1}^{m} z_t^{\varphi_i} \end{cases}$$

To extend the encoding to TBTs $\mathcal{T}$, we introduce binary variables that account for the current segment: $z^{\mathcal{T}}_{[t_1:t_2]}$. let $l = t_2 - t_1 + 1$, the encoding is as follows:

$$\begin{aligned}
\mathcal{T} &= \mathsf{Leaf}(\varphi) &&: z^{\mathcal{T}}_{[t_1:t_2]} = z^{\varphi}_{[t_1:t_2]} \\
\mathcal{T} &= \mathsf{Fback}([\mathcal{T}_1, \dots, \mathcal{T}_n]) &&: z^{\mathcal{T}}_{[t_1:t_2]} = \bigvee_{j=1}^{n} \bigvee_{i=t_1}^{t_2} z^{\mathcal{T}_j}_{[i:t_2]} \\
\mathcal{T} &= \mathsf{Par}_M([\mathcal{T}_1, \dots, \mathcal{T}_n]) &&: z^{\mathcal{T}}_{[t_1:t_2]} = (\textstyle\sum_{j=1}^{n} z^{\mathcal{T}_j}_{[t_1:t_2]}) \geq M \\
\mathcal{T} &= \mathsf{Seq}([\mathcal{T}_1, \mathcal{T}_2]) &&: z^{\mathcal{T}}_{[t_1:t_2]} = \bigvee_{i=t_1}^{t_2-1} (z^{\mathcal{T}_1}_{[t_1:i]} \wedge z^{\mathcal{T}_2}_{[i+1:t_2]}) \\
\mathcal{T} &= \mathsf{Tout}_t(\hat{\mathcal{T}}) &&: z^{\mathcal{T}}_{[t_1:t_2]} = l \leq t \wedge z^{\hat{\mathcal{T}}}_{[t_1:t_2]} \\
\mathcal{T} &= \bigstar_n(\hat{\mathcal{T}}) &&: z^{\mathcal{T}}_{[t_1:t_2]} = (l > 0 \wedge n > 0 \wedge z^{\bigstar_{n-1}(\hat{\mathcal{T}})}_{[i+1:t_2]})) \vee \neg(l > 0 \wedge n > 0)
\end{aligned} \tag{5.3}$$

We denote this encoding by $\mathsf{encode}_\sigma(\mathcal{T})$. A trace σ satisfies a specification $\mathcal{T}$ iff $z^{\mathcal{T}}_{[0:N-1]} = 1$. Let $|\varphi|$ represent the size of the temporal formula. Unlike the previous STL encoding, which introduces $(|\mathrm{AP}| + |\varphi|) \cdot N$ binary variables, this encoding requires $(|\mathrm{AP}| + |\mathcal{T}|) \cdot N^2$ binary variable to account for the different segments with the same number of constraints that track their satisfaction.

Example 5.2.1. Consider the TBT $\mathsf{Seq}([\mathsf{Leaf}(\Diamond\Box \mathit{hasKey}), \mathsf{Leaf}(\mathit{openDoor})])$, abbreviated by $\mathsf{Seq}(\dots)$, that specifies that a robot must first find a key before opening the door. Given a trace of length ten, to check whether the trace satisfied the TBT, i.e., $z^{\mathsf{Seq}(\dots)}_{[0:9]} = 1$, it is necessary to search for a satisfying transition between its children. We therefore compute $\bigvee_{i=0}^{8} (z^{\mathsf{Leaf}(\Diamond\Box \mathit{hasKey})}_{[0:i]} \wedge z^{\mathsf{Leaf}(\mathit{openDoor})}_{[i+1:9]})$. △

Encoding of Cost Functions The cost function in Equation (5.2) ensures that the repaired trace σ_R is *similar* to the original trace σ, thereby providing a clear and intuitive explanation of *what should have been done differently* to satisfy the TBT $\mathcal{T}$ specification. We consider the following cost functions:

- *L1-Distance* $L1(\sigma, \sigma_R) = \sum_{i=1}^{|\sigma|} \|\pi_X(\sigma(i)) - \pi_X(\sigma_R(i))\|_1$
  This metric computes the point-wise distance between both trajectories.
- *Hamming-Distance* $H(\sigma, \sigma_R) = \sum_{i=1}^{|\sigma|} \begin{cases} 1 & , \pi_X(\sigma(i)) \neq \pi_X(\sigma_R(i)) \\ 0 & , \text{otherwise.} \end{cases}$
  This metrics counts the changes necessary to the original trajectory.

We also consider the robust semantics of TBTs as cost function. Instead of providing a Boolean verdict, the robust semantics of a TBT yields a numerical value (see Theorem 2). →Thm. 2, p. 46 A positive value indicates that the specification is satisfied, while a negative value corresponds to a violation. Additionally, the magnitude of the numerical value reflects the degree to which the specification is satisfied or violated. In essence, the robust semantics of TBTs can be viewed as a variation of the Boolean semantics described in Definition 3.9. Specifically, in the robust semantics, all instances of $\exists$ and $\vee$ are replaced by max, while all instances of $\forall$ and $\wedge$ are replaced by min. This transformation allows the semantics to yield a numerical value that indicates the degree of satisfaction that can be used as cost function $R(\sigma_R)$ as in [82] to maximize "satisfaction". We also introduce a weighted combination $W(\sigma, \sigma_R)$, which integrates multiple objectives and constraints into the repair strategy: $\tau_{L1} \cdot L1(\sigma, \sigma_R) + \tau_H \cdot H(\sigma, \sigma_R) + \tau_R \cdot R(\sigma_r)$ where τ_{L1}, τ_H, and τ_R are positive weights that sum up to one. Note that we can encode $|x|$ using linear constraints within the cost function by defining an auxiliary variable $a = |x| : a \geq x \wedge a \geq -x$ or using the Big-M method [144]. This thesis focuses on cost functions that are intuitive for pilots and control engineers. Other cost functions that find repairs using shorter traces will be explored in future.

→Def. 3.9, p. 45

5.3. Repair Strategies

Equation (5.2) provides an optimal repair, but for long traces and complex specifications, the solution quickly becomes impractical, as indicated by the reported number of variables. We introduce two repair strategies, to address two distinct scalability concerns. The first strategy is an incremental approach, enabling local repairs of violating trace segments, which avoids encoding the full trace. The second strategy utilizes landmarks to resolve choices introduced by disjunctions in the specification. This reduces the MILP encoding to a linear program, which allows for more efficient optimization algorithms, such as the simplex algorithm [145]. The strategies can be applied both in isolation and in combination with each other.

5.3.1. Incremental Repair Strategy

Incremental repair strategy is based on a *segmentation* of the original violating trace with respect to the TBT. Segmentation was presented in detail in Chapter 4, wherein a segmentation of a trace w.r.t. a TBT shows how the overall robustness of the trace w.r.t. a specification $\mathcal{T}$ can be decomposed into the robustness of sub-traces with respect to sub-trees of $\mathcal{T}$. Incremental repair uses the segmentation to attempt *local repairs* of parts of the trace w.r.t. parts of the specification. If the local repair fails, the incremental repair widens the scope of the repair iteratively, falling back on the original full repair of the trace w.r.t. the entire specification in the worst case.

→Def. 4.2, p. 56

First, we recall the notion of a segmentation (see Definition 4.2). A segmentation of a trace with respect to a TBT $\mathcal{T}$ divides the trace σ into multiple subtraces of the form $\sigma[i : j]$ and assigns a (sub)tree to each of them. Note the correspondence between the nodes in a segmentation and the notion of a trace satisfying/violating a TBT from

Definition 3.9. From the construction of a TBT, we establish an important result that satisfaction of a TBT reduces to checking the temporal formulas at the leaves given the segmentation. → Def. 3.9, p. 45

Proposition 10. $\sigma \models_G \mathcal{T}$ *if and only if for all* $(\mathsf{Leaf}(\varphi), i, j) \in V, \sigma[i:j] \models \varphi$.

Proof. The proof of the proposition follows by induction on the structure of the TBT and matching the semantics of TBT (Definition 3.9) against the definition of a segmentation (Definition 4.2). □ → Def. 3.9, p. 45 → Def. 4.2, p. 56

However, an "optimal" segmentation can be defined and computed even for violating traces, i.e., a subtrace violates its assigned subtree. Such a segmentation provides useful clues for repairing the trace.

Example 5.3.1. Consider a robot tasked with searching for objects, specifically apples and oranges[1], while avoiding certain areas. Assume there are three known locations, L_1, L_2, and L_3, where these objects are most likely to be found. The robot needs to search these locations in some order and discover either an apple or an orange in each location within a specified time limit. We can encode this search task using a TBT, as shown in Figure 5.2. The APs *found*🍏 and *found*🍊 return 1 if the robots reports finding an apple or an orange, respectively; otherwise, they return -1. The APs atL_1, atL_2, and atL_3 denote that the robot is at positions L_1, L_2, and L_3, respectively. Similar, the AP *avoidArea* is satisfied iff the robot is outside the areas. Note that according to the TBT, the order in which the robot finds an apple, an orange, or both is not relevant for successfully completing the task.

Figure 5.3 provides a segmentation for a trace of length 100 using the TBT specification depicted in Figure 5.2. The segmentation shows that the robot tries to reach location L_2 and then tries to find an orange. To check whether the execution was successful, the satisfaction of each leaf node must be evaluated. Assuming all leaves satisfy their STL formula with respect to their respective segment, except $\mathsf{Leaf}(\square\, avoidArea)$. Since $((\mathsf{Par}_1, 61, 99), (\mathsf{Leaf}(\lozenge_{[0,30]} found$🍊$), 61, 99)\ ) \in E$ and the leaf is satisfied, it follows that $(\mathsf{Par}_1, 61, 99)$ is satisfied. The same holds for $(\mathsf{Seq}, 60, 99)$ and $(\mathsf{Fback}, 0, 99)$: both are satisfied. However, since $(\mathsf{Leaf}(\square\, avoidArea), 0, 99)$ is not satisfied, $(\mathsf{Par}_2, 0, 99)$ is also not satisfied. Ideally, we only need to repair $\mathsf{Leaf}(\square\, avoidArea)$ to satisfy the TBT specification. △

Next, we present the incremental repair strategy. This strategy first tries to repair violating segments *locally* with respect to their respective leaf nodes. In doing so, we may need to ensure that there is a valid transition from the end states of the repaired trace segments to the states of the original trace, so that the repaired segment can be substituted into the original trace. If this local repair fails, it incrementally widens the scope of the repair, first allowing changes in end states (later referred to as *loose*), then

[1]`https://robohub.org/introduction-to-behavior-trees/`

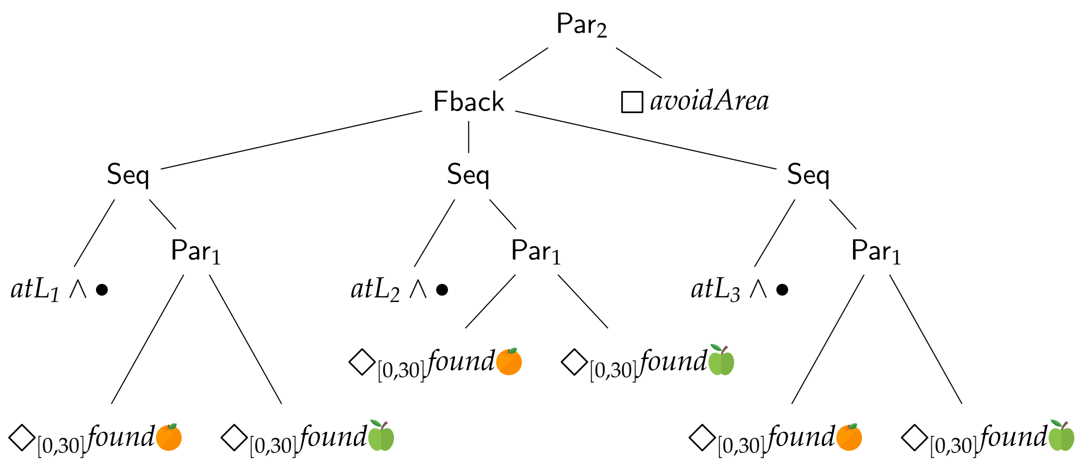

Par$_2$
Fback
$\square$ *avoidArea*
Seq
Seq
Seq
$atL_1 \wedge \bullet$
Par$_1$
$atL_2 \wedge \bullet$
Par$_1$
$atL_3 \wedge \bullet$
Par$_1$
$\Diamond_{[0,30]} found$🍊
$\Diamond_{[0,30]} found$🍏
$\Diamond_{[0,30]} found$🍊
$\Diamond_{[0,30]} found$🍏
$\Diamond_{[0,30]} found$🍊
$\Diamond_{[0,30]} found$🍏

Figure 5.2.: TBT to search for at least an apple or an orange at different locations.

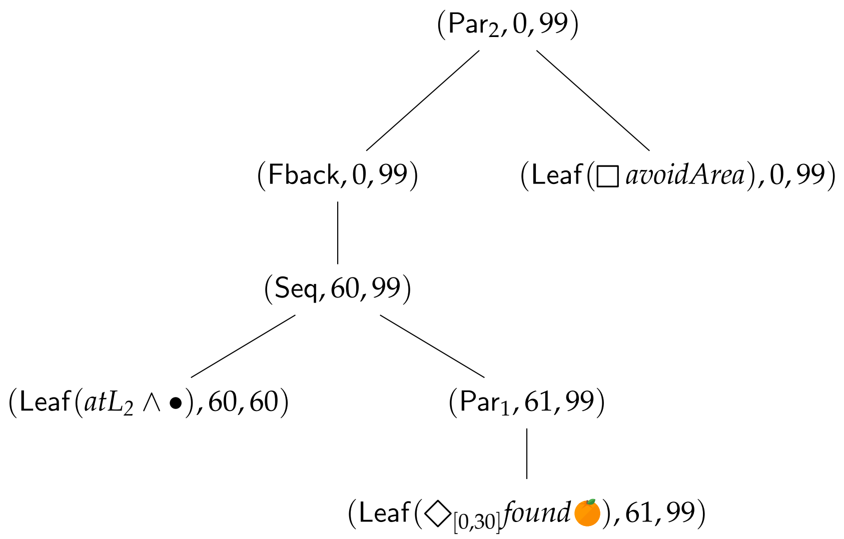

(Par$_2$, 0, 99)
(Fback, 0, 99)
(Leaf($\square$ *avoidArea*), 0, 99)
(Seq, 60, 99)
(Leaf($atL_2 \wedge \bullet$), 60, 60)
(Par$_1$, 61, 99)
(Leaf($\Diamond_{[0,30]} found$🍊), 61, 99)

Figure 5.3.: Segmentation graph for the TBT in Figure 5.2.

adjusting segment boundaries, and moving up to the parent node only in the worst case. The approach aims to avoid the costly search for boundaries during optimization using the guidance provided by a segmentation. We define the local repair strategy as follows.

Definition 5.1 (Local Repair)

Let M be a system model, σ be a trace, $G = (V, E)$ be a segmentation graph with $V' \subseteq V$, $f \in \{valid, loose\}$ indicate whether a valid transition to the next segment is required, and let $\mathsf{validTransition}_M(\sigma, \sigma_R)$ be a function that checks for a valid transition from the repaired trace segment to the original trace. We repair a trace locally, denoted $\mathsf{local}(M, \sigma, V', f)$, as follows:

$$\bigoplus_{(\mathcal{T},i,j)\in V'} \mathsf{repair}_{\sigma_R}(\mathcal{T}, M, \sigma[i:j], \mathsf{validTransition}_M(\sigma, \sigma_R) \vee f = loose)$$

where $\bigoplus$ merges the models, enforcing all constraints while minimizing the sum of the individual costs. Note that $\mathsf{repair}_{\sigma_R}$ is defined in Equation (5.2).

If no leaf node segments overlap, an efficient approach is to locally repair each leaf node and to update the trace when initial or last segment states change. However, for more complex cases like the segmentation graph in Figure 5.3 this is insufficient, as locally repairing $(\mathsf{Leaf}(\square\, avoidArea), 0, 99)$ can impact the satisfaction of the other leaf nodes. incremental repair accounts for that. The incremental repair strategy, shown below as Algorithm 1, repairs a given segmentation G. It initializes two sets: C for the repaired segments (Line 1) and L for the segments to be repaired (Line 2). Initially, L contains all leaves in G that violate its segment. Since segmentation may have overlapping violating segments that must be repaired together to avoid side effects, e.g., $(\mathsf{Leaf}(\square\, avoidArea), 0, 99)$ in Figure 5.3, $\mathsf{mergeOverlap}_G(L)$ merges these overlapping sets in L and removes entries if an ancestor node is already in the set. Afterwards one set is removed from L (Line 5) to be locally repaired next (Line 6 and 8). If there is a existing repair (Line 10), then $\mathsf{affectedLeaves}(\sigma, \sigma_R, V)$ checks if other leaves are affected by changes of σ_R. This also includes the check of valid transition between segments in case of a *loose* local repair (Line 8). For instance consider Example 5.3.1, in the beginning the set L is $\{\{(\mathsf{Leaf}(\square\, avoidArea), 0, 99)\}\}$ and if $\sigma_R[61:99] \neq \sigma[61:99]$ then the evaluation of $(\mathsf{Leaf}(\lozenge_{[0,30]} found$🍊$), 61, 99)$ might change. A repair is considered successful if only the leaves in l are affected (Line 12). Otherwise, a subsequent local repair must account for the affected leaves in the next iteration (Line 16). If the local repairs were not successful (Line 18), then the repair moves to its next common ancestor w.r.t. the elements in l. Finally, if there are no open repairs, i.e., $L = \emptyset$, we can read off σ_R using the set C (Line 18).

The incremental repair leverages information from the segmentation graph to avoid a costly exploration within the optimization model. Note, however, that a segmentation

Algorithm 1: incremental$_G(M, \sigma)$ repair strategy

```
Input: Segmentation $G = (V, E)$, system model $M$, and trace $\sigma$
Output: Repaired trace $\sigma_R$ or none
1  $C \leftarrow \{\}$                                                              ▷ Successful repairs
2  $L \leftarrow \{\{v | v \in V \land v = (\mathsf{Leaf}(\varphi), i, j) \land \sigma[i:j] \not\models \varphi\}\}$ ▷ Start with violating leaves
3  $L \leftarrow \mathsf{mergeOverlap}_G(L)$                    ▷ Groups overlapping leaves into one set
4  while $L \neq \emptyset$ do
5      $l \leftarrow L$.pop()                                ▷ Set of nodes in $V$ that must be repaired
6      $\sigma_R \leftarrow \mathsf{local}(M, \sigma, l, valid)$           ▷ If successful, avoids affecting other leaves
7      if $\sigma_R = none$ then
8          $\sigma_R \leftarrow \mathsf{local}(M, \sigma, l, loose)$        ▷ Could affect transitions to other leaves
9      if $\sigma_R \neq none$ then
10         $affected \leftarrow \mathsf{affectedLeaves}(\sigma, \sigma_R, V)$   ▷ Tracks leaf changes due to $\sigma_r \neq \sigma$
11         if $affected \setminus l = \emptyset$ then
12             $C \leftarrow C \cup (l, \sigma_R)$                                ▷ Successful repair!
13         else
14             $L \leftarrow L \cup affected$                          ▷ Accounts for affected leaves
15     else
16         $L \leftarrow L \cup \mathsf{getCommonAncestor}_G(l)$                   ▷ Move up the TBT
17     $L \leftarrow \mathsf{mergeOverlap}_G(L)$
18 return $\mathsf{compose}(C, \sigma)$
```

omits the different choices for a Fback node, which also means that the incremental repair does not take these into account. Fortunately, the approach presented in Chapter 4, which utilizes dynamic programming, provides “alternative segmentations” for every choice of a Fback node by simply “reading off” the table entries. We denote the set of segmentations that contains all choices of Fback nodes in a TBT $\mathcal{T}$ as $\hat{G}$.

Proposition 11. *If for all $G \in \hat{G}$, $\text{incremental}_G(M, \sigma) = \text{none}$ then there is no σ_R with $|\sigma_R| = |\sigma|$ for which $\sigma_R \models \mathcal{T}$.*

Theorem 12. *The incremental repair strategy is sound w.r.t. a segmentation and terminates.*

Proof. It is sound because if there are no successful local repairs, incremental repair moves up its segmentation graph and finally repairs the root node (see Line 16 in Algorithm 1). In the worst case, the MILP finds a trace σ_R that satisfies the constraints on the TBT and the model, while ignoring the relation to the original trace σ. It only returns none if there is no such trace σ_R with $|\sigma_R| = |\sigma|$. It terminates because mergeOverlap avoids having segments that affect each other over and over again. Assume $L = \{V_1, V_2\}$. There are three cases for an attempt to repair V_2: the repair was successful while no other node is affected (Line 12), the repair of V_2 was successful but there was an affection (Line 14), or there was no successful repair and incremental moves up the tree (Line 16). For the first case, the size of L is reduced since one element was removed from L (Line 5). For the second case, there must be a node v that is affected. Then, it either has an overlap with V_1 and according to Line 17 both sets are merged or there is no overlap and both sets V_1 and V_2 remain in L, while V_2 is increased by an additional element – converging to repairing the whole segmentation. The same holds for the last case. □

5.3.2. Landmark-Based Repair Strategy:

The incremental repair strategy can be complemented by an approach that deals with the disjunctive constraints encountered in the encoding of the repair problem (5.3). These constraints arise, for instance, when a fallback operator is encoded, or at a leaf node with a $\Diamond$ formula. Given a disjunctive formula $\bigvee_{j=1}^{m} \psi_j$, the landmark based strategy uses information from the original trace to select a candidate ψ_j that will be satisfied.

Formally, a candidate for a landmark is a minimal set of propositions that is sufficient to satisfy the TBT specification. This simplifies the optimization problem to a linear program for the L_1 and Hamming distance, as it no longer requires integer or binary variables, and only linear constraints remain[2]. Outside the optimization problem, we can also use the robust semantics of TBTs to rank the candidates, i.e., we interpret robustness of APs as a heuristic. However, in Section 5.4, we will see that the repair provides a fast solution for the repair and improves upon that, similar to an anytime algorithm.

[2]Using the robust semantics of TBTs requires the Big-M encoding of the absolute value function, which in turn adds binary variables.

Example 5.3.2. Given a leaf node $\Diamond_{[0,10]} atL_1$ with $f_{atL_1}(x) = x - 2.5$, $\mathsf{encode}_{\sigma_R}(\mathcal{T})$ unfolds into a disjunction of atL_1 at the next positions. The trace shown in Figure 5.4 violates the property because there is no value of x at any position i in the trace where $f_{atL_1}(x)$ yields a positive value. Landmark-based repair will pick a candidate to solve the disjunction. Here, eleven candidates for landmarks exist. The best candidate w.r.t. the robustness value of atL_1 is at position $i = 7$ with value close to -0.5 (highlighted by the green circle). The landmark is encoded by $\sigma_R(8) \models atL_1$. △

Algorithm 2 provides an iterative repair strategy based on landmarks. The repair receives a segmentation, a system model, and the violating trace. In Line 3, the set of candidates is computed and ordered. It then takes the most promising candidate (Line 5), sets an upper bound on the repair cost (Line 6), encodes the repair (Line 7), and checks if the current landmark returns a better repair (Line 8). Note, the constraints on the system model and cost function remain unchanged throughout the iteration (Line 4). Therefore, the LP can be efficiently updated by simply removing the previous landmark and adding the next one, avoiding the need to rebuild the LP from scratch. For the experiments, two optimizations were implemented but are omitted here for brevity. First, candidates are computed on-the-fly instead of upfront as robustness maximizes. Second, candidates are ranked by robustness and explored while maintaining a minimum distance w.r.t. the position of previously tested candidates. This distance starts large and gradually decreases whenever no candidates are left within this distance. Once the distance reaches 1 and no further candidates are available, the repair terminates. For instance, considering Example 5.3.2 where position $i = 7$ is chosen as first landmark, a distance of two excludes Positions $5, 6, 8, 9$ and allows to explore Position 3 next. So far, the correct distance and its rate of decrease are user-defined parameters.

Algorithm 2: $\mathsf{landmark\text{-}based}_G(M, \sigma)$ repair strategy

```
   Input: Segmentation $G$, system model $M$, and trace $\sigma$
   Output: Repaired trace $\sigma_R$ or none
1  $\sigma_R \leftarrow$ none
2  $J_{upper} \leftarrow \infty$
3  $candidates \leftarrow$ computeCandidates$(G, \sigma)$ ▷ List of landmarks ordered by heuristic
4  while $candidates \neq \emptyset$ do
5      $landmark \leftarrow candidates$.pop()
6      $improve \leftarrow J(\sigma, \sigma_R) < J_{upper}$          ▷ Upper bound on cost of repair
7      $\sigma'_R \leftarrow$ repair$_{\sigma_R}(true, M, \sigma, landmark \wedge improve)$   ▷ $landmark$ replaces TBT
8      if $\sigma'_R \neq none$ then
9          $\sigma_R \leftarrow \sigma'_R$;
10         $J_{upper} \leftarrow J(\sigma, \sigma'_R)$;
11 return $\sigma_R$;
```

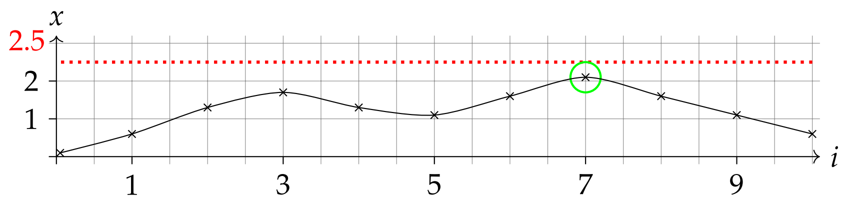


Figure 5.4.: The values of x for several positions within a trace. Given $\Diamond_{[0,10]} atL_1$ with $f_{atL_1}(x) = x - 2.5$, position $i = 7$ is a good candidate for a landmark.

Discussion Both strategies can be combined to efficiently solve complex repairs. The incremental repair allows to divide the trace into smaller segments, each of which can be repaired locally. When a local property contains multiple disjunctions, e.g., when using an unbounded $\Diamond$, the optimization becomes more challenging. In such cases, we can use the landmark-based repair. If the combination of both strategies fails, refining the TBT specification is a viable option.

The closer the violating trace is to satisfying the specification, the more effective segmentation and landmarks are as starting points for the repair. Unfortunately, the inverse does also hold. For instance, if the TBT consists of a sequence of two nodes, with the segmentation assigning a very short segment to the first node – potentially too short for a repair – and the remainder of the trace to the second node, incremental repair will converge to the full encoding. In such cases, starting with the full encoding or even finding a new unrelated trace would be faster.

5.4. Empirical Evaluation and Case-Studies

This section presents two case studies: the robot search task shown in Figure 5.2 and the automated landing of a UAV on a ship. The first case study illustrates the impact of different cost functions, while the second showcases incremental repair and compares it to landmark-based repair. All experiments were run on a single 16-core machine with a 2.50 GHz 11th Gen Intel(R) Core(TM) i7-11850H processor with 32 GB RAM. The algorithms are implemented in Python using Gurobi[3] as optimizer. Segmentations were obtained using the approach from [146].

→Fig. 5.2, p. 78

5.4.1. Robot Search Task

Using the robot search task introduced in Figure 5.2, we demonstrate two different cost functions using incremental repair. The trace we used has a best segmentation in which both the location and one of the fruits were narrowly missed. Also, the restricted area that was meant to be avoided was breached. The original trace, consisting of 540 entries, was reduced to 68 entries through subsampling by [146], which computed the best

[3]`https://www.gurobi.com/`: Gurobi Optimizer version 11.0.0 build v11.0.0rc2

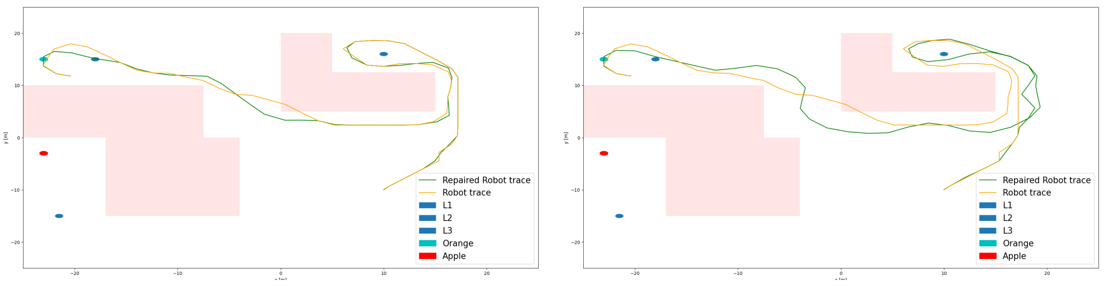


Figure 5.5.: Repair using different cost functions: on the left using $L1$ and on the right using combination W with weights ($L1 : 0.01, R : 0.99$).

segmentation in under a second. Figure 5.5 depicts the results of the incremental repair: on the left, using *L1* as cost function, and on the right, using a weighted combination W of *L1* and robustness R with weights 0.01 and 0.99, respectively. It took 6s to repair using *L1* and 21s using W. Both repairs reach the location $L2$ and find a fruit (top left). Note that the repair using W provides larger separation from the restricted area compared to *L1* but still resembles the original trace. The runtime of the repair is mostly impacted by $\square$ *avoidArea*, as it relies on the whole trace. Also, AP *avoidArea* adds constraints to keep points outside the region and maintain a minimum distance to the corners of the restricted areas. This can be avoided by a syntactic reformulation of the TBT, optimizing it to provide segmentation graphs that are easier to repair. For instance, by moving the $\square$ *avoidArea* invariant into the individual leaf nodes. Yet, this is not the scope of this thesis.

5.4.2. Automated Ship Deck Landing

Landing on a ship deck is a challenging task, wherein various landing aids and maneuvers need to be carefully selected [141, 142]. The benefits of TBT segmentation for an automated lander were previously discussed in Chapter 4. The depicted TBT in Figure 5.1a is a simplified version of the used TBT $\mathcal{T}$ in Figure 4.7, where all landing maneuvers are used in the fallback node: *Straight-in*, *Lateral*, *45-Degree*, and *Oblique*. Each of them is represented by a sequence node. Here, we consider the *45-Degree* sequence node: $\mathsf{Seq}([\mathsf{Leaf}(\Diamond\ \mathit{diagonalBehind}(s)), \mathsf{Leaf}(\square_{[0,5s]}\ (\mathit{diagonalBehind}(s) \wedge \mathit{alignedHeading}(s))), \mathsf{Leaf}(\mathit{alignedHeading}(s)\ \mathcal{U}\ \mathit{aboveTouchdown}(s) \wedge \bullet)])$ where s contains the position, velocity, and heading of the ship and the UAV. We abbreviate this sequence by $\mathsf{Seq}([\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3])$, e.g., $\mathcal{T}_1 = \mathsf{Leaf}(\Diamond\ \mathit{diagonalBehind}(s))$. The AP *diagonalBehind* represents that the UAV is diagonally behind the ship, *alignedHeading* ensures that the UAV has a heading that is aligned with the ship heading, *aboveTouchdown* represents that the UAV is above the touchdown point of the ship. The other sequence nodes beneath the fallback are similar, only the target position and the prescribed heading change. The final leaf node also given in Figure 5.1a is common for all behaviors and

→Fig. 5.1a, p. 72
→Fig. 4.7, p. 68
→Fig. 5.1a, p. 72

specifies the descend property $\mathsf{Leaf}(\Diamond\, \mathit{descended}(s))$, where $\mathit{descended}(s)$ states that the UAV landed on the touchdown point. We abbreviate this node by $\mathcal{T}_4$. For more detailed information on the TBT $\mathcal{T}$, we refer to [49]. Next, we repair a violating trace from [49] using segmentation information, showing the *45-Degree* landing maneuver being the closes to satisfy the specification. The repair only impacts the UAV's states.

Incremental Repair All segmentations were computed in under 30s. The original trace had a mission-time of 126s and a length of $25,349$, subsampled to 1014 by the segmentation. Nonlinear helicopter dynamics were simplified into four independent integrator chains similar to Example 5.1.1: one for each of the three inertial axes (x, y, z) and one for the heading [147]. The repair works on the subsampled trace using $L1$ as the cost function. The segments of the leaf nodes are: $(\mathcal{T}_1, 0, 265)$, $(\mathcal{T}_2, 266, 306)$, $(\mathcal{T}_3, 307, 581)$, and $(\mathcal{T}_4, 582, 1013)$. Segments $\mathcal{T}_1, \mathcal{T}_2$, and $\mathcal{T}_3$ are violating. As a reference for a full MILP encoding of the entire trace, we encoded a simplified landing as $\mathsf{Leaf}(\mathit{land})$ using the formula $\mathit{land} = \Diamond(\mathit{diagonalBehind}(s) \wedge \bigcirc(\mathit{stayPos} \wedge \Diamond_{[6,\infty]}(\mathit{alignedHeading}(s) \wedge \mathit{aboveTouchdown}(s) \wedge \Diamond \mathit{descended}(s))))$, i.e., we replaced $\mathcal{U}$ with $\Diamond$ by omitting the left side. We chose $\mathsf{Leaf}(\mathit{land})$ because a full MILP encoding, i.e., directly solving the root node, caused an out-of-memory error. Note that this allows a baseline comparison to standard approximations techniques supported by the optimizer. We use Gurobi with its default parameters, which include features such as root relaxation and presolve. Experimental results in Table 5.7 show that the reference full MILP repair of $\mathsf{Leaf}(\mathit{land})$ does not scale well for longer trace – it took over 2000s – while incremental repair took less than a minute. The detailed steps of the incremental repair show that ensuring *valid* transitions did not save time for Steps 1, 4, and 6 (8.87s were unnecessary spend), but Step 3 saved presumably around 11s. The reason for this is that the last state of the segment must change too radically to satisfy the specification and only when both leaf nodes are encoded (Step 3), a matching state can be found. The evaluation shows that getCommonAncestor (Line 16 in Algorithm 1) was never invoked; only the affected leaves needed to be accounted for. This avoided the need for a costly repair. The result of the repair is shown in Figure 5.1b and Figure 5.6. The plots show that the segmentation provides a good way to decompose the specification as only small adjustments of the positions were required to satisfy $\mathcal{T}$. The most significant changes were due to $\mathcal{T}_3$ that requires an aligned heading (see Timesteps 350 to 600 in Figure 5.6).

→Ex. 5.1.1, p. 73

→Alg. 1, p. 80

→Fig. 5.1b, p. 72

Landmark-based Repair To illustrate the impact of disjunctions, we use incremental repair while omitting $\bullet$ from the last leaf node of the *45-Degree* sequence node. Specifically, we consider $\mathsf{Leaf}(\mathit{alignedHeading}(s)\ \mathcal{U}\ \mathit{aboveTouchdown}(s))$ instead of $\mathsf{Leaf}(\mathit{alignedHeading}(s)\ \mathcal{U}\ \mathit{aboveTouchdown}(s) \wedge \bullet)$. Therefore, the optimizer must determine the optimal position to satisfy $\mathit{aboveTouchdown}(s)$, while $\bullet$ constrains it to the last position of this segment. As a result, the computation time increases from 29.62s as in Table 5.7 to 278.32s using incremental repair. Figure 5.8 illustrates this effect, comparing it to the results from the iterative landmark-based repair, where dots represent when solutions were found. The time limit was set to 300s. The landmark-based repair finds

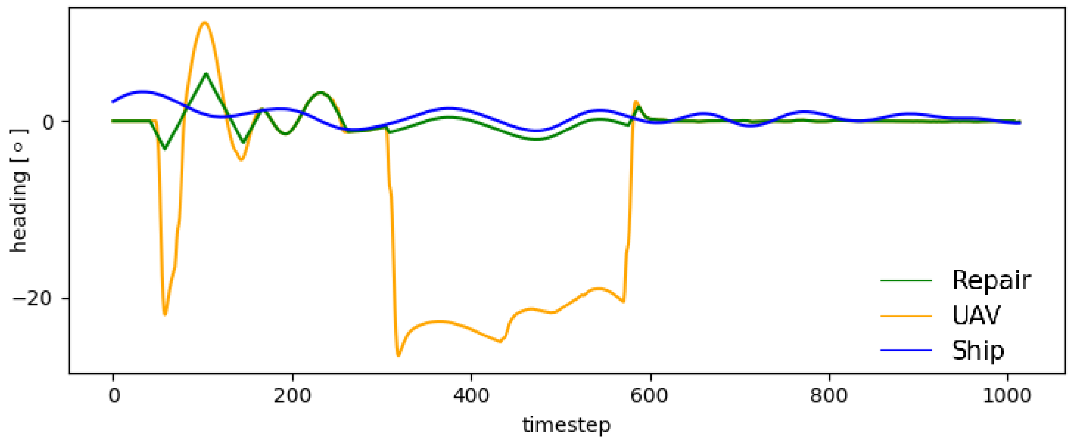


Figure 5.6.: Repair to satisfy the heading constrains in $\mathcal{T}_3$.

Formula	L	**Time (s)**
Leaf$(land)$	$\{\{(\text{Leaf}(land), 0, 1013)\}\}$	2200.40
$\mathcal{T}$	$\{\{(\mathcal{T}_1, 0, 265)\}, \{(\mathcal{T}_2, 266, 306)\}, \{(\mathcal{T}_3, 307, 581)\}\}$	**29.62**
Step 1	$l = \{(\mathcal{T}_3, 307, 581)\}, f = valid$	*3.72* ✗
Step 2	$l = \{(\mathcal{T}_3, 307, 581)\}, f = loose$	*3.86* ✓
	$\{\{(\mathcal{T}_1, 0, 265)\}, \{(\mathcal{T}_2, 266, 306)\}, \{(\mathcal{T}_3, 307, 581), (\mathcal{T}_4, 582, 1013)\}\}$	
Step 3	$l = \{\{(\mathcal{T}_3, 307, 581), (\mathcal{T}_4, 582, 1013)\}\}, f = valid$	*10.93* ✓
	$\{\{(\mathcal{T}_1, 0, 265)\}, \{(\mathcal{T}_2, 266, 306)\}\}$	
Step 4	$l = \{(\mathcal{T}_2, 266, 306)\}, f = valid$	*0.60* ✗
Step 5	$l = \{(\mathcal{T}_2, 266, 306)\}, f = loose$	*0.85* ✓
	$\{\{(\mathcal{T}_1, 0, 265), (\mathcal{T}_2, 266, 306)\}\}$	
Step 6	$l = \{(\mathcal{T}_1, 0, 265), (\mathcal{T}_2, 266, 306)\}, f = valid$	*4.55* ✗
Step 7	$l = \{(\mathcal{T}_1, 0, 265), (\mathcal{T}_2, 266, 306)\}, f = loose$	*5.11* ✓

Table 5.7.: Trace repair results with intermediate steps of the incremental repair. The Time column includes setting up the model and solving it. The results show that incrementally repairing $\mathcal{T}$ is more efficient than the reference repair of Leaf$(land)$. ✗ represents infeasible runs whereas ✓ represents successful runs.

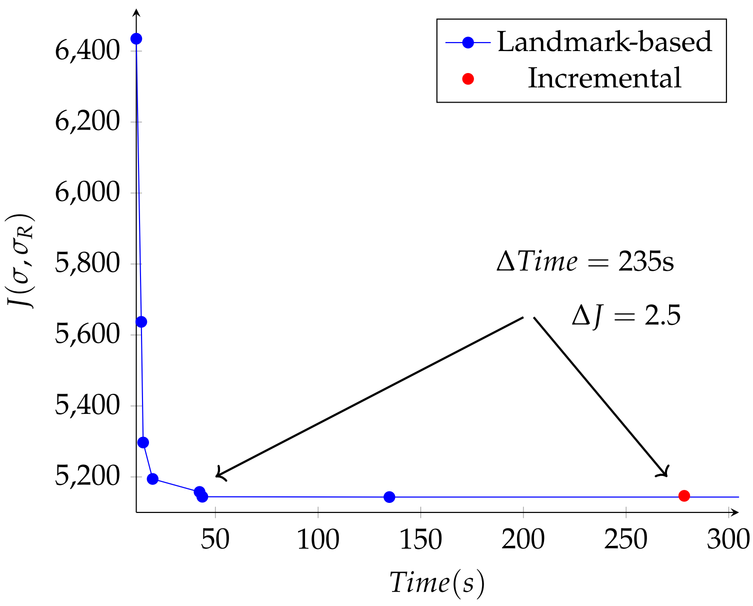


Figure 5.8.: Comparison of repair strategies.

its first solution after just 12s and continues to improve upon it. Within approximately 40s, a repair is achieved that is comparable to the one found by the incremental repair while saving 235s. Figure 5.9 shows the repair. Note that, without the •, the leaf node $\mathcal{T}_3$, which contains the *aboveTouchdown* proposition, can be satisfied earlier (around Timestep 450), thereby preventing the need of repairing heading thereafter. Additionally, the repair chooses the same position to repair the *diagonalBehind* proposition in $\mathcal{T}_1$ as in Figure 5.1b. As next experiment, we applied the same TBT specification, but instead of using the subsampled trace, we encoded the full original trace that contains 25,348 entries. We were able to successfully identify a repair within 362s.

→Fig. 5.1b, p. 72

5.5. Summary

This chapter presented methods for repairing traces of CPS that violate a given TBT specification. While a MILP could theoretically solve the problem, our experiments show that this is too expensive in practice. To address this, we introduced an incremental repair strategy that uses the segmentation information from a TBT monitor to repair violating segments locally. Additionally, we presented a landmark-based repair strategy, an iterative approach that avoids MILP encoding of the TBT by using landmarks. The landmarks allow us to formulate the repair as a linear program. Our experiments demonstrate that the two strategies make it possible to repair traces of more than 25,000 entries in under ten minutes, while the full MILP runs out of memory.

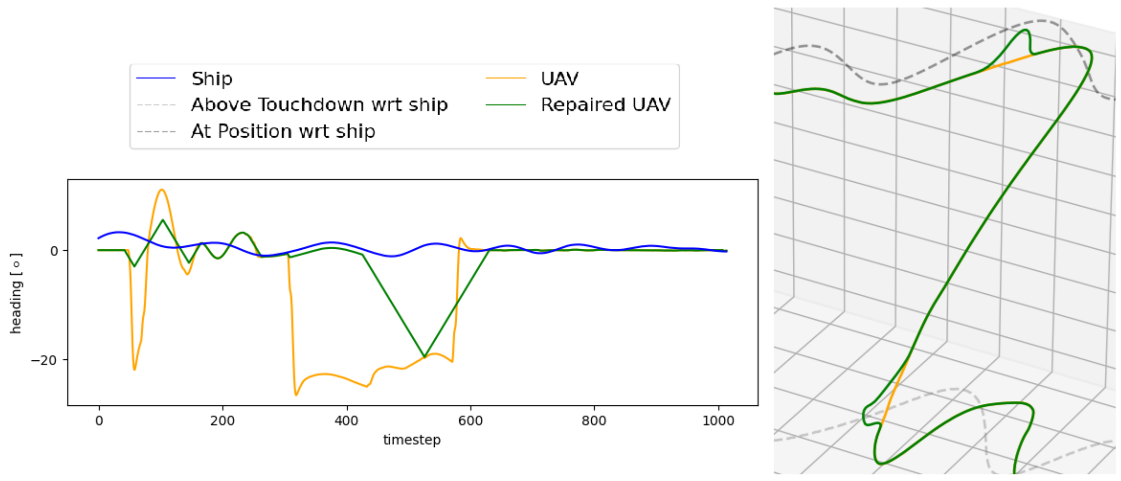


Figure 5.9.: The repair chooses an earlier *aboveTouchdown* when omitting ●.

Part III

Online Monitoring: Tools & Applications

Hot Take:
Expressiveness is not the main RV-CPS barrier;
domain-specific tool support and qualification are.

Chapter 6

Safeguarding Machine Learning

In this chapter, we present an adapter that simplifies the integration of RTLola into robotic applications running within the Robot Operating System (ROS). ROS is an open-source framework that provides essential middleware functionality and abstractions to manage the complexity of scaling autonomous robotic missions. For instance, ROS is designed with distributed computing in mind, where essential robotic functions are executed in *nodes* that receive data from other nodes and pass their results along. Such an essential function can be low-level as reading sensors but also more high-level such as navigation and object recognition. For a safe execution, ROS has basic monitoring capabilities. While sufficient for low-level tasks, these capabilities are too limited to ensure the safety of high-level autonomous functions, which often require monitoring complex temporal system properties with asynchronous inputs. As a result, users are typically forced to implement custom monitoring solutions using general-purpose code, increasing the likelihood of errors. To mitigate this risk, runtime verification (RV) offers a formal, lightweight alternative where complex properties are specified concisely using a formal language, from which executable monitors are automatically generated. In this chapter, we use the stream-based language RTLola for specifying real-time and asynchronicity properties.

We present a ROS adapter for RTLola[1]. This adapter enables seamless integration of a monitor generated with the RTLola framework [125] by wrapping it as a ROS node, eliminating the need for any manual adjustments. The key features of the adapter are:

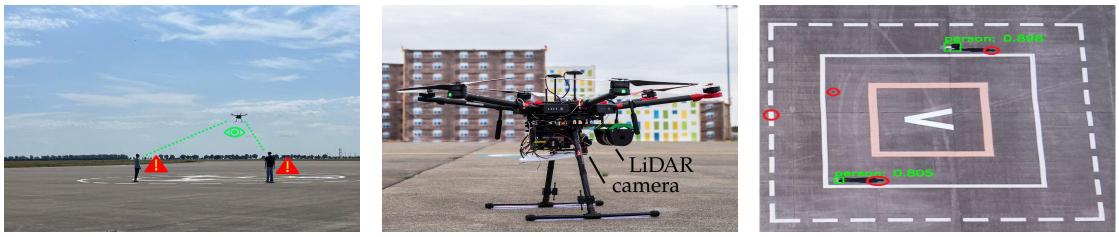
- *automatic input mapping*: inspecting the RTLola specification and automatically subscribing to ROS topics to match required streams;
- *automatic output mapping*: inspecting the RTLola ROS topic and automatically publish corresponding streams;

[1] `https://github.com/DLR-FT/RTLola-ROS2-Adapter`

- *supports ROS service*: inspection of RTLola specification and stream mapping for direct request-reply communication;
- *implemented in Rust*: safe and efficient, compatible with any ROS 2 version;
- *tool validated in flight*: used for cross-validation of bounding boxes given by machine-learning-component and blobs detected in LiDAR image using traditional computer vision techniques.

Alongside the technical details presented in the subsequent sections, we also provide information on the experimental flight test, starting with the use-case.

Experimental Use-case We deployed the adapter for safeguarding a machine-learning (ML) component running onboard of an unmanned aircraft. This ML component is responsible for detecting people near a designated landing site. Figure 6.1a illustrates the scenario, where mannequins simulate people standing on a vertiport – the intended landing site for the hovering unmanned aircraft. Figure 6.1b depicts the unmanned aircraft equipped with a camera and a LiDAR. In the background, the "container city" is shown. Some mannequins were positioned within this city to provide occlusion scenarios. The aircraft's perspective is shown in Figure 6.1c, highlighting both correct detections (green) and challenging conditions that may lead to false positives (red) of the ML component. In this experiment, we use the adapter to integrate an RTLola monitor that cross-validates ML-based detections against objects in LiDAR depth images using a traditional Blob detector to reduce the false positive rate. In the following, we focus on the monitor integration, while referring to [47] for the experimental results where also other properties such as geofencing were considered for monitoring the operational design domain.



(a) Task: Safeguard detecting people at landing site.

(b) Compare detected persons with objects in a LiDAR depth image.

(c) True positives are shown in green, potential false positives in red.

Figure 6.1.: Experimental setup: landing site with mannequins, and scenario.

6.1. An RTLola Specification for Cross-Validation

The stream-based specification language RTLola is introduced in more detail in Section 3.2. As a reminder: `Output` streams aggregate, combine, and transform input streams to compute valuable statistics about the system. Special output streams called `trigger` streams can be declared to specify a Boolean verdict about the systems health based on these statistics. Output streams can be generalized to sets of stream instances that are uniquely identified by a set of parameters. Such parameterized output streams include a `spawn` and `close` condition, specifying when and how new stream instances are created and removed.

→Sec. 3.2, p. 38

Experimental Setup The specification used in the experiments[1] is given in Listing 6.1. The specification tests whether a blob exists within a given bounding box, which should be the case for an actual person. First, in Line 2, four input streams are specified that characterize a single bounding box in the camera image by the `x` and `y` coordinate of its upper left corner and its width and height. Similarly, in Line 4, the input streams `blob_x`, `blob_y`, and `blob_r` describe the `xy`-position and the radius `r` of a single circular blob in the LiDAR image. Since the images of the camera and the LiDAR have different resolutions and only partially overlap w.r.t. their field of views, the two output streams `blob_x_in_cam` and `blob_y_in_cam` transform the coordinates of the detected blob in the LiDAR image into corresponding coordinates in the camera image (Lines 6 and 7). Note that due to lens curvature, the transformation is inherently non-linear. Furthermore, the solution depicted here represents a sub-optimal solution that is valid only within a specific altitude band. The next output stream in Line 9 is parameterized over the `x` and `y` coordinates of blobs and their radii `r`, specified by the parameter list followed after its stream name. The `spawn` condition of the stream (Line 10) instantiates these parameters with the (transformed) coordinate and radius of a blob, effectively creating an instance of the stream for each known blob. Each stream instance then validates whether the current bounding box contains the blob the stream instance corresponds to, specified in the `eval` clause of the stream (Line 11). Since memorizing every blob received at run-time is infeasible, we use the `close` condition: a stream instance (and with that a blob) becomes irrelevant after 200 milliseconds, using the instance's local clock starting upon creation (Line 12). Lastly, the `validate` stream in Line 14 tests whether the current bounding box intersects with any blob by aggregating over all instances of the `check` stream[2]. If such an instance exists, the stream evaluates to true.

[1]The specification together with a toy trace can be tested within a playground: https://rtlola.cispa.de/playground/tutorial

[2]In practice, bounding boxes are sent with a delay to not only consider past blobs.

```
// Service inputs: incoming bounding boxes
input bb_x, bb_y, bb_w, bb_h: Float64, Float64, Float64, Float64
// Subscribed inputs: detected blobs in depth image
input blob_x, blob_y, blob_r: Float64, Float64, Float64
// Transform depth image coordinates to camera coordinates
output blob_x_in_cam := -0.01*blob_y**3.0 + 1.81*blob_y**2.0 - 66.24*blob_y + 1765.75
output blob_y_in_cam := 13.91*blob_x - 6627.13
// Whenever a new bounding box arrives compare it to all recent blobs
output check(x, y, r)
   spawn with (blob_x_in_cam, blob_y_in_cam, blob_r)
   eval with x - r > bb_x ∧ x + r < bb_x + bb_w ∧ y - r > bb_y ∧ y + r < bb_y + bb_h
   close @Local(200ms) // recent is defined as 200ms
// Service response
output validate := check.aggregate(over_instances: all, using: ∃)
```

Listing 6.1: An RTLola specification matching bounding boxes to blobs.

6.2. ROS 2 Environment

ROS is a middleware that enables the execution of a distributed system made up of nodes, where each *node* typically handles a specific task in the robotic application. For example, one node may handle a sensor driver that collects sensor readings, another node may manage control tasks, and a third node may focus on monitoring. ROS supports different ways of communication between nodes. Most prominent are topics and services.

A *topic* is a named communication channel over which structured messages of a specific type are sent. To send data on a topic, a node must *publish* messages to it, while to receive, a node must *subscribe* to the topic. Multiple nodes can simultaneously publish to the same topic, and multiple nodes can subscribe. Each topic is strictly associated with one message type, and all publishers and subscribers must agree on this type. Listing 6.2 provides the interface file for the "blob" topic published by the Blob detector: a blob's xy-position and radius.

A *service* is a named communication channel that allows one node to request a specific response from another node. To initiate a service, a node must send a *request* message (client), while the node providing the service must process the request and sends a *response* (server). Each service is associated with a request and response message type, which must be defined beforehand, ensuring both the client and server nodes agree. Unlike topics, services operate in a one-to-one manner. A service call is blocking, meaning the client waits for a response before continuing. Listing 6.3 provides the interface file for the RTLola service that validates bounding boxes. The request and the response are separated by "---": the request specifies the xy-position, width, and height of the bounding box, while the response returns a single Boolean which indicates a matching blob.

Quality of Service (QoS) defines how data is exchanged between nodes. Some example settings include the *history* policy, which determines whether to keep only a limited number of recent messages (“keep last”) or all messages (“keep all”), with the *queue size* applying only to the former; *reliability*, which defines whether messages are delivered with possible loss (“best effort”) or guaranteed with retries (“reliable”); and *durability*, which controls if messages are persisted for late subscribers (“transient local”) or discarded after publishing (“volatile”).

ROS primarily support C++ and Python. C++ is widely used for performance-critical and hardware-related tasks, while Python is popular for scripting, rapid prototyping, and creating simpler nodes. Therefore, most core ROS components are written in C++. ROS comes with its own build system using CMake/Catkin, which makes the use of other compiling languages building on other build systems challenging, e.g., Rust using Cargo.

```
float64 x // x-position
float64 y // y-position
float64 r // radius
```

Listing 6.2: “blob” interface file

```
float64 x // x-position
float64 y // y-position
float64 w // width
float64 h // height
--- // Separates request and response
bool validate // is true positive
```

Listing 6.3: “bb” service interface file

Experimental Setup The unmanned aircraft was a hexacopter with a maximal takeoff weight of 15.5 kg that followed a preprogrammed waypoint mission flying above the mannequins. To guarantee safety, a human remote pilot had always the possibility to takeover control when observing unintended drone behavior. As hardware payload, the drone carries a Pixhawk 4 flight controller hardware, a Jetson jAi Go 2400 camera, an Ouster OS0 LiDAR, and a Nvidia Jetson AGX Orin companion computer. The companion computer is connected via 5 GHz Wlan to a ground control station for controlling the experiment and visualizing the validated bounding boxes.

Software-wise, the Pixhawk 4 runs the PX4 autopilot software and the Nvidia Jetson executes a ROS environment with multiple nodes. As PX4 and ROS provide a deep integration that directly allows to exchange information[3] numerous PX4 messages are directly available as ROS topics for the companion computer. This includes for instance information about battery, actuators, and sensor readings such as position[4]. The central ROS nodes for this experiments running on the companion computer were a node that publishes the camera images, a node for publishing the LiDAR depth images, a node that runs a OpenCV Blob detector on the depth images, an RTLola monitor, and DLR’s UAVision that runs an ML-based object detector and streams verified detections to a ground control station. The ML-component ran at $\sim$ 3 Hz and the Blob detector ran at around $\sim$ 20 Hz. UAVision is subscribed to the images from the camera publisher.

[3]https://docs.px4.io/main/en/ros2/user_guide.html
[4]https://github.com/PX4/px4_msgs

When UAVision receives a camera image, it first runs the ML-based detector to obtain bounding boxes. It then uses a service provided by the RTLola monitor node to send each detected bounding box for cross-validation with the Blob detections subscribed to by the monitor If such a corresponding blob exists, the bounding box is validated; otherwise, it is falsified. This information, along with the bounding box, is transmitted to the ground control station. Validated bounding boxes are depicted in cyan, while falsified ones are shown in pink.

6.3. Tool: A ROS Adapter for RTLola

The adapter [5] is written in Rust to ensure seamless integration with RTLola. It assumes a running ROS 2 workspace. To build and integrate the monitor for a given RTLola specification, the user simply needs to execute:

`cargo run - <specification>`.

Figure 6.2 shows the corresponding pipeline stages. During the "Generate" stage, the adapter collects information about available topics using ROS command-line tools. It starts by executing `ros2 topic list` to retrieve a list of currently available topics. Then, `ros2 interface show <topic/service>` is used to obtain the corresponding interface definitions. Finally, QoS details are accessed via `ros2 topic info <topic/service> -verbose`. This gathered information is used to generate Rust code instances of predefined templates, enabling topic subscription. These templates utilize the Rust crate `r2r`[6], which avoids the integration with the ROS build system and instead relies solely on `cargo`. By default, the subscriptions matches the QoS settings of the publisher to ensure compability. Whether the monitor publishes or provides a service depends on the presence of the `RTLolaOutput` topic and the `RTLolaService` service interface in the running ROS workspace, respectively. If either interface is available, corresponding templates are instantiated to implement a publisher or a service server. Note that this stage is independent of the provided RTLola specification – the mapping of input/output RTLola streams are derived in a later stage. Further, a configuration file allows for QoS customization and includes a flag to "freeze" this stage's output, preventing overwriting of already generated code. During the next stage, the generated template instances and the monitor code are compiled. Finally, the wrapped RTLola monitor is executed as ROS node. An executed RTLola monitor first generates a mapping between RTLola input/output streams and corresponding topic members. This mapping adheres to a naming convention: <*package_name_lowercase*>_<*member*>, which is also applied during template instantiation. For example, the input stream `input adc_a0: Float32` maps to the float member `a0` of topic `adc`. Moreover, fixed-length arrays in topics are unrolled, e.g., a topic *Gps* with member `float64[2] x` results in two streams: `gps_x_0` and `gps_x_1`. If any required stream is missing, the adapter will raise an error

[5] https://github.com/DLR-FT/RTLola-ROS2-Adapter

[6] https://github.com/sequenceplanner/r2r

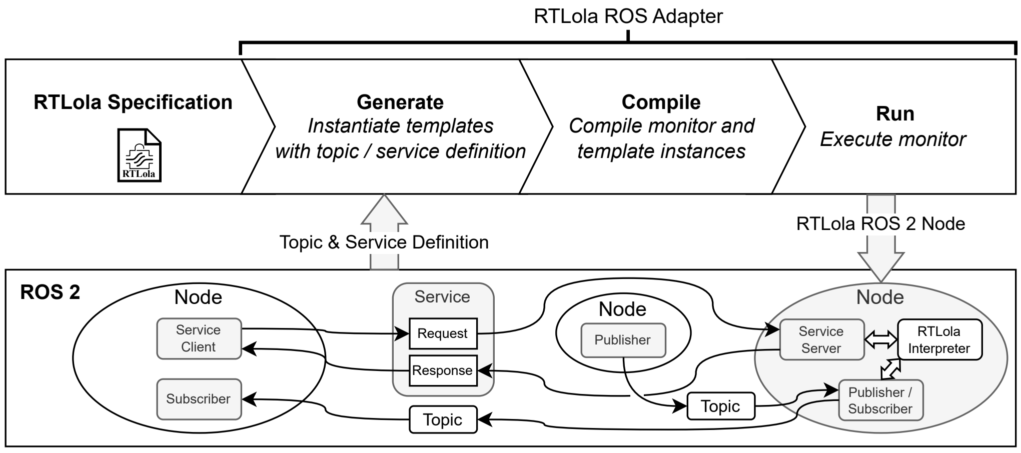


Figure 6.2.: Execution pipeline of the ROS adapter for RTLola

to notify the user. Note that the adapter only needs to be recompiled when topics or services are added or modified. Limitations: When responding to a service request, the adapter uses the immediate output stream values after processing the inputs. As a consequence, outputs of periodic streams may remain unchanged in the response. Additionally, topic members of unbounded array types are currently not supported and are therefore ignored.

Experimental Results Prior to the flight test, the adapter was used during ground tests. The code generated during these tests was frozen by setting the respective flag to avoid code generation during flight tests. The PX4 autopilot provides over 50 topics. The adapter generated code for all necessary topics, so the user can focus fully on writing the specification while required topics are subscribed automatically

The provided RTLola service simplified the implementation of the cross-validation called by UAVision. If these requests were implemented using topics, additional boilerplate code would be required to handle the request-response pattern manually. Using a service avoids this overhead. Listing 6.4 shows a code snippet demonstrating how to integrate the service. Line 4 and 5 retrieve an image, detect persons, and create requests. For each detected bounding box, a blocking request `req_bb` is sent and handled asynchronously using a "future" response (Line 7). This response, once available, is passed to a callback function (Line 8), responsible for forwarding the result to the ground control station.

The experiment also demonstrates that runtime verification improves the system's performance. Figure 6.3 illustrates a scenario where detections were successfully validated and could therefore be trusted. In contrast, Figure 6.4 shows an example of a false positive detection that was identified as untrustworthy and subsequently discarded by the system.

```
def callback(request, response): # callback function
  ...
while True:
  frame = self.camera.read() # reads camera frame
  req_bbs = self.ml.detect(frame) # detection and creation of requests
  for req_bb in req_bbs: # iterates over bounding box requests
    self.ros_client.call_async(req_bb).add_done_callback(
      functools.partial(self.callback, req_bb)) # bind request to callback
```

Listing 6.4: Excerpt of Python code demonstrating how the RTLOLA service simplifies monitor integration into UAVISION, avoiding explicit subscription handling.

6.4. Summary

This chapter introduced a ROS adapter for RTLOLA, designed to simplify the integration of RTLOLA into ROS-based robotic applications. The adapter automatically maps ROS topics to RTLOLA streams and generates the required code, thereby reducing developer effort when working with complex systems involving numerous topics.

In real-flight experiments, the adapter successfully generated code for all required topics, significantly reducing user effort given the large number of available topics. Furthermore, we demonstrated that providing a service interface by the monitor simplifies the usage of monitoring verdicts by eliminating manual request-response handling. Finally, the adapter was used to validate detections from a machine-learning component by cross-checking them with LiDAR-based observations, demonstrating RTLOLA's ability to safeguard autonomous decision-making against false positives. The chapter illustrates how a tool such as the adapter streamlines the integration of monitors into existing systems, thereby reducing integration costs and making adoption more accessible.

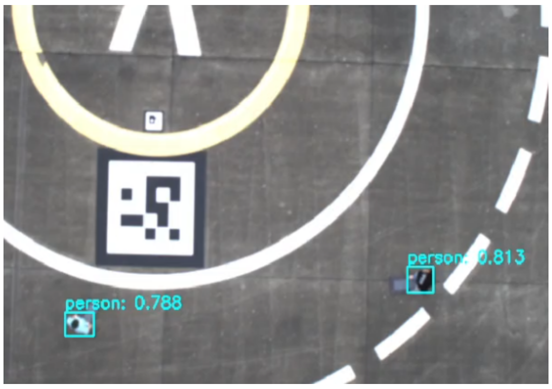


Figure 6.3.: The monitor validated two correct human bounding boxes (cyan).

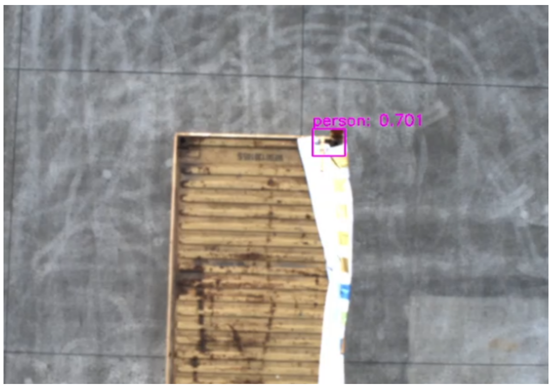


Figure 6.4.: Monitor flags a false positive (pink) with no matching blob.

Chapter 7

Safe Operation Monitoring

In this chapter, we introduce *safe operation monitoring* (SOM) in the context of SORA. SOM applies across all levels of the hierarchy as presented in Section 2.2. For illustration, we present one mission-level property and one operation-level property in detail. Specifically, we focus on flight path monitoring as a mission-level property and geofencing as an operation-level property.

→Sec. 2.2, p. 20

Geofencing is a virtual spatial boundary that a UAV is not allowed to cross. In the context of SORA, it is used to ensure the UAV remains within the Operational Volume, which consists of the Flight Geography and the Contingency Volume as depicted in Figure 7.1. Upon crossing the Flight Geography, a contingency procedure is triggered, and an emergency procedure is activated when leaving the Contingency Volume. Since testing flight terminations is costly in real-world test flights, we conducted simulation-based experiments, demonstrating that online monitoring provides an efficient method also during development.

Flight path monitoring ensures that a UAV follows its planned trajectory by detecting deviations that may require corrective actions. While geofencing is an operation-level requirement according to the hierarchy, flight path monitoring is a mission-level requirement that guarantees that the UAV follows the intended mission plan. To validate this function, we carried out a real-world test flight using a waypoint mission together with a specification that ensures continuous progress toward the final waypoint.

Using both properties, we demonstrate that tools for generating specifications can help to bridge the gap between CPS engineering and formal languages. During development, changes in the Operational Volume or UAV constraints often require updates to the specification. Moreover, engineers typically rely on graphical tools to derive the safety case used in SORA. Therefore, RTLola specifications need to integrate seamlessly into these established processes and should not introduce additional overhead. The derived monitors were tested in Software-in-the-Loop and Hardware-in-the-Loop setups before being validated in flight tests.

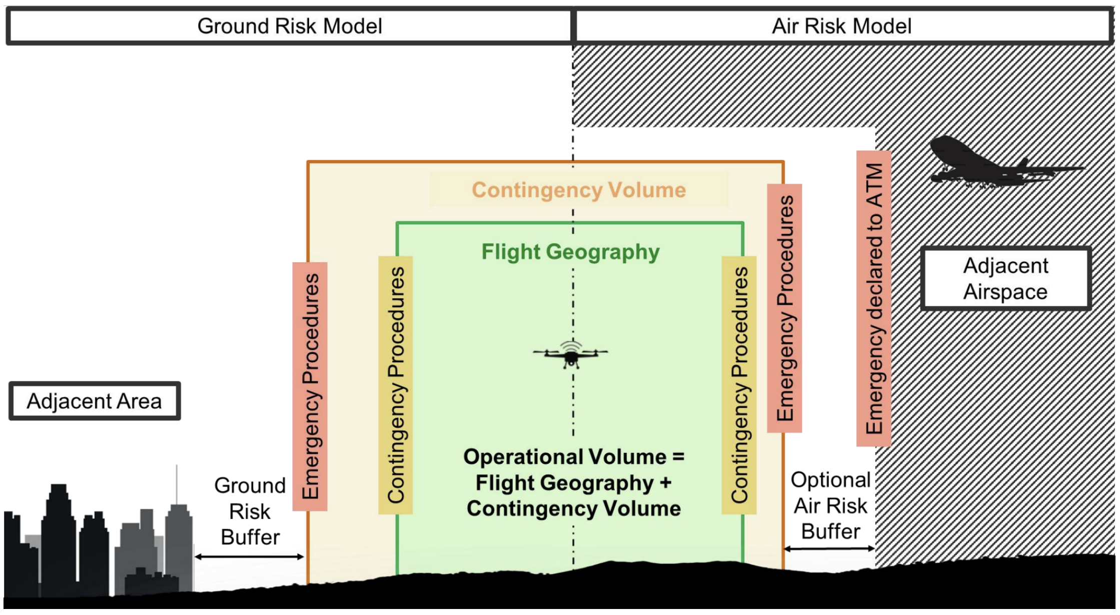


Figure 7.1.: Graphical Representation of SORA Semantic Model [148]

The remainder of this chapter is structured as follows. Section 7.1 presents the integration details, providing the runtime assurance architecture employed. In Section 7.2, we then present our tool *FencyCreator* which automatically generates geofencing and flight path monitoring specifications, streamlining the development process. Finally, Section 7.3 provides details on the generated specifications and presents experimental results that show the benefits of monitoring during simulation and in test flights. →Sec. 7.3, p. 104

Remark 7.1 (Justifying RTLola)

The decision to use RTLola for implementing geofencing and flight path monitoring was mainly motivated by three factors. First, we tested if RTLola could efficiently handle these requirements and how the specification would look like. Second, RTLola allows to write robust monitors as discussed in Remark 3.2. →Rem. 3.2, p. 42 Last, and most importantly, RTLola supports the transcompilation to VHDL code, which can be synthesized on an FPGA. While the analysis of an RTLola specification provides memory guarantees, execution on a hardware FPGA provides runtime guarantees determined by the depth of the circuit, avoiding the cumbersome computation of the worst-case execution for software code.

7.1. System Architecture

The DLR project ALAADy focused on automated air cargo delivery *beyond visual line of sight* for up to one ton, using various UAV configurations, such as fixed-wing aircraft with parachute or gyrocopter. It served as a feasibility study for an early version of SORA. ALAADy investigated if low-risk cargo delivery is achievable by automatically planning low-altitude trajectories that avoid populated areas and by monitoring flights with a RTA system (see Figure 2.2 for ASTM F3269-21). Both measures are intended → Fig. 2.2, p. 20 to significantly reduce the overall operational risk. The RTA for ALAADy is depicted in Figure 7.2[1]. Navigation, Mission Manager, and Flight Control together represent the Complex Function. The Safety Monitor supervises operational constraints, including geofencing, flight path monitoring, UAV stability, and connectivity to the remote operator and pilot. Both the Safety Monitor and the Input Manager, responsible for synchronizing inputs as defined in the full version of F3269-21, were implemented using RTLola. Two recovery functions are provided: one allows a safety pilot to take over control, and the other ensures safe flight termination so that the operation remains within the Operational Volume according to SORA. Note that this flight termination is considered safe, as the UAV is flying over unpopulated areas with a low risk of harm to people and infrastructure.

[1]This architecture is included as an appendix in ASTM F3269-21.

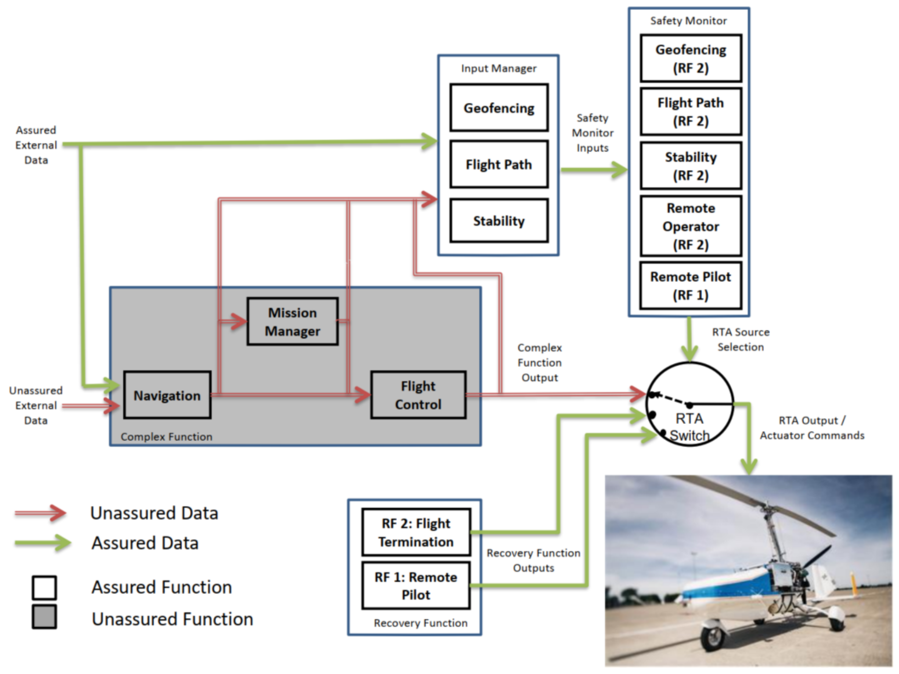


Figure 7.2.: ALAADy's RTA architecture for SORA compliance.

7.2. Tool: FencyCreator

SORA evaluates the risk of an operation using the concept of operations (ConOps). The ConOps defines how the UAV will be deployed. For instance, it defines the planned trajectory and the Operational Volumes. This information is provided as visual file to the agencies to do the assessment, e.g., Figure 7.3 depicts an Operational Volume in KML format. The volume and the trajectory is given as polygon chains.

The tool FencyCreator imports these KML files containing the Operational Volumes and the planned trajectory, allows to add information on UAV operational constraints such as maximal velocity and altitude, computes and adjusts buffer distances for contingency and emergency procedures, and finally generates a corresponding RTLola specification. Consider Figure 7.4, FencyCreator enables adjustment of the *risk buffer* and the *contingency buffer* to compute the Flight Geography and the Contingency Volume. In addition, a so-called *tube* can be created around the planned trajectory, with a configurable radius. Besides the generation of the RTLola specification, FencyCreator also checks the validity of the computed volumes and tube. It implements the Cross-Check Geofence Scaling Algorithm proposed in [93]. Cross-Checking highlights cases where downscaling an original geofence results in two or more disjoint geofences. For example in Figure 7.5, a contingency buffer of 150 meters and a risk buffer of 170 meters was used, resulting in valid volumes. In contrast, in Figure 7.6, a contingency buffer of 150 meters and a risk buffer of 250 meters was used, resulting in invalid volumes. Here, the volumes are split into multiple disjoint areas, as the geofence was too narrow at Latitude 52.40 and Longitude 10.55, preventing any trajectory from reaching the final waypoint from the initial point. Cross-checking helps to adjust the buffers.

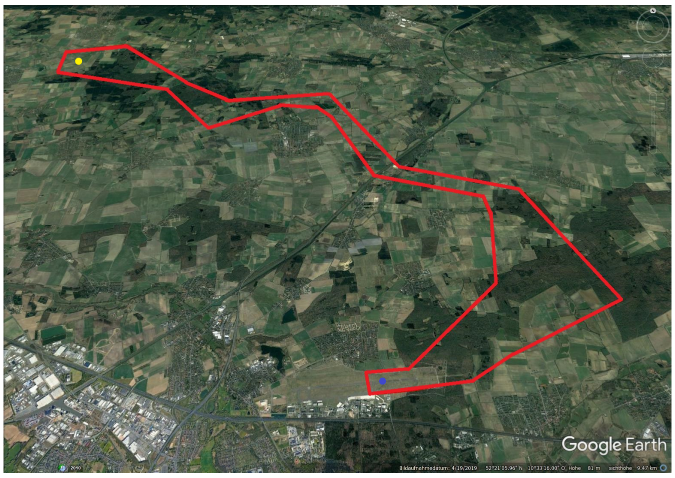


Figure 7.3.: Example Operational Volume, defined as polygon chain in a KML file, added to the ConOps. The blue dot and the yellow dot represent the initial position and the final waypoint, respectively. The complete trajectory is omitted.

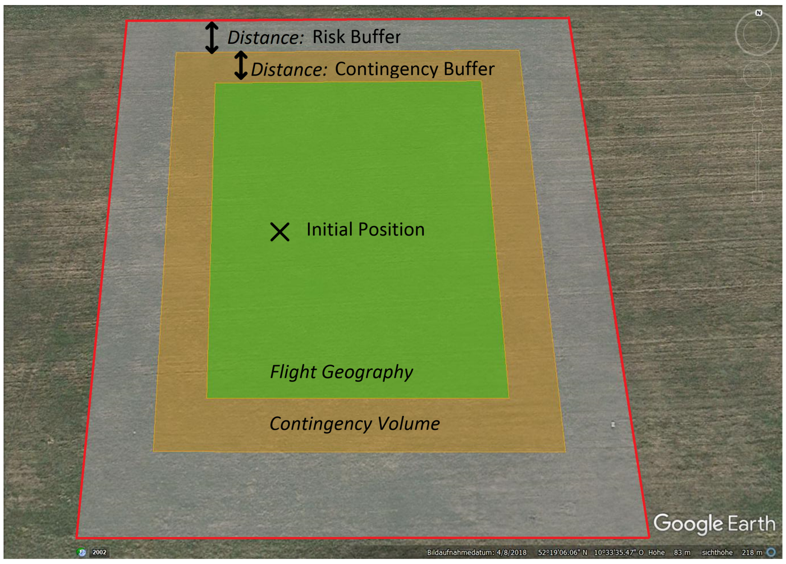


Figure 7.4.: Description of the SORA flight volumes, adjustable in FencyCreator.

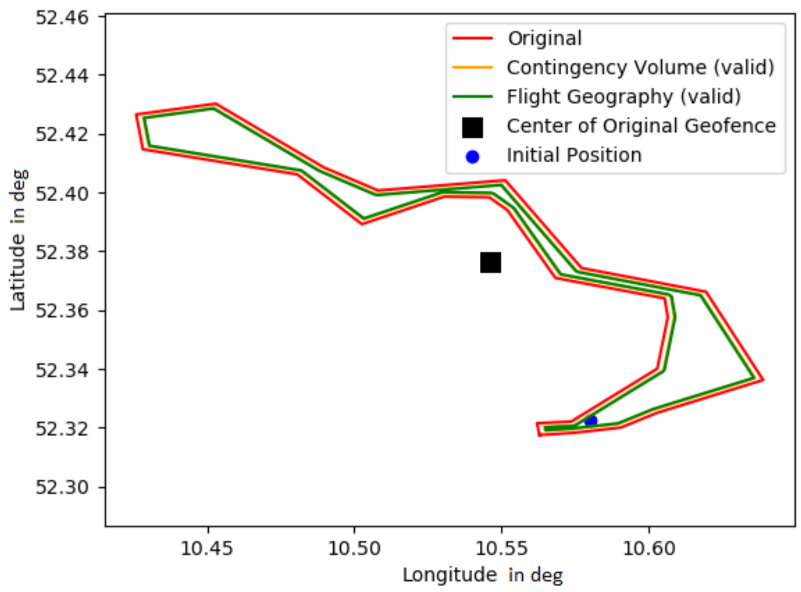


Figure 7.5.: Valid result of FencyCreator using a contingency buffer distance of 170 meters and a risk buffer distance of 150 meters.

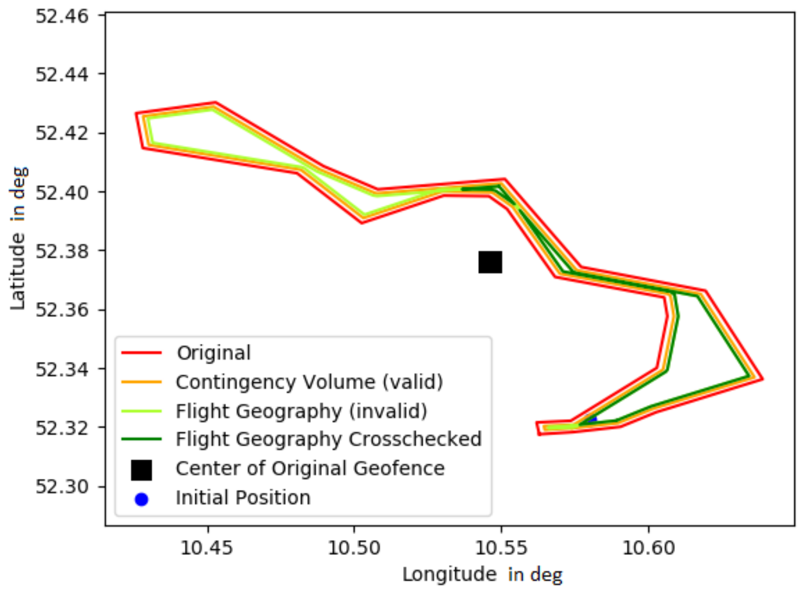


Figure 7.6.: Invalid result of FencyCreator using a contingency buffer distance of 250 meters and a risk buffer distance of 150 meters.

Remark 7.2 (Monitoring Other Levels of the Hierarchy)

The tools also supports adding constraints for monitoring sensor-, component-, and aircraft-level properties. For instance, the quality of the GPS-position can be monitored, e.g., see Remark 3.2. →Rem. 3.2, p. 42

7.3. Applications

In this section, we present the specifications for geofencing and flight path monitoring, along with experimental results. The geofencing algorithm was evaluated in simulation runs to validate safe flight termination. The flight path monitoring was deployed during test flights to monitor the progress toward the final waypoint. We assume a single Operational Volume represented as polygon chain $P = (p_1, p_2, \dots p_n)$ with $p_1 = p_n$, the planned trajectory $W = (w_1, w_2, \dots, w_n)$, and the currently flown trajectory $T = (t_1, t_2, \dots, t_n)$ where new sample points extend T by t_{n+1}. We also assume a NED-coordinate system, where each x- and y-coordinate can be accessed using the notation by $.x$ and $.y$. For instance, $t_1.x$ represents the x-coordinate of the trajectory point t_1.

7.3.1. Geofencing Monitoring

For the automatic detection of geofence violations, we check for line crossings between the polygonal chain P and the flown trajectory T. We assume a fixed altitude limit, hence, we can project the 3D geofencing to 2D with an additional altitude check as in Example 3.2.2.

→Ex. 3.2.2, p. 41

The algorithm consists of two steps. First, the trajectory line is determined. Second, the intersection between the trajectory line and each polygon line is computed.

(1) Trajectory Line Computation
With t_0 being the initial position, for the most recent trajectory line $t = [t_i, t_{i+1}]$ in T we compute the straight line $y = m_t \cdot x + b_t$ along with the *orientation* and distance values (*dst*), which are later used for the polygon line intersection check:

- $\Delta_{tx} := t_i.x - t_{i+1}.x$ ▷ x-difference
- $\Delta_{ty} := t_i.y - t_{i+1}.y$ ▷ y-difference
- $m_t := \Delta_{ty} / \Delta_{tx}$ ▷ slope of the trajectory
- $b_t := t_i.y - m_t \cdot t_i.x$ ▷ y-intersect of trajectory
- $dst := \sqrt{\Delta_{tx}^2 + \Delta_{ty}^2}$ ▷ flown distance
- $orientation_x := \Delta_{tx} < 0$ ▷ heading of x-coordinate
- $orientation_y := \Delta_{ty} < 0$ ▷ heading of y-coordinate

(2) Polygon Segment Intersection Computation
For each polygon segment $[p_k, p_l]$ in the polygonal chain, we compute an intersection point and check whether the point is valid, i.e., lies on both the current trajectory line and the polygon segment. Since the shape of the polygon remains fixed during operation, we precompute the slope m_p and y-intersect b_p of each polygon segment in advance, just as we have computed them for the trajectory.

The additional computations that check for a valid intersection are:

- $i_x := \frac{b_t - b_p}{m_p - m_t}$ ▷ Computes the x-coordinate of intersection point
- $i_y := m_p \cdot i_x + b_p$ ▷ Computes the y-coordinate of intersection point
- $\Delta_{ix} := i_x - t_i.x$ ▷ Intersection point x-difference
- $\Delta_{iy} := i_y - t_i.y$ ▷ Intersection point y-difference
- $orientation_{valid} := ((\Delta_{ix} < 0) = orientation_x) \wedge ((\Delta_{iy} < 0) = orientation_y)$
  ▷ Checks that the trajectory flew towards the intersection

- $dst_{valid} := \sqrt{\Delta_{ix}^2 + \Delta_{iy}^2} \leq dst$
  ▷ Checks if the travel distance suffices to reach intersection
- $bound_{valid} := \min(p_k.x, p_l.x) < i_x < \max(p_k.x, p_l.x)$
  ▷ Checks that intersection is within polygon line
- $violation_p := orientation_{valid} \wedge dst_{valid} \wedge bound_{valid}$
  ▷ Geofence exceeded if UAV was moving towards segment (*orientation*), trajectory reached intersection point (*dst*), and crossed the segment line (*bound*).

Remark 7.3 (Edge Cases)

When a trajectory or polygon segment is vertical ($\Delta_{tx} = 0$), the x-coordinate can be directly substituted into the other line's equation to compute the intersection. When one of the lines is horizontal ($m = 0$), the y-coordinate can be directly substituted into the other line's equation. If two lines have the same slope ($m_p = m_t$) but different intercepts, they are parallel and do not intersect. The same holds for two vertical lines but different x-coordinates.

One observation when formalizing the computations in RTLola is that these edge cases tend to drastically increase the size and complexity in terms of readability of the specification. This is due to adding "if-then-else"-statements within the specification.

Design Rational At its core, geofencing answers the question: "Is the UAV within or outside of an area?". Several algorithms exist [94] that range from grid-based algorithms to ray casting. We chose this approach for its simplicity and efficiency when translating it to RTLola. It relies on basic line intersection computations, a well-known geometry problem. The approach is highly parallelizable. Each polygon segment intersection (see Step 2) can be computed independently, making it well-suited for GPUs or FPGAs. Unlike approaches that rely on precomputed data structures, polygonal boundaries can be updated dynamically at runtime by computing the precomputation in Step 2 online. Also, arbitrary polygonal chains are supported, i.e., not only convex polygons. Further, basic geometry allows the computations to be expressed in RTLola, which allows FPGA synthesis. Last, in our applications, travel distances are sufficiently short enough that accounting for the Earth's curvature is unnecessary.

Remark 7.4 (Time to Intersection)

The algorithm allows to combine distance calculations with the UAV's velocity to estimate both how far and how long until a potential line violation occurs, flight tested in [48]. This represents an important information for drone operators, especially when operating under Unmanned Traffic Management control, i.e., U-Space in Europe.

7.3.2. Geofencing Experimental Results

The case study focuses on validating the risk buffer under different scenario parameters, i.e., the buffer whose violation triggers the safe flight termination. Figure 7.7 shows the Ground Risk Buffer (see Figure 7.1) in red, within which the UAV must terminate, while some of the flown simulation trajectories are displayed in white. The simulation allows variation of both wind conditions and wind direction. As depicted, a few trajectories exceed the Ground Risk Buffer due to strong wind. Upon termination, the gyrocopter cuts off power and sets the actuators to a predefined position, which is called the safe flight termination. Since its rotors are not engine-powered but rotate freely, it will descend slowly in autorotation, often compared to a descending maple seed. The reason for exceeding the Ground Risk buffer is that the wind counteracts this passive rotation of the gyrocopter's safe flight termination. The results indicate that either a larger buffer is needed or the vehicle should not operate under such strong wind conditions. Overall, monitoring simulations allows to draw conclusions before flight.

→Fig. 7.1, p. 100

Simulation Setup The experiments are conducted in a MATLAB Simulink environment. The simulation consists of several modules: Sensor Fusion, Mission Manager, Flight Control, Monitor, Flight Mechanics, and Environment. The Sensor Fusion module estimates the UAV's position and forwards it to the Mission Manager and Flight Control. The Mission Manager selects and executes the trajectory while the Flight Control module maps the manager's commands to actuator inputs. The RTLola-based Monitor checks for geofence violations and triggers a flight termination if necessary.

The ALAADy simulation supports three different flight mechanics: a gyrocopter, a box-wing aircraft, and a twin-boom aircraft. For the gyrocopter, the safe flight termination is the predefined autorotation described earlier. For fixed-wing aircraft, parachutes are used for termination. A risk buffer of 1200 meters is applied for the gyrocopter, and 400 meters for the fixed-wing aircraft. The simulation also supports CS-AWO (All Weather Operations) compliant environmental modules for wind turbulences.

Scenario Parameters For validating geofencing, we provoked geofence violations under different aircraft states and wind conditions. We differentiate between four wind conditions based on the Beaufort scale: none (0 m/s), light breeze (2.6 m/s), moderate breeze (6.3 m/s), and strong breeze (12.4 m/s). Winds from the south and south-west were chosen as the most relevant for the scenario, as they provide a tailwind. The UAV was also flown at different altitudes: 60 m, 100 m, and 140 m.

The gyrocopter simulation scenarios are shown from a top view in Figure 7.8. The top views for the fixed-wing aircraft are essentially identical, as the same trajectory is used. Only the flight mechanics and parachutes upon termination differ. All scenarios start at the bottom-left corner. The objective of the scenarios is to trigger flight termination at the same predefined point of the geofence under different attitudes of the aircraft.

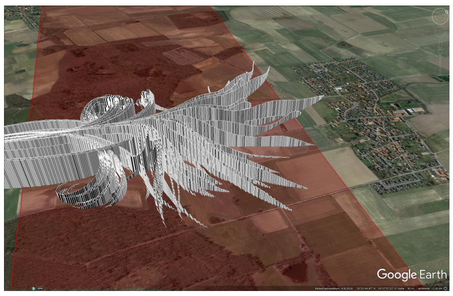


Figure 7.7.: Gyrocopter flight termination parameter study.

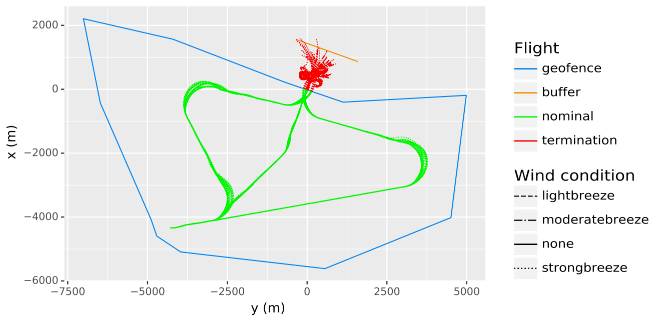


Figure 7.8.: Top view of the termination scenario.

Therefore, three families of trajectories are chosen. One triggers the termination when the UAV is on a left turn, one when the UAV is on a right turn, and the last one triggers the termination when the UAV is flying in a straight line towards the fence. The effects of the different wind condition can be seen in the different simulation runs. In simulation runs under heavier wind conditions, the UAV is dragged further away from the planned trajectory. For the following, note that the figures of the results focus on the relevant part of the buffer.

Gyrocopter Results Figure 7.9a shows the gyrocopter simulation results, starting ten seconds before the geofence violation and ending at ground impact. The plot highlights the impact of the gyrocopter's attitude and wind conditions. When termination begins during a left or right turn, the maneuver distance is shorter, as the gyrocopter can reach the desired spiral behavior more easily. Strong winds make this spiral behaviors harder to achieve, especially when flying straight toward the geofence with tailwinds. Buffer violations occur, indicated by crossing the orange line, likely due to underestimating wind effects on termination distance. Two mitigations are possible. First, the buffer can be increased and a new geofencing specification generated using FencyCreator, leading to an earlier flight termination. Second, the ConOps used for SORA can be updated to prohibit operations under strong-breeze conditions or at reduced allowed velocities.

Box-Wing Aircraft Results The simulation results for the box-wing aircraft are given in Figure 7.9b. As before, the plot covers ten seconds before geofence violation until ground contact. The buffer distance is approximately 400 m. Due to the parachute, vehicle attitude and wind conditions have little effect on the termination behavior. In all cases, the buffer line remained sufficiently distant, and no violations occurred.

Twin-Boom Aircraft Results Figure 7.9c illustrates the twin-boom aircraft results. The constraints and outcomes are similar to those of the box-wing aircraft. The same parachute model is used, and differences in flight mechanics are too minor to affect termination behavior.

Conclusions of the Simulation Parameter Study Wind condition and the vehicle attitude play an important role for the gyrocopter termination, unlike fixed-wing aircraft with parachute-based termination. To account for heavy wind conditions, either the buffer can be increased or the ConOps updated. The first option is preferable, as a single safe trajectory would suffice. If no safe trajectory exists due to limited airspace, the second option may be necessary.

Note that the experiment does not address impact energy upon ground contact. For future work, it would be valuable to compare gyrocopter gliding with parachute descent, as well as system complexity between gyrocopter and parachute termination chains, as these factors affect certification and vehicle cost. The simulation used is not tool-qualified, meaning results must be manually verified. Each module must accurately model the actual system, particularly the flight mechanics, which is challenging if buffer validation relies solely on simulations. Nonetheless, simulations provide a systematic way to fine-tune operational parameters. In this study, initial buffer distances were derived and found to underestimate environmental effects. Based on these observations, runtime monitors were refined using FencyCreator to prevent buffer breaches under such conditions. In future, monitors should be extended to provide statistics upon triggering a safe flight termination, such as the time until ground contact or the distance traveled after activation.

Remark 7.5 (Geofencing in Flight Tests)

An actual flight test for geofencing, using the transcompilation from RTLola to VHDL and demonstrating realizability on an FPGA [2], was conducted in [21]. The implementation provided guarantees on the execution time of the monitor implementation and constant memory consumption as ensured by analyzing the RTLola specification.

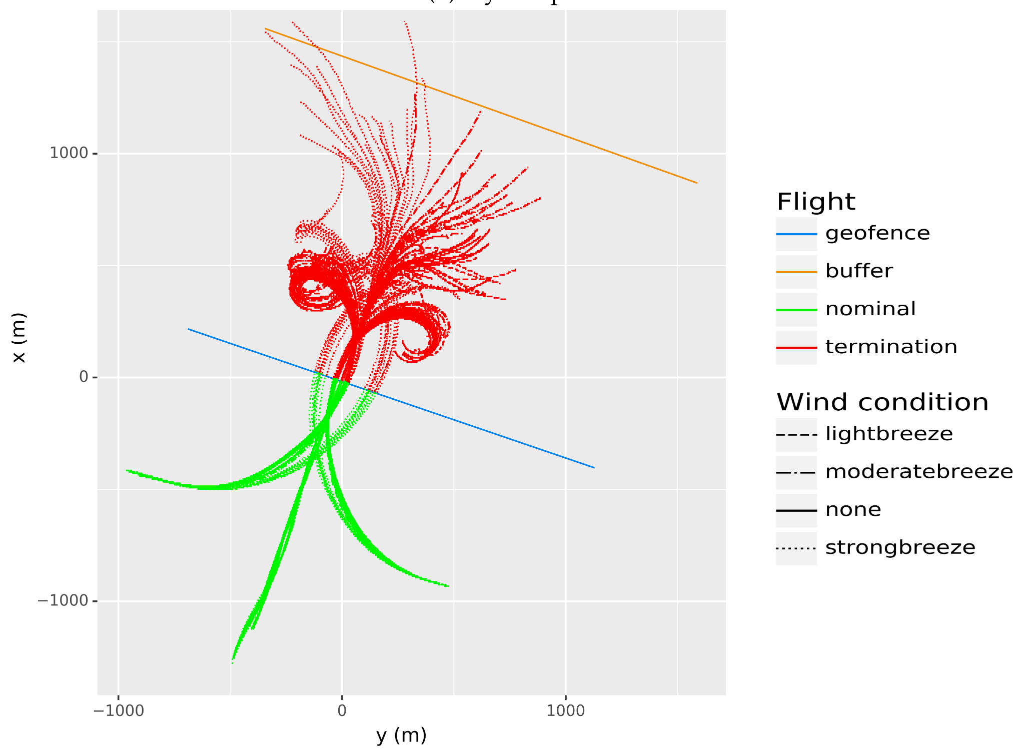
7.3.3. Flight Path Monitoring

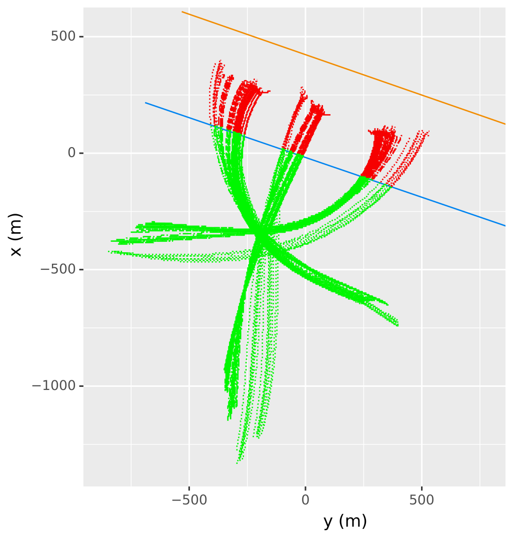
Similar to checking each polygon segment of the geofence, we also check each segment of the planned trajectory W. Instead of computing intersection points between lines, we determine the distance between the current UAV position and the trajectory segments. If the minimal distance is smaller than the defined tube radius, the UAV is within the tube. Otherwise, it is outside.

The algorithm computes the distance to a single planned trajectory segment $[w_i, w_{i+1}] \in W$ in two steps, and this process is repeated for all segments of W. First, it computes

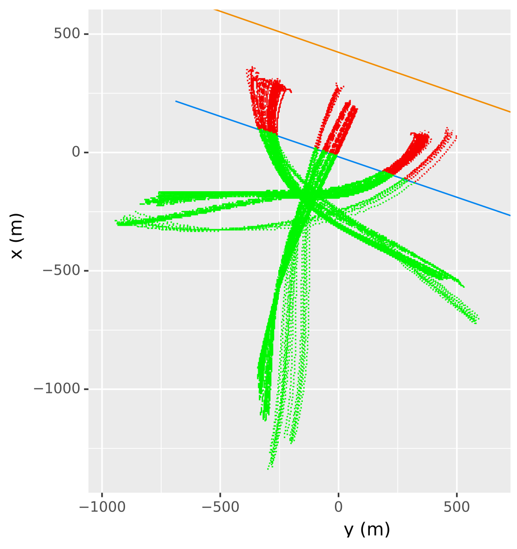
[2] ZC702 with around 85.000 LUTs on board of the Xilinx Zynq-7000 SoC Evaluation Board

(a) Gyrocopter





(b) Box-wing aircraft



(c) Twin-boom aircraft

Figure 7.9.: Top view of the various aircraft configurations at safe flight termination.

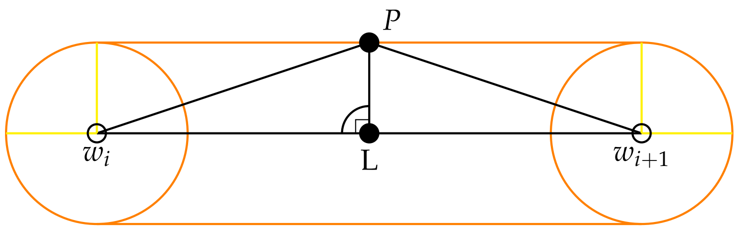


Figure 7.10.: 2D visualization of the distance computation.

distances from the current position P to the segment endpoints as well as to the orthogonal projection L on the segment line. Second, it evaluates if P lies within the tube based on its radius and these distances.

(1) Distance Computation

Figure 7.10 illustrates the computation in 2D. The same algorithm works for 3D. First, we compute L of the current UAV position P onto the trajectory segment line $[w_i, w_{i+1}]$. Next, we compute the distances $|w_iP|$, $|w_{i+1}P|$, $|w_iw_{i+1}|$, $|w_iL|$, $|w_{i+1}L|$, and $|LP|$.

(2) Check Tube

The distance $|LP|$ is valid only if L lies on the segment, i.e., if $|w_iL| < |w_iw_{i+1}|$ and $|w_{i+1}L| < |w_iw_{i+1}|$. If it is valid, the minimal distance to the segment is $|LP|$. Otherwise, it is $\min(|w_iP|, |w_{i+1}P|)$.

The overall minimal distance to the tube is then obtained by taking the minimum over all the trajectory segments. If the minimal distance exceeds the tube radius, the UAV is considered outside the tube. Otherwise, it is considered inside.

Progress Monitoring

In addition to geometric containment, mission progress is monitored using watchdog timers that track if the UAV approaches each waypoint within a specified time limit. The following RTLola specification implements a simplified version of such a watchdog:

```
input position: (Float32,Float32,Float32) // in m                                    1
const waypoint: (Float32,Float32,Float32) := (1.0,1.0,1.0)                           2
const watchdog_limit : Float32 := 30.0 // seconds                                    3
const epsilon : Float32 := 5.0 // meters                                             4
output watchdog @1Hz := if (|position-waypoint|).aggregate(over: 1s, using: min)     5
      .defaults(to: epsilon) < epsilon) then 0.0 else watchdog[-1,0.0] + 1.0         6
trigger @1Hz watchdog > watchdog_limit "Bark! Timer expired"                         7
```

Given the current position as input (Line 1) and the target waypoint (Line 2), the output `watchdog` (Lines 5-6) stores the time passed. If the waypoint is reached within ϵ distance, the timer resets to 0.0. Otherwise, the timer is incremented by the watchdog frequency, i.e., by one second. If the timer exceeds the predefined limit of 30 seconds, a trigger is raised to notify the operator about the progress stagnation. Note that the `aggregation` in Line 5 is required by RTLola's type system. Further, multiple waypoints and their required sequential order can be encoded directly in RTLola.

Design Rational Flight path monitoring addresses the question: "Is the UAV close enough to its planned trajectory and does it make progress toward the final waypoint?". Similar to geofencing, it relies on basic geometric computations. However, instead of intersections, it computes distances. A key advantage of the algorithm is that the computations can be parallelized on hardware, as the distance to each trajectory segment can be calculated independently (as for geofencing). A potential limitation is the assumption of linear segments between waypoints, which may not accurately represent curved trajectories.

7.3.4. Flight Path Monitoring Experimental Results

We report on the use of Flight Path Monitoring.

Setup We deployed the Flight Path Monitor onboard a UAV [47], with position data provided by a GPS sensor. The UAV executed a straightforward pre-planned waypoint mission consisting of ten waypoints, numbered from 0 to 9. The waypoints were required to be reached in sequence, i.e., first 0, then 1, and so on. After reaching the last waypoint, the mission was terminated and the UAV returned to its home position close to Waypoint 1. For tube containment, a radius of 10 meters was used. The epsilon distance for reaching a waypoint was set to 5 meters and a time limit of 30 seconds was applied.

Tube Distance Results Figure 7.11 shows the results of monitoring the tube distance during the flight test. The pre-planned mission is depicted in blue, and the actual flight path is illustrated as a color-varying line where green indicates that the UAV remained inside the tube, orange indicates the UAV being close to the tube boundary, and red indicates that the UAV was outside the tube. The results demonstrate that the UAV successfully followed the planned mission and remained within the tube throughout the flight, except after reaching the final waypoint when it returned to the home position.

Watchdog Results Figure 7.12 shows the waypoint watchdog results. Brighter colors represent larger timer values. The timer was successfully reset at most waypoints, confirming that they were reached within the specified ϵ distance. However, at Waypoint 3, the timer was reset but it took too long to reach Waypoint 4, and at Waypoint 9 the timer was never reset because the waypoint was not reached within the epsilon distance. All other waypoints were sufficiently reached, ensuring proper mission progress.

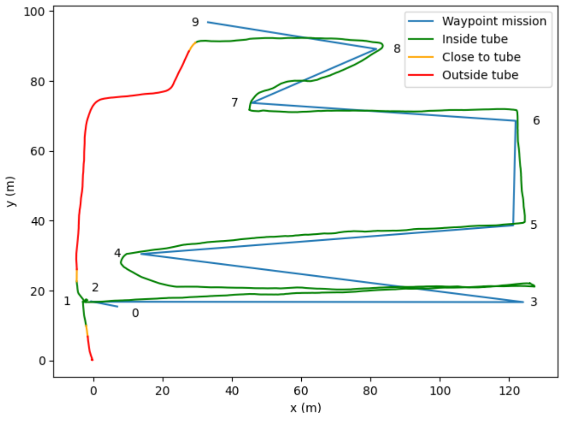


Figure 7.11.: Top-view of the tubing results. The UAV started outside the tube, entered and remained within the tube (green) until the waypoint mission stopped at Waypoint 9. Then, the UAV left the tube to fly to the home position close to Waypoint 1.

Conclusions Overall, the flight test confirmed that Flight Path Monitoring works as intended: it reliably reported both the UAV's progress toward the mission objective and its adherence to the pre-planned trajectory.

7.4. Summary

This chapter introduced Safe Operation Monitoring (SOM) in the context of SORA and demonstrated its use for two key properties: geofencing as an operation-level requirement and flight path monitoring as a mission-level requirement. We presented the FencyCreator tool, which automatically generates RTLola specifications from operational data such as planned trajectories and Operational Volumes. It thereby reduces engineering effort and ensures consistency with safety cases.

Through simulation studies, we used RTLola to analyze the impact of environmental factors such as wind and different vehicles configurations on geofencing buffers. The simulation results showed that the gyrocopter's safe termination maneuver can violate operational limits under strong winds, while parachute-based termination provides a more reliable solution, albeit at the cost of increased system complexity. Additionally, in real-world flight tests we confirmed that flight path monitoring effectively detects deviations from planned missions and supports progress supervision, which is an essential capability for operations beyond visual line of sight.

Together, these results demonstrate that SOM, supported by formal specifications and automation tools such as FencyCreator, provides a practical and scalable approach for supervising UAV operations and ensuring regulatory compliance.

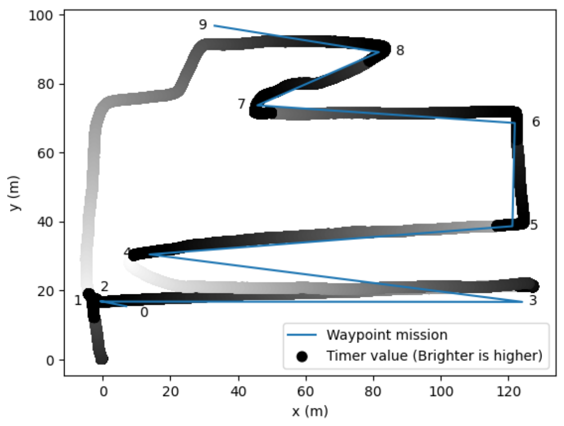


Figure 7.12.: Top-view of the tubing watchdog. At each waypoint a timer is started that is reset when the next waypoint is reached within an ϵ distance. The brighter colors of the flight show that the UAV did not reach Waypoint 4 in time and Waypoint 9 was outside the ϵ distance to reset the timer. All other waypoints reset the timer in time.

Chapter 8

Monitoring with Verified Guarantees

In this chapter, we address the concern that writing monitoring specification, while generally less prone to errors than handwritten code, can still lead to errors that are often more subtle and complex. Consequently, effective methods for detecting such specification error are essential. To address this, we introduce a verification feature to the Lola framework. Specifically, we extend the specification language with *assumptions* and *assertions*. The framework statically verifies that the assertions are guaranteed to hold if the input to the monitor satisfies the assumptions. Additionally, we provide a proof of soundness and present an online monitoring approach with experimental results that checks the satisfaction of assumptions during runtime to activate assertion checks only if necessary.

In practice, system engineers report that support for specification development is necessary, e.g., sanity checks and proofs of correctness. Additionally, recent developments in unmanned aviation regulations and standards indicate a similar necessity. One such development is the previously mentioned industry standard F3269-21 (Standard Practice for Methods to Safely Bound Flight Behavior of Unmanned Aircraft Systems Containing Complex Functions). At the core of the architecture is a safety monitor that takes the inputs and outputs of the complex function, and decides whether the complex function behaves as expected. If not, the monitor switches the control from the complex function to a matching recovery function. For instance, the flight of an unmanned aircraft could be separated into different phases: e.g., take-off, cruise flight, and landing. For each of these phases, a dedicated recovery could be defined, e.g., braking during take-off, the activation of a parachute during cruise flight, or a go-around maneuver during landing. Further, it is crucial that recoveries are only activated under certain conditions and that only one recovery is activated at a time. For instance, a parachute activation during a landing approach is considered safety-critical. The verification extension of Lola introduced in this thesis can be used to guarantee statically that such decisions are avoided within the monitor specification. Consider the simplified Lola specification

```
input event_a, event_b, value: Bool, Bool, Float32
assume <a1> !(event_a and event_b)
output braking  : Bool := ...computation...
output parachute : Bool := ...computation...
output go_around : Bool := ...computation...
assert <a1> !(braking and parachute)
```

that declares an assumption on the system input `events` and asserts that `braking` and `parachute` never evaluate to *true* simultaneously.

In the following, we present the verification approach, give details on the tool implementation, and report on our tool experience with specifications that were written based on interviews with aviation experts. Our results demonstrate that standard Lola specifications remain prone to error, but that these errors can be effectively detected with the formal verification enabled by our extension. Last, we examine scenarios in which assumptions may not hold during runtime. We observe that the extensions can also reduce runtime overhead, in particular by checking *complex assertions* only when the corresponding *simple assumptions* are violated.

8.1. Assumptions and Assertions

In this section, we present the verification extension for the Lola specification language. The extension allows the developer to annotate the Lola specification with *assumptions* and *assertions* in order to verify the desired guarantees on the computed streams. As an example, consider the simplified specification in Listing 8.1, which is structured into stream computations in Lines 1 to 23, and assumptions and assertions from Line 26 onwards.

The computation part specifies a safety monitor within a RTA architecture that triggers recovery functions for three different flight phases. First, the take-off recovery function is triggered (Line 21) when the targeted take-off speed was not achieved on a runway up to a predefined point (Line 13). The distance between the current position and the end of the runway with local coordinates $(0, 0)$ is computed in Line 8. Second, in-flight a parachute is activated (Line 22) when virtual barriers for the aircraft, i.e., a geofence, are exceeded (Line 15) [95]. Last, during landing, up to a point of no return (`alt < 10.0`), a new landing attempt is initiated (Line 23) if the aircraft's speed is too fast or its landing gear is not yet ready. To be more robust, the current and the previous value of the `landing_gear_ready` is taken into account (Lines 17-18).

With the verification extension, the specification assures that recoveries are not activated simultaneously (Lines 30-31), i.e., for instance there is no possibility that a parachute is activated during a landing approach. The first two conjunctions in Line 30 evaluate to *false* because relevant outputs use a disjoint altitude condition. The last conjunction requires an assumption. Here, two assumptions are linked by the identifier `a1` to the assertion. The assumptions specify: the known bound of received speed data

```
input alt : Float32 // Height above ground
input x, y : Float32, Float32 // Position in local coordinate system
input speed : Float32 // Velocity of aircraft
input landing : Bool // Indicates landing mode
input lg_status : (Float32,Float32,FLoat32) // Status of landing gear

// Complex computations
output dst_on_runway      : Float32 := √(x² + y²)
output geofence_violation : Bool    := ...
output landing_gear_ready : Bool    := ...

// Take-off contingency
output decelerate := alt < 1.0 ∧ speed < 10.0 ∧ dst_on_runway > 20.0
// In-flight contingency
output parachute := geofence_violation ∧ alt > 100.0
// Landing contingency
output gain_alt := landing ∧ alt ≥ 10.0
                   ∧ (landing_gear_ready[-4..0, true, ∧] → speed > 10.0)

// Notifications to the system
trigger decelerate "RECOVERY: Stop take-off by decelerating aircraft."
trigger parachute  "RECOVERY: Activate parachute."
trigger gain_alt   "RECOVERY: Gain altitude for next landing attempt."

// By concept of operations: landing is always within geofence.
assume <a1> ¬(landing ∧ geofence_violation)
assume <a1> abs(speed) ≤ 80.0 // Given by data protocol

// Only one contingency is activated at once.
assert <a1> ¬( (decelerate ∧ parachute) ∨ (decelerate ∧ gain_alt)
               ∨ (parachute ∧ gain_alt) )
// Parachute SHALL ONLY be activated 100 meters above ground.
assert <a2> parachute → alt > 100.0
```

Listing 8.1: A simplified Run-Time Assurance RTLola specification with three recovery functions for three different flight phases. Assumptions and assertions are used to show that only one recovery function is activated at once.

(Line 27) as well as operational information (Line 26), e.g., given by the concept of operation a nominal landing is only foreseen within the predefined operational airspace. Note that assumptions are provided by the user and are *assumed* to be valid. Further, a second assertion is stated in Line 33 that guarantees that *the parachute should only be activated when the aircraft is 100 meters above ground*. In this case, the property can be shown assumption-free. Assertions help engineers to show that certain properties are *true*. The given assertions indicate how specification debugging and management can benefit from the extension – it avoids digging into complex computations.

The extension and its verification approach are presented in the following. In general, the verification extension is used if a Lola specification (see Definition 3.4) is annotated in the following way:

→ Def. 3.4, p. 38

$$
\left.\begin{array}{rl}
\textbf{assume}\ \langle \alpha_1 \rangle & \theta_1 \\
\vdots & \\
\textbf{assume}\ \langle \alpha_m \rangle & \theta_m
\end{array}\right\}\text{Assumptions}
$$

$$
\left.\begin{array}{rl}
\textbf{assert}\ \langle \alpha_{m+1} \rangle & \psi_1 \\
\vdots & \\
\textbf{assert}\ \langle \alpha_{m+n} \rangle & \psi_n
\end{array}\right\}\text{Assertions}
$$

where $\alpha_1, \ldots, \alpha_{m+n} \in \Gamma$ are identifiers for $\theta_1, \ldots, \theta_m, \psi_1, \ldots, \psi_n$, which are Boolean stream expressions with possibly temporal operators. For convenience, we define functions which return all θ and ψ that are linked to a given α identifier:
$\mathit{assume}(\alpha) = \{\theta_j \mid \forall \alpha_j \in \Gamma, \alpha = \alpha_j\}$ and $\mathit{assert}(\alpha) = \{\psi_j \mid \forall \alpha_j \in \Gamma, \alpha = \alpha_j\}$. The set of assertion $\psi_1, \ldots, \psi_n$ is *correct* for all input streams if and only if whenever an assumption is satisfied, its corresponding assertion is satisfied as well.

The verification of assertions relies on the encoding of the Lola execution in Satisfiability Modulo Theory (SMT). We define the *smt* function that encodes a stream expression in Definition 8.1. It can be used to encode independent and dependent variables as well as expressions of assumptions and assertions.

Definition 8.1 (SMT-Encoding of Stream Expressions)

Let Φ be a Lola specification over independent stream variables $t_1, \ldots, t_m$ and dependent stream variables $s_1, \ldots, s_n$. Further, let the natural number $N+1$ be the length of the input streams, c be an SMT constant symbol, and $\tau_1^0, \ldots, \tau_1^N, \ldots, \tau_m^0, \ldots, \tau_m^N, \zeta^0, \ldots, \zeta_1^N, \ldots, \zeta_n^0, \ldots, \zeta_n^N$ be SMT variables. Then, the function *smt* recursively encodes a stream expression e at position j with $0 \leq j \leq N$ in the following way:

Base cases:	Inductive cases:	
$smt(c)(j) = \texttt{c}$	$smt(f(e_1,\dots,e_n))(j) =$	$\texttt{f}(smt(e_1)(j), \dots, smt(e_n)(j))$
$smt(t_i)(j) = \tau_i^j$	$smt(ite(e_b,e_1,e_2))(j) =$	$\texttt{ite}(smt(e_b)(j), smt(e_1)(j), smt(e_2)(j))$
$smt(s_i)(j) = \zeta_i^j$	$smt(e[k,c])(j) =$	$\begin{cases} smt(e)(j+k) & \text{if } 0 \leq j+k \leq N, \\ \texttt{c} & \text{otherwise} \end{cases}$

where `ite` is an SMT encoding of *if-then-else*; `f` is an interpreted function if f is from a theory supported by the SMT solver and an uninterpreted function otherwise.

Next, Proposition 13 proves the correctness of asserted stream properties for finite input streams. If the set of assertions is correct, asserted stream properties are guaranteed to be valid in each step of the monitor execution. In practice, such specifications are preferable. In the following, let Φ be a Lola specification with verification annotations. Further, we refer to the set of input streams and computed output streams as stream execution.

Proposition 13 (Assertion Verification of a Finite Stream Execution)**.**
Let Φ be a Lola specification and let $s_1,\dots,s_n$ be dependent stream variables used in Φ. The set of assertions is correct for a finite stream execution with length $N+1$ under given assumptions, if the following formula is valid:

$$\bigwedge_{0\leq i\leq N} \Big(\bigwedge_{\alpha\in\Gamma} \Big(\bigwedge_{\theta\,\in\, assume(\alpha)} smt(\theta)(i) \wedge \bigwedge_{s_k\in\Phi} \zeta_k^i = smt(e_k)(i) \rightarrow \bigwedge_{\psi\,\in\, assert(\alpha)} smt(\psi)(i) \Big)\Big)$$

The formula in Proposition 13 unfolds the complete stream execution and expresses that an assertion must hold in each stream position whenever its corresponding assumption and implementation are satisfied.

To avoid the complete unfolding and allow arbitrary stream lengths, an inductive argument is given in Proposition 14 that defines proof obligations for an annotated Lola specification. Next, we present a template for the stream unfolding that helps to define the proof obligation at the *Begin*ning (Definition 8.3), during *Run* (Definition 8.4), and at the *End* (Definition 8.5) of a stream execution. →Prop. 14, p. 123

Definition 8.2 (Template Stream Unfolding)
We define the template formula ϕ_t that states proof obligations as:

$$\bigwedge_{\alpha\in\Gamma}\Bigg(\bigwedge_{i\in p\_asm}\Big(\bigwedge_{\theta\in assume(\alpha)} smt(\theta)(i)\Big) \;\wedge \bigwedge_{i\in p\_asserted}\Big(\bigwedge_{\psi\in assert(\alpha)} smt(\psi)(i)\Big)$$

$$\wedge \bigwedge_{i\in p\_streams}\Big(\bigwedge_{0<k\leq n} \zeta_k = smt(e_k)(i)\Big) \;\rightarrow \bigwedge_{i\in p\_assert}\Big(\bigwedge_{\psi\in assert(\alpha)} smt(\psi)(i)\Big)\Bigg)$$

where *p_asm*, *p_asserted*, *p_streams*, and *p_assert* are template parameters. They are sets of positions for the unfolding of assumptions, previously proven assertions, output streams, and assertions, respectively.

The template formula in Definition 8.2 uses template parameters for the stream unfolding. For instance, the parameter assignment $p\_asm := \{i \mid 0 \leq i < 10\}$ adds assumptions at the first ten positions of the stream execution. Further, the parameter *p_asserted* allows to incorporate the induction hypothesis.

In the following, we will use the Lola specification in Listing 8.2 as a running example for the template stream unfolding. Here, the input `reset` represents a reset command for the output stream `o1` that counts how long no `reset` occurred. Output `o1` is used by output `o2` which aggregates over the previous, the current, and the next outcome of `o1`. This is achieved by using the offset operator, e.g., `o1[-1, 0]` accesses the previous value of `o1` if it exists, otherwise it takes the default value `0`. As assertion, we show that `o2` is always positive and never larger than three given the assumption that in each execution step either the previous or the next `reset` is *true*. The assumption ensures that at most two consecutive `resets` are *false*. Given the `reset` sequence of input values $\langle true; false; false\rangle$ that satisfies the assumption, the resulting `o1` stream evaluates to $\langle 0; 1; 2\rangle$. Here, at the second position of the sequence, `o2` evaluates to three. To show that the assertion also holds at the first and the last position of the sequence, out-of-bounds values must be considered.

We show how the template ϕ_t can be used at the beginning of a stream execution. Here, default values due to past stream accesses beyond the beginning of a stream need

```
assume<a1> reset[-1, false] ∨ reset[1, false]
input reset : Bool
output o1 := if reset then 0 else o1[-1, 0] + 1
output o2 := o1[-1, 0] + o1 + o1[1, 0]
assert<a1> 0 ≤ o2 and o2 ≤ 3
```

Listing 8.2: Lola specification with assumptions on a reset that guarantees that an output remains within bounds.

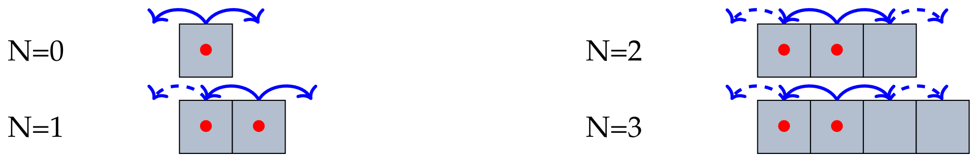


Figure 8.1.: Four stream executions of different length $N+1$ with the respective template unfolding are depicted. The stream executions consider all cases with past out-of-bound accesses. A gray-colored box indicates that an assumption has been unfolded at this position, while a red dotted box indicates that an assertion has been unfolded at this position. Solid and dashed arrows indicate accesses by streams and annotations, respectively.

to be captured by the obligation to guarantee that the assertions hold in these cases. The combination of past out-of-bounds and future out-of-bounds default values must also be covered by the obligations in case the stream is stopped early. These scenarios are depicted for the running example in Figure 8.1.

The figure shows four finite stream executions with different lengths. All stream positions are colored gray, while only some positions contain a single red dot. These features indicate the unfolding of stream variables and annotations using the template ϕ_t. A gray-colored position means that the assumptions have been unfolded and a dotted position means the assertion has been unfolded. Further, arrows indicate temporal stream accesses where solid lines correspond to accesses by outputs and dashed lines correspond to accesses by annotations, i.e., assumptions and assertions. For each stream execution, only the arrows for a single position are depicted – the arrows for other positions have been omitted for the sake of clarity. For example, for $N = 0$, the accesses of output `o2` are both out-of-bounds, i.e., the default value zero is used. While for $N = 3$, the accesses at the second position are shown where only the past access of the assumption leads to an out-of-bounds access, i.e., only the dotted line towards the beginning of the stream execution. The figure depicts all necessary stream executions that cover all combinations of past out-of-bounds accesses, i.e., with and without future bound violations. The described unfoldings of Figure 8.1 are formalized as proof obligations in Definition 8.3.

Definition 8.3 (Proof Obligations ϕ_{Begin} for Past Out-of-bounds Accesses)

Let $w_p = \sup(\{0\} \cup \{|k| \mid e[k,c] \in \Phi \text{ where } k < 0\})$ be the most negative offset and $w_f = \sup(\{0\} \cup \{k \mid e[k,c] \in \Phi \text{ where } k > 0\})$ be the greatest positive offset. The proof obligations ϕ_{Begin} for past out-of-bounds accesses are defined as the conjunction of template formulas:

$$\bigwedge_{N:\, 0 \leq N < \max(1,\, 2\cdot(w_p + w_f))} \phi_t(p\_asm,\ p\_asserted,\ p\_streams,\ p\_assert)$$

with template parameters:

- p_asm $:= \{i \mid 0 \leq i \leq N\}$,
- $p\_asserted$ $:= \emptyset$,
- $p\_streams$ $:= \{i \mid 0 \leq i \leq N\}$,
- p_assert $:= \{i \mid 0 \leq i < \max(1, \min(N+1, 2 \cdot w_p))\}$.

Next, the case where no out-of-bounds access occurs is considered. Hence, the obligations capture the nominal case where no default value is used. Since we have shown that past out-of-bounds accesses are valid we can use these proven assertions as assumptions. Figure 8.2 depicts a stream execution with a single dotted position, i.e., the position where the assertion must be proven. As can be seen, all accesses from this position are within bounds. Further, note that the accesses of the first and the last unfolded assumption, i.e., the first and the last gray-colored position, are also within bounds. The described unfolding is formalized as proof obligations in Definition 8.4.

Definition 8.4 (Proof Obligations ϕ_{Run} for No Out-of-bounds Accesses)

The proof obligations ϕ_{Run} without out-of-bounds accesses are defined as $\phi_t(p\_asm, p\_asserted, p\_streams, p\_assert)$ with template parameters:

- p_asm $:= \{i \mid w_p \leq i \leq N - w_f\}$,
- $p\_asserted$ $:= \{i \mid 2 \cdot w_p \leq i \leq N - 2 \cdot w_f \wedge i \neq 3 \cdot w_p\}$,
- $p\_streams$ $:= \{i \mid 2 \cdot w_p \leq i \leq N - 2 \cdot w_f$,
- p_assert $:= \{i \mid i = 3 \cdot w_p\}$,

where $N = 3 \cdot (w_p + w_f)$.

Last, we consider the case where only future out-of-bounds accesses occur. Hence, the respective obligations need to incorporate default values of future out-of-bounds accesses. As before, we can use the previously proven assertions as assumptions. Figure 8.3 depicts a stream execution with two dotted positions, i.e., positions where the assertion must be proven. The position where arrows are given represents the case

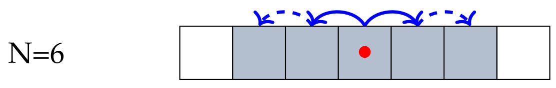


Figure 8.2.: A stream execution of length $N + 1$ with the corresponding template unfolding is depicted. The stream execution considers the case where no out-of-bound access occurs. Gray-colored and red dotted positions represent unfolded assumptions and assertions, respectively. Solid and dashed arrows indicate accesses by streams and annotations, respectively.

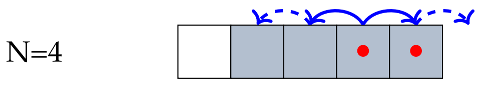


Figure 8.3.: A stream execution of length $N+1$ with the corresponding template unfolding is depicted. The stream execution covers all cases where future out-of-accesses occur. Gray-colored and red dotted positions represent unfolded assumptions and assertions, respectively. Solid and dashed arrows indicate accesses by streams and annotations, respectively.

where only the assumption results in a future out-of-bounds access. The last position of the stream execution represents the case in which both the assumption and the stream result in future out-of-bounds accesses. The presented unfolding is formalized as proof obligations in Definition 8.5.

Definition 8.5 (Proof Obligations ϕ_{End} for Future Out-of-bounds Accesses)
The proof obligations ϕ_{End} for future out-of-bounds accesses are defined as $\phi_t(p\_asm,\ p\_asserted,\ p\_streams,\ p\_assert)$ with template parameters:

- $p\_asm \quad := \{i \mid w_p \leq i \leq N\}$,
- $p\_asserted \quad := \{i \mid 2 \cdot w_p \leq i < 3 \cdot w_p\}$,
- $p\_streams \quad := \{i \mid 2 \cdot w_p \leq i \leq N\}$,
- $p\_assert \quad := \{i \mid 3 \cdot w_p \leq i \leq N\}$

where $N = 3 \cdot w_p + w_f$.

So far, we have defined proof obligations for certain positions in the stream execution with and without out-of-bounds accesses. Together, the proof obligations constitute an inductive argument for the correctness of the assertions, see Proposition 14. Here, the base case is given by Definition 8.3 and induction steps are given by Definition 8.4 and Definition 8.5. The induction steps use the induction hypothesis, i.e., valid assertions, due to the template parameter *p_asserted*.

Proposition 14 (Assertion Verification by Lola Unfolding).
The set of assertions is correct if the formula $\phi_{Begin} \wedge \phi_{Run} \wedge \phi_{End}$ is valid.

To prove that Proposition 14 holds, we distinguish exhaustively four specification cases: no temporal accesses (Proposition 15), past temporal accesses only (Proposition 16), future temporal accesses only (Proposition 17), and past and future temporal accesses (Proposition 18). First, in the case without temporal accesses, we show that all the necessary obligations for a Hoare triple are encoded in the formula and that this formula only evaluates to *false* if the assertions are not satisfied while all assumptions are.

Proposition 15 (Assertion Verification for Zero Offsets)**.**
For a Lola *specification with $w_p = 0$ and $w_f = 0$, the set of assertions is correct if the formula $\phi_{Begin} \wedge \phi_{Run} \wedge \phi_{End}$ is valid.*

Proof. We prove the proposition by showing that the formula encodes an argument for the correctness of the assertions. The full proof is given in Section A.1.1. □

→Sec. A.1.1, p. 143

Next, we show that all necessary obligations for temporal accesses to previous stream values are encoded. Further, we show that an encoding of an inductive argument is provided that considers all possible combinations of out-of-bounds accesses at the beginning of stream execution as the base case and a monitoring step with no out-of-bounds accesses as the inductive step.

Proposition 16 (Assertion Verification for Past Offsets Only)**.**
For a Lola *specification with $w_p > 0$ and $w_f = 0$, the set of assertions is correct if the formula $\phi_{Begin} \wedge \phi_{Run} \wedge \phi_{End}$ is valid.*

Proof. We prove the proposition by showing that the formula encodes an argument for the correctness of the assertions. The full proof is given in Section A.1.2. □

→Sec. A.1.2, p. 144

Example 8.1.1. Consider the Lola specification

```
assume<a1> reset[-1, false] ∨ reset
input reset : Bool
output o1 := if reset then 0 else o1[-1, 0] + 1
output o2 := o1[-1, 0] + o1
assert<a1> 0 ≤ o2 and o2 ≤ 3
```

→Listing 8.2, p. 120

that simplifies Listing 8.2, e.g., only past offset accesses are used. Here, w_p is 1 and w_f is 0. The unfolding of ϕ_{Begin} checks all possible combinations of out-of-bounds accesses of annotations, i.e., `reset[-1, false]`, and outputs, i.e., `o1[-1, 0]`. In comparison to Figure 8.1, ϕ_{Begin} would only require $N = 0$ and $N = 1$ without future accesses. ϕ_{Run} represents the induction step where no default values are taken. △

→Fig. 8.1, p. 121

After specifications with past offsets only, we now consider future-only specifications. Similar to before, we show that an encoding of an inductive argument is provided. In contrast to before, the encoding must also prove that default values do not violate assertions at the end of a stopped stream execution.

Proposition 17 (Assertion Verification for Future Offsets Only)**.**
For a Lola *specification with $w_p = 0$ and $w_f > 0$, the set of assertions is correct if the formula $\phi_{Begin} \wedge \phi_{Run} \wedge \phi_{End}$ is valid.*

Proof. We prove the proposition by showing that the formula encodes an argument for the correctness of the assertions. The full proof is given in Section A.1.3. □

→Sec. A.1.3, p. 147

Example 8.1.2. Consider the LOLA specification

```
assume<a1> reset ∨ reset[1, false]
input reset : Bool
output o1 := if reset then 0 else o1[1, 0] + 1
output o2 := o1[1, 0] + o1
assert<a1> 0 ≤ o2 and o2 ≤ 3
```

that simplifies Listing 8.2, e.g., only future offset accesses are used. Here, w_p is 0 and w_f is 1. The unfolding of ϕ_{Begin} checks all possible combinations of out-of-bounds accesses of annotations, i.e., `reset[1, false]`, and outputs, i.e., `o1[1, 0]`. ϕ_{Run} represents the induction step where no default values are taken. △

→Listing 8.2, p. 120

Finally, the next case provides an inductive argument for LOLA specification with past and future temporal accesses. The inductive argument incorporates default values in the base case at the beginning and at the end of stream executions. Further, it handles all combinations of past and future out-of-bounds accesses.

Proposition 18 (Assertion Verification for Non-Zero Offsets)**.**
For a LOLA specification with $w_p > 0$ and $w_f > 0$, the set of assertions is correct if the formula $\phi_{Begin} \wedge \phi_{Run} \wedge \phi_{End}$ is valid.

Proof. We prove the proposition by showing that the formula encodes an argument for the correctness of the assertions. The full proof is given in Section A.1.4. □

→Sec. A.1.4, p. 149

Example 8.1.3. Consider the LOLA specification in Listing 8.2, i.e., $w_p = 1$ and $w_f = 1$. The unfolding of ϕ_{Begin} checks all possible combinations of out-of-bounds accesses of annotations, i.e., `reset[-1, false]` and `reset[1, false]` , and outputs, i.e., `o1[-1, 0]` and `o1[1, 0]`. ϕ_{Run} represents the induction step where no default values are taken. △

→Listing 8.2, p. 120

By proving all cases of temporal accesses in the Proposition 15 to Proposition 18, the Proposition 14 is proven – the verification approach is sound. Soundness refers to the ability of an analyzer to prove the absence of errors — if a LOLA specification is accepted, it is guaranteed that the assertions are not violated. The converse does not hold, i.e., the presented verification approach is not complete. Completeness refers to the ability of an analyzer to prove the presence of errors — if a LOLA specification is rejected, the counter-example given should be a valid stream execution that results in an assertion violation. The following LOLA specification is rejected even though no assertion is violated:

```
1 input a: Int32
2 assume <a1> a ≤ 10
3 output sum := if sum[-1, 0] ≤ 10 then 0 else sum[-1, 0] + a
4 assert <a1> sum ≤ 100
```

Here, since the `if`-condition in Line 3 evaluates to *true* at the beginning of the stream execution, `sum` is a constant stream with value zero. Hence, the assertion in Line 3 is never violated. The verification approach rejects this specification. The reason for this is that `sum` ≤ 100 is added as an *asserted* condition in ϕ_{Run}. Therefore, the SMT solver can assign a value between 91 and 100 to the earliest `sum` variable of the unfolding, resulting in an assertion violation of the next `sum` variable.

8.2. Tool: SpecVerifier

The tool SpecVerifier is based on the open source RTLola framework[1]. Specifically, it uses the Lola frontend to parse a given specification into an intermediate representation. Based on this representation, the SMT formulas are created and evaluated using the Rust z3 crate[2]. At its current phase of the crate's development, a combined solver is implemented that internally uses either a non-incremental or an incremental solver. There is no information on the implemented tactics available, but all our requests could be solved within seconds. For functions that are not natively supported by the Rust Z3 solver, the output is arbitrarily chosen by the solver with respect to the range of the function. The tool expects a Lola specification augmented by *assumptions* and *assertions*. The verification is done automatically and produces a counter-example stream execution, if any exists. The counter-example can then be used by the user to debug its specifications. Two different kinds of users are targeted. First, users that write the entire augmented specification. Such a user could be a system engineer who is developing a safety monitor and wants to ensure that it contains critical properties. Second, users that augment an existing specification. Here, one reason could be that an existing monitor shall be composed with other critical components and certain behavioral properties are expected. Also, similar to software testing, the task of writing a specification and their respective assumptions and assertions could be separated between two users to ensure the independence of both.

8.3. Application Experience in Avionics

In this section, we present details about the tool experiences on practical avionic specifications. To gain practical tool experience, previously written specifications based on interviews with engineers of the German Aerospace Center [149] were extended by assumptions and assertions. The previous specifications were tested using log files and simulations – the authors considered them correct. We report several specification errors in Table 8.4 that were detected by the presented verification extension. In fact, the detected errors would have resulted in undetected failures. After the errors in the previous specifications were fixed, all assertions were proven correct. Note that the errors could have been found by manual reviews. However, such reviews are tedious and error-prone, especially when temporal behaviors are involved. The detected errors in Table 8.4 can be grouped into three classes: *Classical Bugs*, *Operator Errors*, and *Wrong*

[1]`https://rtlola.org/`

[2]`https://docs.rs/z3/0.9.0/z3/`

Specification	**Appx.**	**#o**	**#a**	**#g**	**Detected errors**
gps_vel_output	A.1.5	14	6	6	–
gps_pos_output	A.1.5	19	3	10	–
imu_output	A.1.5	18	6	6	Wrong default value Division by zero
nav_output	A.1.5	25	3	5	Missing abs()
tagging	A.1.5	6	2	2	–
ctrl_output	A.1.5	25	7	8	Wrong threshold comparisons
mm_output_1	A.1.5	4	1	2	–
mm_output_2	A.1.5	17	6	9	Missing if condition Wrong default value
contingency_output	A.1.5	4	8	1	Observation: both contingencies could be true in case of voting, ie both at 50%
health_output	A.1.5	1	5	1	–

Table 8.4.: Detected errors by the verification extension, where #o, #a, and #g represent the number of outputs, assumptions, and assertions given in the specification, respectively.

Interpretations. Classical bugs are errors that occur when implementing an algorithm. Operator errors are LOLA specific errors, e.g., temporal accesses. Last, wrong interpretations refer to gaps between the specification and the user's design intend, e.g., violated assertions due to incomplete specifications. Next, we give one representative example for each group. We reduced the specification to the representative fragment.

Example 8.3.1 (Classical Bug)**.**
The LOLA specification in Listing 8.3 monitors the fuel level. A monitor shall notify the operator when one of the three different fuel levels are reached: half (Line 8), warning (Line 9), and danger (Line 10). The fuel level is computed as a percentage in Line 7. It uses the fuel level at the beginning of the flight (Line 6) as a reference for its computation. Given the documentation of the fuel sensor, it is known that `fuel` values are within $\mathbb{R}^+$ and decreasing. This is formalized in Line 4 as an assumption. As an invariant, we

```
1  // Inputs
2  input fuel: Float64
3  // Assumptions
4  assume<a5> fuel > 0.0 and fuel < fuel[-1, fuel + 0.1]
5  // Outputs
6  output start_fuel := start_fuel[-1, fuel]
7  output fuel_level := 1.0 - (start_fuel - fuel) / start_fuel
8  output fuel_half := fuel_level < 0.50
9  output fuel_warning := fuel_level < 0.25
10 output fuel_danger := fuel_level < 0.10
11 trigger_once fuel_half "INFO: Fuel is below 50%"
12 trigger_once fuel_warning "WARNING: Fuel is below 25%"
13 trigger_once fuel_danger "DANGER: Fuel is below 10%"
14 // Assertions
15 assert<a5> start_fuel >= fuel
16         and (fuel_half[-1, false] -> fuel_half)
17         and (fuel_warning[-1, false] -> fuel_warning)
18         and (fuel_danger[-1, false] -> fuel_danger)
```

Listing 8.3: The fixed version of the Lola ctrl_output specification that monitors the fuel level. Three level of engagement are depicted: half, warning, and danger.

asserted that the starting fuel is greater or equal to `fuel` (Line 15). Further, in Lines 16 to 18, we stated that once a level is reached it should remain at this level. During our experiment, the assertion led to a counter-example that pointed to the previously used and erroneous fuel level computation:

```
output fuel_level := (start_fuel - fuel) / start_fuel
```

In short, the output computed the consumed fuel and not the remaining fuel. The computation could be easily fixed by converting consumed fuel into remaining fuel, see Line 7. Thereafter, Listing 8.3 satisfies its assertion. Note, that offset accesses were used to assert the temporal behavior of the fuel level output stream. Further, `trigger_once` is an abbreviation which states that only the first raising edge is reported to the user.

△

Example 8.3.2 (Operator Error)**.**
An important monitoring property is to detect frozen values as these indicate a deteriorated sensor. Such a specification is depicted in Listing 8.4. Here, as an input, the acceleration in x-direction is given. The frozen value check is computed from Line 6 to Line 10. It compares previous values using Lola's offset operator. To check this computation, we added the sanity check that asserts that no frozen value shall be detected (Line 13) when small changes in the input are present (Line 4). In the previous version, the frozen values were computed using the abbreviated offset operator:

```
output frozen_ax := ax[-5..0, 0.0, =]
```

```
// Inputs                                                    1
input ax: Float32                                            2
// Assumptions                                               3
assume <a1> ax != ax[-1, ax + ε]                             4
// Outputs                                                   5
output frozen_ax := ax[-5, 0.1] = ax[-4, 0.0]                6
              and ax[-4, 0.0] = ax[-3, 0.0]                  7
              and ax[-3, 0.0] = ax[-2, 0.0]                  8
              and ax[-2, 0.0] = ax[-1, 0.0]                  9
              and ax[-1, 0.0] = ax                           10
trigger frozen_ax "WARNING: x-acceleration is frozen!"       11
// Assertions                                                12
assert <a1> !frozen_ax                                       13
```

Listing 8.4: The LOLA imu_output specification that monitors frozen acceleration values.

This resulted in a counter-example that pointed to wrong default values. Although the abbreviated version is easier to read and reduces the size of the specification, it is unfortunately not suitable for this kind of property. The tool detected the unlikely situation that the first value of `ax` is 0.0 which would have resulted in evaluating `frozen_ax` to true. Although unlikely, this should be avoided as contingencies activated in such situations depend on correct results and otherwise could harm people on the ground. By unfolding the operator and adding a different default value to one of the past accesses, the error was resolved (Line 6). Listing 8.4 shows the fixed version which satisfies its assertion.

△

Example 8.3.3 (Wrong Interpretation)**.**
In Listing 8.5, two visual sensor readings are received (Lines 2-3). Both, readings argue over the same observations where `avgDist` represents the average distance to the measured obstacle, `actual` is the number of measurements, and `static` is the number of unchanged measurements. A simple rating function is introduced (Lines 5-6) that estimates the corresponding rating – the higher the better. Using these ratings, the trust in each of the sensors is computed probabilistically (Lines 7-8). When considering the integration of such a monitor as an ASTM F3269-21 switch condition that decides which sensor value should be forwarded, the specification should be revised. This is the case because the assertion in Line 12 produces a counter-example which indicates that both trust triggers (Lines 9-10) can be activated at the same time. A common solution for this problem is to introduce a priority between the sensors. △

The examples show how the presented LOLA verification extension can be used to find errors in specifications. We also noticed that the annotations can serve as documentation. System assumptions are often implicitly known during development and are (sometimes) documented in natural language in separate files. Having these as-

```
// Inputs
input avgDist_laser, actual_laser, static_laser: Float64, Float64, Float64
input avgDist_optical, actual_optical, static_optical: Float64, Float64, Float64
// Outputs
output rating_laser := 0.2 * static_laser + 0.4 * actual_laser + 0.4 * avgDist_laser
output rating_optical := 0.2*static_optical + 0.4*actual_optical + 0.4*avgDist_optical
output trust_laser := rating_laser / ( rating_laser + rating_optical)
output trust_optical := 1.0 - trust_laser
trigger trust_laser >= 0.5
trigger trust_optical >= 0.5
// Assertions
assert <a1> trust_laser != trust_optical
```

Listing 8.5: The Lola contingency_output specification that uses an heuristic to decide which sensor is more trustworthy.

sumptions explicitly stated within the monitor specification potentially reduces future mistakes when reusing the specification, e.g., when composing with other monitor specifications. Listing 8.6 depicts such an example specification. Here, the monitor interfaces are clearly defined by the domain of input a (Line 5) and output o (Line 13). Also, reset is assumed to be valid at least once per second (Line 5). Further, no deeper understanding of the internal computations (Lines 7-10) is required in order to safely compose this specification with others.

```
1  // Inputs with frequency 5Hz
2  input a: Float64
3  input reset: Bool
4  // Assumptions
5  assume <a1> 0.0 ≤ a ≤ 1.0 and reset[-4..0, false, ∨]
6  // Outputs
7  output o_1 := ...
8  ...
9  output o_n := ...
10 output o := o_1 + ... + o_n
11 trigger o ≥ 0.5 "Warning: Output o exceeds threshold!"
12 // Assertions
13 assert <a1> 0.0 ≤ o ≤ 1.0
```

Listing 8.6: Lola specification annotations describe interface properties.

8.4. Monitoring Assumptions

The presented verification approach offers an analysis of the specification that can guarantee the desired behavior of the monitor outputs. Yet, these guarantees are often based on assumptions that can be violated at runtime. Further, a violated assumption does not directly result in a violated assertions. As an example consider Listing 8.4

that states an assertion which is violated when six consecutive ax values are the same. Hence, the sequence $\langle 0.1; 0.1; 0.1; 0.1; 0.4; 0.5\rangle$ satisfies the assertion. Yet, it violates the assumption at the first positions.

In this section, we will give a translation of an annotated specification into a specification that checks assumptions at runtime and efficiently activates and deactivates assertion checks. Further, we will present experimental results showing that the verification extension not only provides static guarantees, but that translating it into a corresponding specification can lead to better runtime performance compared to a monitor that simply checks all assertions during runtime.

8.4.1. Translation into Lola 2.0

We translate an annotated Lola specification into a Lola 2.0 specification [35]. Lola 2.0 extends Lola by streams that can be spawned, filtered, and closed at runtime. For the translation, we only need to handle assumptions and assertions since trigger, input, and output streams are directly supported by Lola 2.0.

We replace each assumption by a parameterized output stream o. A stream `output` o `: Bool spawn when` e_s `eval when` e_f `with` $e(t_1, \ldots, t_m, s_1, \ldots, s_n)$ `close` e_c is an output stream that is created when its spawn condition e_s is true. It then starts producing values by evaluating its computation e if its filter condition e_f is *true* until its close condition e_c is satisfied. Since a violated assumption can influence previous and future assertions due to temporal offset accesses, the computation e is a counter that represents how many assertions are impacted by the assumption violation. If the counter is positive, then an assumption was violated that influences an assertion computation at the current time. To compute the impact of a violated assumption, we take the maximum between the longest chain of offset accesses from assertion to assumption plus one. This is achieved by an analysis of the dependency graph [130]. To start the counter, we use the negated assumption as spawn condition. Further, we close the stream when the impact of a violated assumption is over, i.e., when the counter is zero. As filter condition, we use *true* to decrease the counter in each step.

We also represent assertion checks by output streams. For each assertion, we use an output stream that is only extended if one of its corresponding assumptions is violated, i.e., its counter value is positive. We also add a trigger to report assertion violations.

As an example consider the annotated Lola specification

```
input vel : Float32 // Velocity                                   1
assume <a> -20.0 ≤ vel ≤ 20.0                                     2
output vel_max := max(abs(vel), vel_max[-1, 0.0])                 3
trigger vel_max > 20.0 "Velocity threshold exceeded!"             4
assert <a> abs(vel[-2..0, 0.0, +]) / 3.0 ≤ 20.0                   5
```

that checks the maximal velocity value (Line 3) and the average velocity over a discrete window of three (Line 5). The assumption (Line 2) and the assertion (Line 5) are transformed to

```
output assumption
   spawn when !(-20.0 ≤ vel ≤ 20.0)
   eval when true with if -20.0 ≤ vel ≤ 20.0 then assumption[-1,0] - 1 else 3
   close when assumption = 0
trigger assumption > 0 "Assumption violated!"
output assertion
   spawn when true
   eval when assumption > 0 with abs(vel[-2..0, 0.0, +]) / 3.0 ≤ 20.0
   close when false
trigger !assertion "Assertion violated!"
```

The output `vel_max` and the trigger remain unchanged. Note that the trigger could have been replaced by an assertion. Yet, this would not reduce as much overhead as for the window check which the following experiments will show.

8.4.2. Experiments

For our experiments, we compare the performance of the presented translation to a naive translation that checks assumptions and assertions independently in each execution step. As annotated specification, we use

```
input a_i: Float64
assume <a> a_i ≤ 2.0
assert <a> a_i[-w..0, 0.0, +] ≤ w · 2.0
```

that we scale in the number of annotation pairs using the variable i and the computational load of the assertion by the window variable w. For instance, $i = 10$ and $w = 5$ produces ten inputs with the corresponding annotations where each assertion takes the sum over the last five input values including the current one. The naive translation v_n with omitted triggers is

```
input a_i: Float64
output assumption_i := a_i ≤ 2.0
output assertion_i := a_i[-w..0, 0.0, +] ≤ w · 2.0
```

The presented translation v_t also with omitted triggers is

```
input a_i: Float64
output assumption_i
   spawn when a_i > 2.0
   eval when true with if a_i ≤ 2.0 then assumption_i[-1,0] - 1 else w+1
   close when assumption_i = 0
output assertion_i
   spawn when true
   eval when assumption_i > 0 with a_i[-w..0, 0.0, +] ≤ w · 2.0
   close when false
```

For the experiments, the considered values of i were 5, 10, and 15 and the values of w were 0, 5, and 10. The experiments were conducted on three different kinds of log files: no assumption is violated, all assumptions are violated, half of the assumptions are violated. Each log file contains 10.000.000 events that were sufficient to report the average time in nanoseconds required by the monitor to evaluate one input event. Each experiment was carried out three times and the average was taken. For the experiments, an eight-core machine with an 2.5GHz Intel i7 processor with 32GB RAM was used.

The results of the experiments are depicted in Figure 8.5. As can be seen in Figure 8.5a, which considers log files with no violation of assumptions, version v_t significantly improves runtime by up to 64.06%. It can also be seen that version v_t improves the required time per event by 8.17% already in the case of simple assertions. Further, the required time for v_t remained constant while increasing the window size which shows that no unnecessary assertion checks were computed; in contrast to v_n, where the required time correlates with the size of the window. Next, Figure 8.5b considers log files where all assumptions are violated. The results show that this time v_t correlates with the size of the window similar to v_n since all the assertions need to be checked due to violated assumptions. The experiments show that v_t adds an overhead of up to -33.86%. However Figure 8.5c shows that already in the case where half of the inputs violate the assumptions and a more complex assertion is used ($w = 5$), the LOLA 2.0 specification version v_t pays off and outperforms v_n by up to 15.91%. The results are also graphically depicted in Figure 8.6.

Overall, the experiments show that translating an annotated specification into a LOLA 2.0 specification can be used to report assertion violations due to violated assumptions efficiently at runtime. Since assumptions are generally expected to be satisfied in the nominal case, the translation also improves the monitor's runtime without losing its guarantees. Especially complex assertions based on simple assumptions benefit from the translation. If the assertions are simple, the benefits from the translation are negligible.

Remark 8.1 (Alternative LOLA assumption encoding)

We also considered the alternative LOLA assumption encoding

```
output assumption_i
:= if a_i ≤ 2.0 then if assumption_i[-1,0] = 0 then 0 else assumption_i[-1,0] - 1 else w+1
```

that no longer uses LOLA 2.0 features. Our result showed a runtime improvement of up to 59.17% in the case of no violations and a runtime deterioration of only up to -8.54%. Still, we decided on the LOLA 2.0 assumption encoding to gain the best performance, since assumptions should not be violated in the nominal case. Yet, these results indicate that the parameterization of the assumptions has the largest share in the reported deterioration.

Window w	**Number of inputs i**								
	5			**10**			**15**		
	$\mathbf{v_n}$ [μs]	$\mathbf{v_t}$ [μs]	$\mathbf{\Delta}$ [%]	$\mathbf{v_n}$ [μs]	$\mathbf{v_t}$ [μs]	$\mathbf{\Delta}$ [%]	$\mathbf{v_n}$ [μs]	$\mathbf{v_t}$ [μs]	$\mathbf{\Delta}$ [%]
0	1203.67	1105.34	8.17	2404.67	2096.67	12.81	3674.67	3236.34	11.93
5	1951.67	1118.00	42.72	4146.34	2127.67	48.69	6318.00	3273.34	48.19
10	2859.67	1124.34	60.68	5985.00	2151.00	64.06	8977.00	3228.00	64.04

(a) None of the 10.000.000 events in the log file violates the assumption.

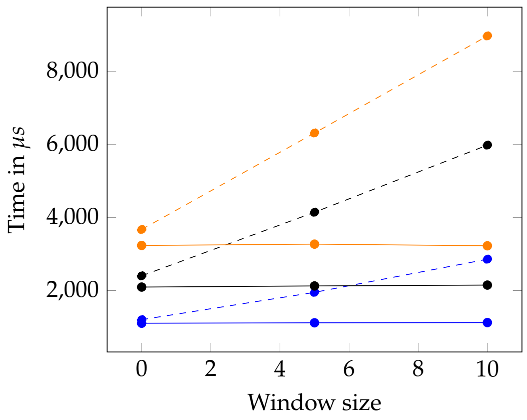
Window w	**Number of inputs i**								
	5			**10**			**15**		
	$\mathbf{v_n}$ [μs]	$\mathbf{v_t}$ [μs]	$\mathbf{\Delta}$ [%]	$\mathbf{v_n}$ [μs]	$\mathbf{v_t}$ [μs]	$\mathbf{\Delta}$ [%]	$\mathbf{v_n}$ [μs]	$\mathbf{v_t}$ [μs]	$\mathbf{\Delta}$ [%]
0	2371.34	3092.67	−30.42	4637.67	6165.34	−32.94	6922.34	9266.34	−33.86
5	3124.00	3881.34	−24.24	6376.00	7956.34	−24.79	9440.34	11842.34	−25.44
10	3972.34	4844.00	−21.94	8203.67	9852.00	−20.09	12157.00	14513.67	−19.39

(b) All of the 10.000.000 events in the log file violate the assumption.

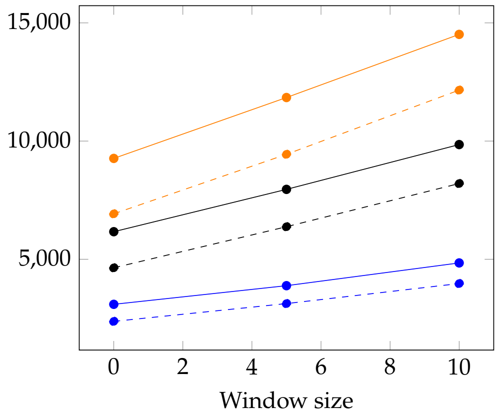
Window w	**Number of inputs i**								
	5			**10**			**15**		
	$\mathbf{v_n}$ [μs]	$\mathbf{v_t}$ [μs]	$\mathbf{\Delta}$ [%]	$\mathbf{v_n}$ [μs]	$\mathbf{v_t}$ [μs]	$\mathbf{\Delta}$ [%]	$\mathbf{v_n}$ [μs]	$\mathbf{v_t}$ [μs]	$\mathbf{\Delta}$ [%]
0	1783.67	2099.00	−17.68	3505.00	4163.00	−18.77	5311.00	6270.34	−18.06
5	2535.00	2515.00	0.79	5222.34	5043.00	3.43	7839.00	7518.34	4.09
10	3409.34	2967.00	12.97	7071.00	6032.34	14.69	10560.67	8880.67	15.91

(c) Half of the 10.000.000 events in the log file violate the assumption.

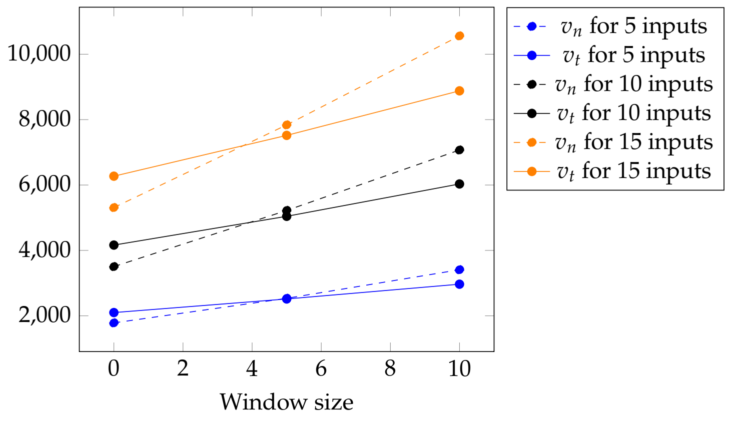
Figure 8.5.: The results of the log file analyses using the translations of an annotated specification is given. Entries in the table represent the time required by the monitor for one input event. The specification version $\mathbf{v_n}$ represents a specification that checks assumptions and assertion for each event in the log file whereas the specification version $\mathbf{v_t}$ checks an assertions only if its corresponding assumption is violated by the use of output streams that use spawn, filter, and close conditions. The symbol $\mathbf{\Delta}$ represents the runtime effect of dynamic assertion checks, i.e., positive values indicate improvement and negative values indicate deterioration.



(a) No violations, see Figure 8.5a



(b) All violated, see Figure 8.5b



(c) Half violated, see Figure 8.5c

Figure 8.6.: Graphical representation of Figure 8.5.

8.5. Summary

As both the relevance and the complexity of cyber-physical systems continue to grow, runtime monitoring is an essential ingredient of safety-critical systems. When monitors are derived from specifications it is crucial that the specifications are correct. In this thesis, we have presented a sound verification approach for the stream-based monitoring language Lola. With this approach, the developer can formally prove guarantees on the streams computed by the monitor, and hence ensure that the monitor does not cause dangerous situations. The verification extension is motivated by upcoming aviation regulations and standards as well as by practical feedback of engineers.

The extension has been applied to previously written Lola specifications that were obtained based on interviews with aviation experts. In this process, we discovered and fixed several serious specification errors.

Further, since assumption can fail during runtime, they must be monitored and only when they are violated, their respective assertions need to be monitored as well. In this thesis, we have efficiently monitored verified guarantees at runtime. Our experiments have shown that our Lola 2.0 encoding can significantly improve the monitors performance while maintaining a low overhead in case of few assumption violations. Yet, this improvement is highly dependent on the given specification. In general, simple assumptions and complex assertions benefit from this approach.

Chapter 9

Conclusion

In this thesis, we contributed to advancing the use of runtime verification for autonomous CPS. We explored ways to bridge the gap between unmanned aviation and formal methods, presented temporal behavior trees as specification language to express CPS behaviors by retrofitting monitoring to the control structure behavior trees, developed novel offline algorithms to provide engineers with feedback from monitoring results, investigated online monitoring to safeguard machine learning components and safe operation, and supported the assurance of specification correctness. In the following, we summarize the results of this thesis and outline directions for future work.

9.1. Part I: Foundations

In the first part of this thesis, we recapped notions of CPS concepts (with a focus on unmanned aviation) for formal methods researchers, and conversely presented formal methods (with a focus on runtime verification) for CPS experts, thereby fostering mutual understanding.

In particular, we presented aviation standards and regulations that yield safe systems and showed how requirements of these documents map to monitoring approaches. We observed that monitoring requires a holistic perspective for autonomous CPS, as low-level sensor data propagate to high-level decision making. To address this, we introduced a hierarchy of monitoring properties and provided example formalized specifications at different levels of this hierarchy. Finally, we discussed that, despite the clear benefits of runtime verification, its adaptation in industry remains limited, possibly due to factors such as limited tool support and qualification challenges.

We then presented specification languages and introduced the novel specification language temporal behavior trees, which combines behavior trees with common temporal specification languages to express complex behaviors. A key feature of this language is its ability to combine different specification languages at its leaf nodes while being easily retrofitted to existing behavior trees commonly used to control robotic applications.

Relation to the Challenges in Section 1.2

This part addressed the first and second challenge: the hierarchy of monitoring properties offers a way to *think about complex systems when writing and managing specifications* (first challenge) and temporal behavior trees provide a formalism *to define [...] a "sandbox"* for autonomous CPS (second challenge). As an example of such a sandbox, consider the TBT in Figure 4.7 that defines the autonomous lander of a UAV on a ship using only the most essential parts of a correct landing maneuver.

→ Fig. 4.7, p. 68

9.2. Part II: Offline Monitoring

In the second part of this thesis, we developed novel offline algorithms for analyzing autonomous CPS post-execution, with the goal of improving these systems.

First, we formulated the segmentation problem, which assigns segments of a trace to corresponding nodes in the TBT. This allows to better understand which parts of the segmentation requires more focus during development. We provided a dynamic programming algorithm to efficiently compute the robustness of a trace w.r.t. its TBT specification, as well as two approximation techniques: subsampling and lazy evaluation. In empirical evaluations, we demonstrated the use of segmentation for analyzing human behavior and for improving the autonomous landing capabilities of a UAV on a ship.

Besides detecting segments of the trace that violate specifications, we formulated the trace repair problem, which minimally adjusts trace events to achieve satisfaction of the TBT specification. We provided an optimal mixed-integer linear program formulation for trace repair, along with two repair strategies which improve scalability of this complex task. The incremental repair uses segmentation information to avoid repairing the whole trace but incrementally repairing it, starting with the leaf nodes. The landmark-based repair is an iterative approach that finds a minimal set of propositions that is sufficient to satisfy the specification. Empirical evaluations showed that both methods efficiently repair traces in case studies on a robotic search task and an autonomous ship deck landing of a UAV.

Relation to the Challenges in Section 1.2

This part addressed the third challenge "*how can verdicts produced by offline monitoring be processed and presented to developers to yield insights that help to refine and improve the system*". Trace segmentation given a TBT specification helps to identify which parts of the specification are violated and therefore require further development. Additionally, trace repair takes a trace that violates a TBT specification and minimally modifies it such that the modified trace satisfies the TBT specification, thereby helping developers to understand what changes are needed.

9.3. Part III: Online Monitoring

In the third part of this thesis, we introduced tools that ease the integration of monitors into CPS, providing monitoring verdicts during execution.

First, we developed a ROS adapter for RTLola that simplifies the integration of RTLola monitors into ROS applications. Using RTLola, we specified the cross-validation of an untrustworthy machine-learning object detector with a traditional trustworthy LiDAR-based blob detector to reduce false positives. This specification highlighted the advantages of runtime verification for complex properties: it is concise and naturally handles asynchronous inputs and real-time aspects. In a flight test, we showed how the cross-validation improved the safe landing of an unmanned aircraft on a vertiport, safeguarding a machine-learning component.

Next, we considered the use of runtime verification for safe operation monitoring. We provided ALAADy's Run-Time Assurance architecture with a safety monitor at its core, which switched to recovery actions to mitigate safety risks. This monitor is based on RTLola specifications for geofencing, flight path monitoring, and progress monitoring. To streamline development, the tool FencyCreator helped us to rapidly adjust specifications during development of the Concept of Operations by generating RTLola specification based on KML-files. In simulation, we demonstrated a geofencing monitor that triggered safe flight termination and thereby guided engineers to find the correct thresholds for the termination. In flight tests, we further demonstrated the use of flight path monitoring and progress monitoring, especially for improving the situational awareness of a safety pilot on the ground.

Last, we addressed the concern that specifications themselves may contain errors and proposed a method to detect them. We extended Lola by Hoare triples and provided an SMT encoding for such an annotated specification. We developed an offline verification algorithm that provided the functional correctness of an assumption/assertion-annotated Lola specification, implemented in the tool SpecVerifier. We applied SpecVerifier to existing avionics specifications and uncovered previously unknown critical bugs. Further, we considered the case when assumptions do not hold during runtime. We provided an online monitoring algorithm that monitors assumptions and dynamically switches to monitoring assertions when related assumptions fail, reverting back once assumptions hold again for sufficient time. Experimental result showed that this approach is not only necessary as assumption can fail, but is also beneficial for runtime performance when having simple assumptions but complex assertions.

Relation to the Challenges in Section 1.2

This part addressed the second challenge *"how to define specifications for online monitoring autonomous CPS that act as "sandbox""*. As forms of such a "sandbox", cross-validation can be used for machine-learning based components and operational constraints such as geofencing help the ensure the safe operation of an autonomous CPS.

9.4. Future Work

We outline directions for future work.

Best-Effort Temporal Behavior Tree Monitoring

During development, robots often operate in a *best-effort* manner: a task may only be partially completed before the robot transitions to the next. This may occur due to environment constraints on the robot behavior, time limitations, or other prioritization decisions. *Best-Effort* Temporal Behavior Tree monitoring captures such imperfections. It allows developers to identify situations during execution where the robot proceeds to the next task without fully succeeding in the current one. The idea is conceptually related to segmentation, where one finds the best mapping of a trace, but here it is applied online where future actions are still unknown. Unlike segmentation, e.g., the fallback operator reports all attempted executions of its child nodes, rather than reporting only the last and best one.

Trace Segmentation 2.0

Trace segmentation aims to decompose execution traces into meaningful segments that best align with the TBT specification. In future, to accelerate finding the best and alternative segmentations, the presented dynamic programming approach could exploit parallelism. Further, it could apply an adaptive subsampling where the step size varies across different portions of the trace instead of choosing a fixed one. Moreover, we envision a deep integration of trace segmentation into the CI/CD workflow, where automated monitoring continuously analyzes updates to controller or similar components to provide timely feedback to developers. Such a workflow would detect regression in behavior satisfaction, thereby supporting faster iterations and more reliable development.

Trace Repair 2.0

An interesting direction is to explore alternative cost functions that favor repairs that minimize the length of the repaired trace. This requires to remove the constraint of the current one-to-one mapping between the original and the repaired trace. Another direction is to leverage trace repair as a teacher for reinforcement learning, focusing on situations where agents fail their task and need guidance. A repaired trace provides such guidance, which enables the agent to efficiently improve its strategy over time.

Verifying Specifications

We plan to develop automatic invariant generation for Lola specifications, which will reduce the manual effort required to establish correctness. Further, we aim to extend the verification to RTLola in order to capture richer timing aspects and provide stronger guarantees for real-time systems. For temporal behavior trees, such a verification is also important as transitions between leaf nodes naturally encode situations where the task switches similar to activating a contingency in response to certain conditions.

Monitoring Autonomous CPS Using External Knowledge

Information that goes beyond raw sensor signals is especially valuable for monitoring autonomous CPS. A promising research direction is to provide monitors with direct access to knowledge stored in ontologies, knowledge bases, or databases as part of the specification language. This adds challenges in designing efficient data structures that allow the monitor to process such complex information in real time. For example, spatial monitors may need to query large sets of objects in a database, even though only a subset is relevant for evaluating a given spatial property. Data structures such as quadtrees have already demonstrated benefits in motion planning by efficiently skipping updates for objects that were classified as irrelevant [150]. Similar techniques could also be adapted to monitoring applications, improving scalability and responsiveness of spatial monitors.

☺

Appendix **A**

Appendix

A.1. Monitoring with Verified Guarantees

A.1.1. Proof of Proposition 15

Proof. The set of assertions $\psi_1, \ldots, \psi_n$ is correct for all input streams if and only if whenever an assumption is satisfied, its corresponding assertion is satisfied as well. Without loss of generality, let $\Gamma = \{\alpha\}$, $assume(\alpha) = \{\theta\}$, and $assert(\alpha) = \{\psi\}$. We prove the proposition by showing that the formula encodes an argument for the correctness of the assertions. We consider the case that $w_p = 0$, $w_f = 0$, and the formula $\phi_{Begin} \wedge \phi_{Run} \wedge \phi_{End}$ is valid.

In this case, we do not have past or future out-of-bounds accesses. Hence, the obligations consider a single *Run* step with $N = 0$ and template parameters:

- $p\_asm \quad := \{i \mid i = 0\}$,
- $p\_asserted \quad := \emptyset$,
- $p\_streams \quad := \{i \mid i = 0\}$,
- $p\_assert \quad := \{i \mid i = 0\}$.

By instantiating the template for *Run*, we encode the following obligations:

$$\begin{aligned}
\phi_{Run} &\overset{\text{Def. 8.3}}{=} \phi_t(p\_asm,\ p\_asserted, p\_streams,\ p\_assert) \\
&\overset{\text{Def. 8.2}}{=} \bigwedge_{i \in \{0\}} smt(\theta)(i) \ \wedge \bigwedge_{i \in \emptyset} smt(\psi)(i) \wedge \bigwedge_{i \in \{0\}} \Big(\bigwedge_{0 < k \leq n} \zeta_k = smt(e_k)(i) \Big) \\
&\quad \rightarrow \bigwedge_{i \in \{0\}} smt(\psi)(i)
\end{aligned}$$

θ	ζ	ψ	$\theta \wedge \zeta \to \psi$	
0	0	0	1	assumption invalid
0	0	1	1	assumption invalid
0	1	0	1	assumption invalid
0	1	1	1	assumption invalid
1	0	0	1	outputs invalid
1	0	1	1	outputs invalid
1	1	0	0	incorrect
1	**1**	**1**	**1**	correct

Table A.1.: Truth table for the encoding of the Hoare triple.

$$= \; smt(\theta)(0) \; \wedge \bigwedge_{0<k\leq n} \zeta_k = smt(e_k)(0) \to \; smt(\psi)(0)$$

As can be seen, the formula encodes "*assume* $\wedge$ *program* $\to$ *assertion*" for a single position. Note that a single position suffices since no temporal dependencies exist. Table A.1 shows that the formula evaluates to $false$ only if an assertion evaluates to false, although the assumptions are valid and the output computations behave as expected. Conversely, if the formula is valid, then the set of assertions is correct or assumptions violated.
Note that no further obligations are added by ϕ_{Begin} and ϕ_{End} since the same template instances are created.

$$\begin{aligned}
\phi_{Begin} &\overset{\text{Def. 8.3}}{=} \bigwedge_{0\leq N<1} \phi_t(\{i \mid 0 \leq i \leq N\}, \emptyset, \{i \mid 0 \leq i \leq N\}, \{i \mid 0 \leq i < 1\}) \\
&= \; \phi_t(\{0\}, false, \{0\}, \{0\}) \overset{\text{Def. 8.4}}{=} \phi_{Run} \\
\phi_{End} &\overset{\text{Def. 8.5}}{=} \phi_t(\{i \mid 0 \leq i \leq 0\}, \emptyset, \{i \mid 0 \leq i \leq 0\}, \{i \mid 0 \leq i \leq 0\}) \\
&= \; \phi_t(\{0\}, false, \{0\}, \{0\}) \overset{\text{Def. 8.4}}{=} \phi_{Run}
\end{aligned}$$

□

A.1.2. Proof of Proposition 16

Proof. The set of assertions $\psi_1, \ldots, \psi_n$ is correct for all input streams if and only if whenever an assumption is satisfied, its corresponding assertion is satisfied as well.

Without loss of generality, let $\Gamma = \{\alpha\}$, $assume(\alpha) = \{\theta\}$, and $assert(\alpha) = \{\psi\}$. We prove the proposition by showing that the formula encodes an inductive argument for the correctness of the assertions. We consider the case that $w_p > 0$, $w_f = 0$, and the formula $\phi_{Begin} \land \phi_{Run} \land \phi_{End}$ is valid.

In this case, we consider only accesses to the past. The formula encodes a k-induction where ϕ_{Begin} encodes the base cases and ϕ_{Run} the step case.
The template parameter for ϕ_{Begin} are:

- p_asm $:= \{i \mid 0 \leq i \leq N\}$,
- $p\_asserted$ $:= \emptyset$,
- $p\_streams$ $:= \{i \mid 0 \leq i \leq N\}$,
- p_assert $:= \{i \mid 0 \leq i < \min(N+1, 2 \cdot w_p)\}$.

$$\phi_{Begin} \overset{\text{Def. 8.3}}{=} \bigwedge_{0 \leq N < 2 \cdot w_p} \phi_t(p\_asm, p\_asserted, p\_streams, p\_assert)$$

$$\overset{\text{Def. 8.2}}{=} \bigwedge_{0 \leq N < 2 \cdot w_p} \left(\bigwedge_{i \in p\_asm} smt(\theta)(i) \land \bigwedge_{i \in p\_streams} \left(\bigwedge_{0 < k \leq n} \zeta_k = smt(e_k)(i) \right) \right.$$
$$\left. \rightarrow \bigwedge_{i \in p\_assert} smt(\psi)(i) \right)$$

ϕ_{Begin} encodes stream executions that handle all possible out-of-bounds accesses. The range of stream execution N from 0 to $2 \cdot w_p$ covers all combinations of stream and assumption out-of-bounds scenarios. Two times the w_p is required to handle cases where an output accesses available past values while at the same position the access of an assumption is out-of-bounds. Figure A.2 depicts an example for $w_p = 1$. The sub-figure on the top shows a stream execution where both the output computation and the assumption have out-ouf-bounds accesses. The sub-figure on the bottom shows a stream execution where the output access to the past at the last position is in-bound but at the accessed position the access of the used assumption is not.

ϕ_{Run} unfolds a stream execution and uses the inductive argument. The unfolding is depicted in Figure A.3. It shows that a stream execution is unfolded with an assertion at position $3 \cdot w_p$. The inductive argument is depicted as *asserted*. The *asserted* unfolding is based on the base-cases of the k-induction where $k = w_p$. Further, our encoding ensures that no out-of-bounds accesses occur by sufficiently unfolding the *assumptions* and *outputs*.

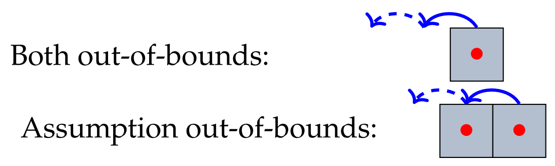


Figure A.2.: All stream executions of ϕ_{Begin} for $w_p = 1$ are depicted. It shows the case where possibly both accesses are out-of-bounds (upper) and the case where only an assumption is out-of-bounds (lower). A gray-colored box indicates that an assumption has been unfolded at this position, while a red dotted box indicates that an assertion has been unfolded at this position. Solid and dashed arrows indicate accesses by streams and annotations, respectively.

The template parameter for ϕ_{Run} are:

- $p\_asm \quad := \{i \mid w_p \leq i \leq 3 \cdot w_p\}$,
- $p\_asserted \quad := \{i \mid 2 \cdot w_p \leq i < 3 \cdot w_p\}$,
- $p\_streams \quad := \{i \mid 2 \cdot w_p \leq i \leq 3 \cdot w_p\}$,
- $p\_assert \quad := \{i \mid i = 3 \cdot w_p\}$.

and $N = 3 \cdot w_p$.

$$\begin{aligned}
\phi_{Run} &\overset{\text{Def. 8.4}}{=} \phi_t(p\_asm,\ p\_asserted, p\_streams, p\_assert) \\
&\overset{\text{Def. 8.2}}{=} \bigwedge_{i \in p\_asm} smt(\theta)(i) \wedge \bigwedge_{i \in p\_asserted} smt(\psi)(i) \wedge \bigwedge_{i \in p\_streams} \Big(\bigwedge_{0<k\leq n} \zeta_k = smt(e_k)(i) \Big) \\
&\quad \rightarrow \bigwedge_{i \in p\_assert} smt(\psi)(i)
\end{aligned}$$

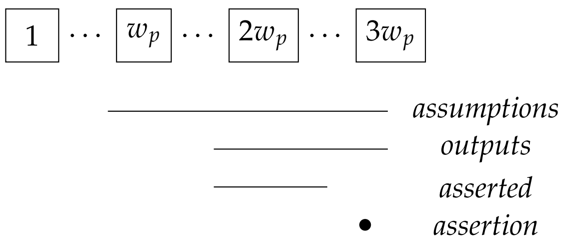


Figure A.3.: The unfolding of a stream execution is depicted. The assertion is proven at Position $3 \cdot w_p$. *Assumptions* and *outputs* are unfolded such that all accesses can be resolved. The inductive argument that is shown in the induction base is represented by *asserted*.

Since $w_f = 0$, no future out-of-bounds access can occur and, therefore, no obligations are added by ϕ_{End}.

$$\phi_{End} \overset{\text{Def. 8.5}}{=} \phi_t\left(\substack{\{i|w_p \leq i \leq 3\cdot w_p\}, \quad \{i|2\cdot w_p \leq i < 3\cdot w_p\}, \\ \{i|2\cdot w_p \leq i \leq 3\cdot w_p\},\ \{i|3\cdot w_p \leq i \leq 3\cdot w_p\}}\right)$$

$$= \phi_t(p\_asm, p\_asserted, p\_streams, p\_assert) \overset{\text{Def. 8.4}}{=} \phi_{Run}$$

The formula $\phi_{Begin} \wedge \phi_{Run} \wedge \phi_{End}$ encodes a k-induction where $k = w_p$:

$$\phi_{Begin} \begin{cases} assume(0) \wedge program(0) \rightarrow assert(0) \\ \ldots \\ assume(k) \wedge program(k) \rightarrow assert(k) \end{cases}$$

$$\wedge\ \phi_{Run} \left\{ assume(n) \wedge program(n) \rightarrow assert(n) \right.$$

Similar to the first case, for each base case and step case, if the formula is valid then the assertions must be correct. □

A.1.3. Proof of Proposition 17

Proof. The set of assertions $\psi_1, \ldots, \psi_n$ is correct for all input streams if and only if whenever an assumption is satisfied, its corresponding assertion is satisfied as well. Without loss of generality, let $\Gamma = \{\alpha\}$, $assume(\alpha) = \{\theta\}$, and $assert(\alpha) = \{\psi\}$. We prove the proposition by showing that the formula encodes an inductive argument for the correctness of the assertions. We consider the case that $w_p = 0$, $w_f > 0$, and the formula $\phi_{Begin} \wedge \phi_{Run} \wedge \phi_{End}$ is valid.

In this case, only future stream accesses are considered. The formula encodes a k-induction where ϕ_{End} represent the base cases and ϕ_{Run} the step case. By unfolding a stream execution of length w_f, ϕ_{End} covers all possible future out-of-bounds combinations for stream as well as annotation accesses. Figure A.4 depicts ϕ_{End} where $N = w_f$. The cases are similar to the base cases in the second case but this time for the future accesses. The template parameter for ϕ_{End} are:

- $p\_asm \quad := \{i \mid 0 \leq i \leq w_f\}$,
- $p\_asserted \quad := \emptyset$,
- $p\_streams \quad := \{i \mid 0 \leq i \leq w_f\}$,
- $p\_assert \quad := \{i \mid 0 \leq i \leq w_f\}$.

and $N = w_f$.

$$\phi_{End} \overset{\text{Def. 8.5}}{=} \phi_t(p\_asm, p\_asserted, p\_streams, p\_assert)$$

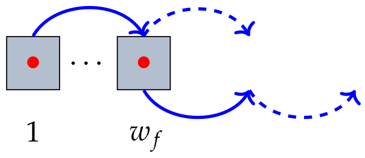


Figure A.4.: A stream executions of length w_f is depicted. It shows the case where only the last assumption access is out-of-bounds. A gray-colored box indicates that an assumption has been unfolded at this position, while a red dotted box indicates that an assertion has been unfolded at this position. Solid and dashed arrows indicate accesses by streams and annotations, respectively.

$$\overset{\text{Def. 8.2}}{=} \bigwedge_{i \in p\_asm} smt(\theta)(i) \wedge \bigwedge_{i \in \emptyset} smt(\psi)(i) \wedge \bigwedge_{i \in p\_streams} \Big(\bigwedge_{0<k\leq n} \zeta_k = smt(e_k)(i) \Big)$$
$$\rightarrow \bigwedge_{i \in p\_assert} smt(\psi)(i)$$

The induction step ϕ_{Run} unfolds a stream execution and uses the inductive argument. The unfolding is depicted in Figure A.5. In this case, the *assertion* is at the first position and the k base cases hold for possible future out-of-bounds accesses, i.e., at the next w_f positions. Further, *outputs* and *assumptions* are unfolded to consider all necessary *output* computations and *assertion* accesses.
The template parameter for ϕ_{Run} are:

- p_asm $:= \{i \mid 0 \leq i \leq 2 \cdot w_f\}$,
- $p\_asserted$ $:= \{i \mid 0 \leq i \leq w_f \wedge i \neq 0\}$,
- $p\_streams$ $:= \{i \mid 0 \leq i \leq w_f\}$,
- p_assert $:= \{i \mid i = 0\}$.

and $N = 3 \cdot w_f$.

$$\phi_{Run} \overset{\text{Def. 8.4}}{=} \phi_t(p\_asm, p\_asserted, p\_streams, p\_assert)$$
$$\overset{\text{Def. 8.2}}{=} \bigwedge_{i \in p\_asm} smt(\theta)(i) \wedge \bigwedge_{i \in p\_asserted} smt(\psi)(i) \wedge \bigwedge_{i \in p\_streams} \Big(\bigwedge_{0<k\leq n} \zeta_k = smt(e_k)(i) \Big)$$
$$\rightarrow \bigwedge_{i \in p\_assert} smt(\psi)(i)$$

Note that ϕ_{Begin} adds no further obligations to the formula since a single execution suffices to include all possible out-of-bounds values.

The formula $\phi_{Begin} \wedge \phi_{Run} \wedge \phi_{End}$ encodes a k-induction where $k = w_f$:

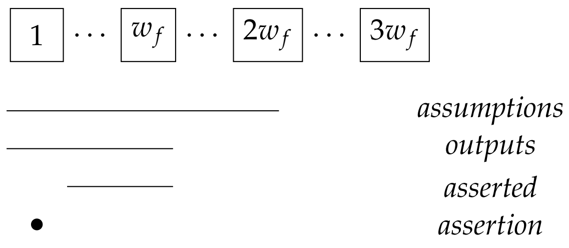


Figure A.5.: The unfolding of a stream execution is depicted. The assertion is proven at the first position. *Assumptions* and *outputs* are unfolded such that all accesses can be resolved. The inductive argument that is shown in the induction base is represented by *asserted*.

$$\phi_{End}\begin{cases} assume(0)\ \wedge\ program(0) \rightarrow assert(0) \\ \dots \\ assume(k)\ \wedge\ program(k) \rightarrow assert(k) \end{cases}$$

$$\wedge\ \phi_{Run}\begin{cases} assume(n)\ \wedge\ program(n) \rightarrow assert(n) \end{cases}$$

Similar to the past-only case, for each base case and step case, if the formula is valid then the assertions must be correct. □

A.1.4. Proof of Proposition 18

Proof. The set of assertions $\psi_1, \dots, \psi_n$ is correct for all input streams if and only if whenever an assumption is satisfied, its corresponding assertion is satisfied as well. Without loss of generality, let $\Gamma = \{\alpha\}$, $assume(\alpha) = \{\theta\}$, and $assert(\alpha) = \{\psi\}$. We prove the proposition by showing that the formula encodes an inductive argument for the correctness of the assertions. We consider the case that $w_p > 0$, $w_f > 0$, and the formula $\phi_{Begin}\ \wedge\ \phi_{Run}\ \wedge\ \phi_{End}$ is valid.

Further, we consider combinations of past and future accesses. Hence, all combinations of out-of-bounds accesses need to be covered. The formula ϕ_{Begin} covers all possible combinations of past out-of-bounds accesses with future accesses. The combinations are depicted in Figure A.6: $N = 0$ represents the case that all accesses are out-of-bounds, $N = w_p$ represents the case that except of the past output stream access at the last position all other accesses are out-of-bounds and vice versa for the future stream access at the first position, $N = 2 \cdot w_p$ represents the case that only accesses of annotations are out-of-bounds, $N = 2 \cdot w_p + w_f$ represents both cases that only one annotation is out-of-bounds, and $N = 2 \cdot (w_p + w_f) - 1$ represents the case that only the past access of an annotation is out-of-bounds.
The template parameter for ϕ_{Begin} are:

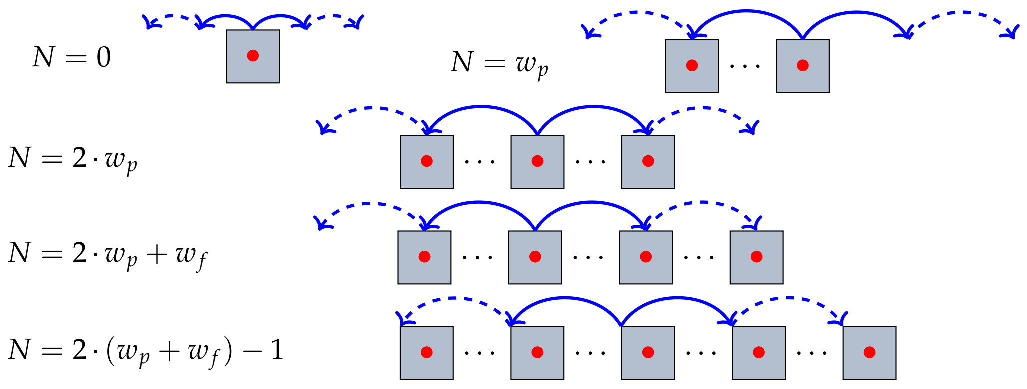


Figure A.6.: All combinations of past out-of-bounds accesses with future accesses are depicted.

- $p\_asm \quad := \{i \mid 0 \leq i \leq N\}$,
- $p\_asserted \quad := \emptyset$,
- $p\_streams \quad := \{i \mid 0 \leq i \leq N\}$,
- $p\_assert \quad := \{i \mid 0 \leq i < \min(N+1, 2 \cdot w_p)\}$.

$$\phi_{Begin} \overset{\text{Def. 8.3}}{=} \bigwedge_{0 \leq N < 2 \cdot (w_p + w_f)} \phi_t(p\_asm, p\_asserted, p\_streams, p\_assert)$$

$$\overset{\text{Def. 8.2}}{=} \bigwedge_{0 \leq N < 2 \cdot (w_p + w_f)} \Bigg(\bigwedge_{i \in p\_asm} smt(\theta)(i) \wedge \bigwedge_{i \in \emptyset} smt(\psi)(i)$$

$$\wedge \bigwedge_{i \in p\_streams} \Big(\bigwedge_{0 < k \leq n} \zeta_k = smt(e_k)(i) \Big)$$

$$\rightarrow \bigwedge_{i \in p\_assert} smt(\psi)(i) \Bigg)$$

Since ϕ_{Begin} covers only combinations of past out-of-bounds accesses with future accesses, ϕ_{End} covers future out-of-bounds accesses only. The unfolding is depicted in Figure A.7. The *assertion*s are unfolded such that all combinations of future out-of-bounds accesses are covered: annotation only and stream and annotation combined. Since the first k base cases are already covered, they can be used here as *asserted*. Further, *outputs* and *assumptions* are unfolded such that all required accesses are available.

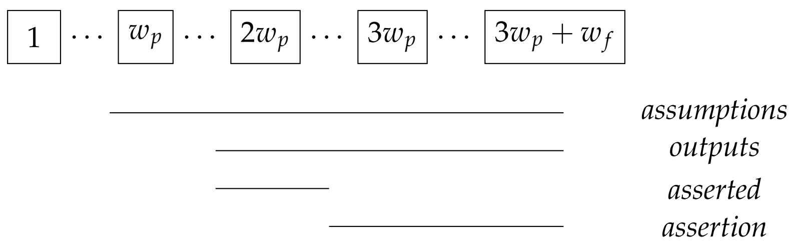


Figure A.7.: The unfolding at the end of a stream execution is depicted. *Assumptions* and *outputs* are unfolded such that all accesses can be resolved. The inductive argument that is shown in the induction base is represented by *asserted*.

The template parameter for ϕ_{End} are:

- p_asm $:= \{i \mid w_p \leq i \leq 3 \cdot w_p + w_f\}$,
- $p\_asserted$ $:= \{i \mid 2 \cdot w_p \leq i < 3 \cdot w_p\}$,
- $p\_streams$ $:= \{i \mid 2 \cdot w_p \leq i \leq 3 \cdot w_p + w_f\}$,
- p_assert $:= \{i \mid 3 \cdot w_p \leq i \leq 3 \cdot w_p + w_f\}$.

and $N = 3 \cdot w_p + w_f$.

$$\begin{aligned}
\phi_{End} &\overset{\text{Def. 8.5}}{=} \phi_t(p\_asm, p\_asserted, p\_streams, p\_assert) \\
&\overset{\text{Def. 8.2}}{=} \bigwedge_{i \in p\_asm} smt(\theta)(i) \wedge \bigwedge_{i \in p\_asserted} smt(\psi)(i) \\
&\quad \wedge \bigwedge_{i \in p\_streams} \Big(\bigwedge_{0<k\leq n} \zeta_k = smt(e_k)(i) \Big) \\
&\quad \rightarrow \bigwedge_{i \in p\_assert} smt(\psi)(i)
\end{aligned}$$

Last, we need to show that the assertions also hold for the case that no out-of-bounds access exists. Hence, the unfolding is encoded as depicted in Figure A.8. The *assertion* is proven at position $3 \cdot w_p$ and already shown assertions of past and future accesses are incorporated by *asserted*. Further, all required accesses of *outputs* and *assumptions* are unfolded such that no out-of-bounds access exists.
The template parameter for ϕ_{Run} are:

- p_asm $:= \{i \mid w_p \leq i \leq 3 \cdot w_p + 2 \cdot w_f\}$,
- $p\_asserted$ $:= \{i \mid 2 \cdot w_p \leq i \leq 3 \cdot w_p + w_f \wedge i \neq 3 \cdot w_p\}$,
- $p\_streams$ $:= \{i \mid 2 \cdot w_p \leq i \leq 3 \cdot w_p + w_f\}$,
- p_assert $:= \{i \mid i = 3 \cdot w_p\}$.

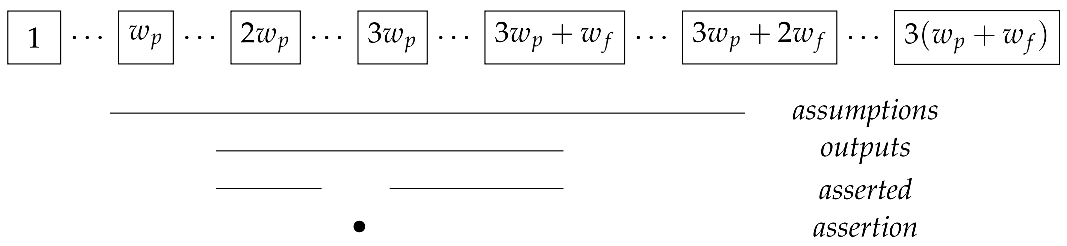


Figure A.8.: The unfolding of a stream execution during run is depicted. The assertion is proven at position $3 \cdot w_p$. *Assumptions* and *outputs* are unfolded such that all accesses can be resolved. The inductive argument that is shown in the induction base is represented by *asserted*.

and $N = 3 \cdot (w_p + w_f)$.

$$\begin{aligned}
\phi_{Run} &\overset{\text{Def. 8.4}}{=} \phi_t(p\_asm,\ p\_asserted, p\_streams, p\_assert) \\
&\overset{\text{Def. 8.2}}{=} \bigwedge_{i \in p\_asm} smt(\theta)(i) \wedge \bigwedge_{i \in p\_asserted} smt(\psi)(i) \\
&\quad \wedge \bigwedge_{i \in p\_streams} \Big(\bigwedge_{0<k\leq n} \zeta_k = smt(e_k)(i) \Big) \\
&\quad \rightarrow \bigwedge_{i \in p\_assert} smt(\psi)(i)
\end{aligned}$$

Similar to the other cases, for each base case and step case, if the formula is valid then the assertions must be correct.

□

A.1.5. Specifications – Experience Report

Specification: *gps_vel_output*

```
input sol_age: Float32
input hor_spd: Float32
input trk_gnd: Float32
input vert_spd: Float32
input time_s: UInt64
input time_us: UInt64
// Assumptions
assume <a1> time − time.offset(by: −1).defaults(to: time − 0.1) > 0.0
  and time − time.offset(by: −1).defaults(to: time − 0.1) <= 0.1
  and trace_pos >= 0

```

```
assume <a2> time − time.offset(by: −1).defaults(to: time − 0.1) > 0.0
  and time − time.offset(by: −1).defaults(to: time − 0.1) <= 0.1
  and trace_pos >= 0
// Frequency computations
output time := cast(time_s) + cast(time_us) / 1000000.0
output start_time := if time.offset(by: −1).defaults(to: −1.0) = −1.0 then time else
     start_time.offset(by: −1).defaults(to: −1.0)
output flight_time := time − start_time
output trace_pos @ sol_age or hor_spd or trk_gnd or vert_spd or time_s or time_us :=
     trace_pos.offset(by: −1).defaults(to: −1) + 1
output frequency :=
      1.0 / ( time − time.offset(by: −1).defaults(to: time − 0.0001) )
output freq_sum :=
      freq_sum.offset(by: −1).defaults(to: 0.0) + frequency
output freq_avg := freq_sum / cast(trace_pos+1)
output freq_max := if frequency > freq_max.offset(by: −1).defaults(to: frequency) then
     frequency else freq_max.offset(by: −1).defaults(to: frequency)
output freq_min := if frequency < freq_min.offset(by: −1).defaults(to: frequency) then
     frequency else freq_min.offset(by: −1).defaults(to: frequency)
// Speed computations
output hor_spd_max := if hor_spd > hor_spd_max.offset(by: −1).defaults(to: 0.0) then
     hor_spd else hor_spd_max.offset(by: −1).defaults(to: 0.0)
output vert_spd_max := if vert_spd > vert_spd_max.offset(by: −1).defaults(to: 0.0) then
     vert_spd else vert_spd_max.offset(by: −1).defaults(to: 0.0)
// Solution age and track over ground (motion direction wrt. north)
trigger sol_age <= 0.5 "Sol age should remain zero!"
output trk_gnd_in_bound := if trk_gnd >= 0.0 and trk_gnd <= 360.0 then
     trk_gnd_in_bound.offset(by: −1).defaults(to: true) else false
output trk_gnd_max := if trk_gnd > trk_gnd_min.offset(by: −1).defaults(to: 0.0) then
     trk_gnd else trk_gnd_min.offset(by: −1).defaults(to: 0.0)
output trk_gnd_min := if trk_gnd < trk_gnd_max.offset(by: −1).defaults(to: 0.0) then
     trk_gnd else trk_gnd_max.offset(by: −1).defaults(to: 0.0)
// Assertions
assert <a1> time.offset(by: −1).defaults(to: −1.0) < time
  and start_time == start_time.offset(by: −1).defaults(to: start_time)
  and flight_time >= flight_time.offset(by: −1).defaults(to: 0.0)
assert <a2> frequency >= 10.0
  and freq_sum >= freq_sum.offset(by: −1).defaults(to: 0.0) + 10.0
assert <a3> trk_gnd_in_bound.offset(by: −1).defaults(to: true)
  or !trk_gnd_in_bound
```

Specification: *gps_pos_output*

```
import math
input lat: Float32
input lon: Float32
input hgt: Float32
input nObjs: UInt64
input nGPSL1: UInt64
input time_s: UInt64
input time_us: UInt64
// Assumptions
assume <a1> time − time.offset(by: −1).defaults(to: time − 0.1) > 0.0
  and time − time.offset(by: −1).defaults(to: time − 0.1) <= 0.1
  and trace_pos >= 0
// Frequency computations
output time: Float32 := cast(time_s) + cast(time_us) / 1000000.0
output start_time := if time.offset(by: −1).defaults(to: −1.0) == −1.0 then time else
    start_time.offset(by: −1).defaults(to: −1.0)
output flight_time := time − start_time
output trace_pos @ lat or lon or hgt or nObjs or nGPSL1 or time_s or time_us :=
    trace_pos.offset(by: −1).defaults(to: −1) + 1
output frequency := 1.0 / ( time − time.offset(by: −1).defaults(to: time − 0.0001) )
output freq_sum := freq_sum.offset(by: −1).defaults(to: 0.0) + frequency
output freq_avg := freq_sum / cast(trace_pos+1)
output freq_max := if frequency > freq_max.offset(by: −1).defaults(to: 0.0) then frequency
    else freq_max.offset(by: −1).defaults(to: 0.0)
output freq_min := if frequency < freq_min.offset(by: −1).defaults(to: 0.0) then frequency
    else freq_min.offset(by: −1).defaults(to: 0.0)
// Statistics
output lat_max := if lat > lat_max.offset(by: −1).defaults(to: lat) then lat else
    lat_max.offset(by: −1).defaults(to: lat)
output lat_min := if lat < lat_min.offset(by: −1).defaults(to: lat) then lat else
    lat_min.offset(by: −1).defaults(to: lat)
output lon_max := if lon > lon_max.offset(by: −1).defaults(to: lon) then lon else
    lon_max.offset(by: −1).defaults(to: lon)
output lon_min := if lon < lon_min.offset(by: −1).defaults(to: lon) then lon else
    lon_min.offset(by: −1).defaults(to: lon)
output lat_in_bound := max( abs(lat_max), abs(lat_min) ) <= 90.0
output lon_in_bound := max( abs(lon_max), abs(lon_min) ) <= 180.0
trigger !lat_in_bound "Irregular latitude value!"
trigger !lon_in_bound "Irregular longitude value!"
output begin := false
output start_height := if begin.offset(by: −1).defaults(to: true) then hgt else
    start_height.offset(by: −1).defaults(to: 0.0)
output hgt_inc_max := max( hgt_inc_max.offset(by: −1).defaults(to: 0.0), hgt − start_height
    )
output hgt_dec_max := min( hgt_dec_max.offset(by: −1).defaults(to: 0.0) , hgt −
    start_height )
trigger hgt_inc_max > 100.0 "Never increase height by more than 100m!"
trigger hgt_dec_max < −100.0 "Never decrease height by more than 100m"
// Assertions
assert <a1> time.offset(by: −1).defaults(to: −1.0) < time
```

```
  and start_time == start_time.offset(by: −1).defaults(to: start_time)
  and flight_time >= flight_time.offset(by: −1).defaults(to: 0.0)
assert <a2> hgt_inc_max >= 0.0 and hgt_dec_max <= 0.0
  and hgt_inc_max >= hgt_inc_max.offset(by: −1).defaults(to: 0.0)
  and hgt_dec_max <= hgt_inc_max.offset(by: −1).defaults(to: 0.0)
  and start_height = start_height.offset(by: −1).defaults(to: start_height)
  and (lat_in_bound.offset(by: −1).defaults(to: true) or !lat_in_bound)
  and (lon_in_bound.offset(by: −1).defaults(to: true) or !lon_in_bound)
```

Specification: *imu_output*

```
import math
input ax: Float32
input ay: Float32
input az: Float32
input time_s: UInt64
input time_us: UInt64
input counter: Int64
// Assumptions
assume <a1> time − time.offset(by: −1).defaults(to: time − 0.1) > 0.0
  and time − time.offset(by: −1).defaults(to: time − 0.1) <= 0.1
  and trace_pos >= 0
assume <a2> ax != ax.offset(by: −1).defaults(to: ax + 0.1)
  and ay != ay.offset(by: −1).defaults(to: ay + 0.1)
  and az != az.offset(by: −1).defaults(to: az + 0.1)
// Frequency computations
output time := cast(time_s) + cast(time_us) / 1000000.0
output start_time := if time.offset(by: −1).defaults(to: −1.0) == −1.0 then time else
    start_time.offset(by: −1).defaults(to: −1.0)
output flight_time := time − start_time
output trace_pos @ ax or ay or az or time_s or time_us or counter := trace_pos.offset(by:
    −1).defaults(to: −1) + 1
output frequency := 1.0 / ( time − time.offset(by: −1).defaults(to: time − 0.0001) )
output freq_sum := freq_sum.offset(by: −1).defaults(to: 0.0) + frequency
output freq_avg := freq_sum / cast(trace_pos+1)
// Statistics
output deviation := abs( frequency − 100.0)
output exceeds_worst := deviation > worst_dev.offset(by: −1).defaults(to: 0.0)
output worst_dev_pos := if exceeds_worst then trace_pos else worst_dev_pos.offset(by:
    −1).defaults(to: 0)
output worst_dev := if exceeds_worst then deviation else worst_dev.offset(by:
    −1).defaults(to: 0.0)
output ax_max := max(abs(ax),ax_max.offset(by:−1).defaults(to:0.0))
output ay_max := max(abs(ay),ay_max.offset(by:−1).defaults(to:0.0))
output az_max := max(abs(az),az_max.offset(by:−1).defaults(to:0.0))
trigger ax > 15.0 or ay > 15.0 or az > 15.0
```

```
output frozen_ax := ax.offset(by:−1).defaults(to:0.0) = ax
  and ax.offset(by:−2).defaults(to:0.0)=ax.offset(by:−1).defaults(to:0.0)
  and ax.offset(by:−3).defaults(to:0.0)=ax.offset(by:−2).defaults(to:0.0)
  and ax.offset(by:−4).defaults(to:0.0)=ax.offset(by:−3).defaults(to:0.0)
  and ax.offset(by:−5).defaults(to:0.1)=ax.offset(by:−4).defaults(to:0.0)
output frozen_ay := ay.offset(by:−1).defaults(to:0.0) = ay
  and ay.offset(by:−2).defaults(to:0.0)=ay.offset(by:−1).defaults(to:0.0)
  and ay.offset(by:−3).defaults(to:0.0)=ay.offset(by:−2).defaults(to:0.0)
  and ay.offset(by:−4).defaults(to:0.0)=ay.offset(by:−3).defaults(to:0.0)
  and ay.offset(by:−5).defaults(to:0.1)=ay.offset(by:−4).defaults(to:0.0)
output frozen_az := az.offset(by:−1).defaults(to:0.0) = az
  and az.offset(by:−2).defaults(to:0.0)=az.offset(by:−1).defaults(to:0.0)
  and az.offset(by:−3).defaults(to:0.0)=az.offset(by:−2).defaults(to:0.0)
  and az.offset(by:−4).defaults(to:0.0)=az.offset(by:−3).defaults(to:0.0)
  and az.offset(by:−5).defaults(to:0.1)=az.offset(by:−4).defaults(to:0.0)
trigger frozen_ax or frozen_ay or frozen_az
output check_counter := if trace_pos = 0 then false else (counter != (counter.offset(by:
    −1).defaults(to: −1) + 1) % 100)
trigger check_counter "A counter value was ignored."
// Assertions
assert <a1> time.offset(by: −1).defaults(to: −1.0) < time
  and start_time == start_time.offset(by: −1).defaults(to: start_time)
  and flight_time >= flight_time.offset(by: −1).defaults(to: 0.0)
assert <a2> !frozen_ax and !frozen_ay and !frozen_az
```

Specification: *nav_output*

```
import math
input lat: Float32
input lon: Float32
input ug: Float32
input vg: Float32
input wg: Float32
input time_s: UInt64
input time_us: UInt64
// Assertion
assume <a1> trace_pos >= 0
  and time − time.offset(by: −1).defaults(to: time − 0.1) <= 0.1
  and time − time.offset(by: −1).defaults(to: time − 0.1) > 0.0
// Frequency Computation
output time := cast(time_s) + cast(time_us) / 1000000.0
output start_time := if time.offset(by: −1).defaults(to: −1.0) == −1.0 then time else
    start_time.offset(by: −1).defaults(to: −1.0)
output flight_time := time − start_time
output trace_pos @lat or lon or ug or vg or wg or time_s or time_us := trace_pos.offset(by:
    −1).defaults(to: −1) + 1
```

```
output frequency := 1.0 / ( time − time.offset(by: −1).defaults(to: time − 0.0001) )
output freq_sum := freq_sum.offset(by: −1).defaults(to: 0.0) + frequency
output freq_avg := freq_sum / cast(trace_pos+1)
output freq_max := if frequency > freq_max.offset(by: −1).defaults(to: frequency) then
      frequency else freq_max.offset(by: −1).defaults(to: frequency)
output freq_min := if frequency < freq_min.offset(by: −1).defaults(to: frequency) then
      frequency else freq_min.offset(by: −1).defaults(to: frequency)
// Statistics
output velocity := sqrt( ug*ug + vg*vg + wg*wg)
output lon1_rad := lon.offset(by: −1).defaults(to: 0.0) * 3.1415926535 / 180.0
output lon2_rad := lon * 3.1415926535 / 180.0
output lat1_rad := lat.offset(by: −1).defaults(to: 0.0) * 3.1415926535 / 180.0
output lat2_rad := lat * 3.1415926535 / 180.0
output dlon := lon2_rad − lon1_rad
output dlat := lat2_rad − lat1_rad
output a := (sin(dlat/2.0))*(sin(dlat/2.0)) + cos(lat1_rad) * cos(lat2_rad) *
      (sin(dlon/2.0))*(sin(dlon/2.0))
output x_atan2 := sqrt(a)
output y_atan2 := sqrt(1.0−a)
output c := 2.0 * if x_atan2 > 0.0 then arctan(y_atan2/x_atan2)
  else if x_atan2 < 0.0 and y_atan2 >= 0.0
    then arctan(y_atan2/x_atan2) + 3.1415926535
  else if x_atan2 < 0.0 and y_atan2 < 0.0
    then arctan(y_atan2/x_atan2) − 3.1415926535
  else if x_atan2 = 0.0 and y_atan2 > 0.0 then 3.1415926535 / 2.0
  else if x_atan2 = 0.0 and y_atan2 < 0.0 then −3.1415926535 / 2.0
  else 0.0
output gps_distance := 6373000.0 * c
output passed_time := time − time.offset(by: −1).defaults(to: 0.0)
output distance_max := velocity * passed_time
output dif_distance := abs( gps_distance − distance_max )
output detected_jump :=if trace_pos=0 then false else dif_distance>1
trigger detected_jump "Jump!"
// Assertions
assert <a1> time.offset(by: −1).defaults(to: −1.0) < time
  and start_time == start_time.offset(by: −1).defaults(to: start_time)
  and flight_time >= flight_time.offset(by: −1).defaults(to: 0.0)
assert <a2> (!detected_jump or gps_distance > distance_max)
  or (!detected_jump or distance_max > gps_distance)
```

Specification: *tagging*

```
import math
input time_s: UInt64
input time_us: UInt64
input vel: Float64
```

```
// Assumptions
assume<a1> (time_s = time_s.offset(by: −1).defaults(to: 0)
  and time_us > time_us.offset(by: −1).defaults(to: 0))
  and ( time_s > time_s.offset(by: −1).defaults(to: 0)
       or time_us > time_us.offset(by: −1).defaults(to: 0))
// Exemplary State Statistics
output time := cast(time_s) + cast(time_us) / 1000000.0
output correct_vel := abs( vel ) < 0.3
output cur_state := if correct_vel then
  if cur_state.offset(by: −1).defaults(to: 0) = 0 then 1 else 2 else 0
output start_interval := cur_state = 2
output interval_start := if start_interval then interval_start.offset(by: −1).defaults(to: 0.0)
     else time
trigger start_interval "Interval started!"
output end_interval := cur_state.offset(by: −1).defaults(to: 0) > 0 and !correct_vel and
     time_since_start > 5.0
trigger end_interval "Interval ended!"
output time_since_start := time − interval_start.offset(by: −1).defaults(to: 0.0)
// Assertions
assert <a1> !(start_interval and end_interval)
  and time_since_start > 0.0
```

Specification: *ctrl_output*

```
import math
input time_s: UInt64
input time_us: UInt64
input vel_x: Float64
input vel_y: Float64
input vel_z: Float64
input fuel: Float64
input power: Float64
input vel_r_x: Float64
input vel_r_y: Float64
input vel_r_z: Float64
// Assumptions
assume <a1> trace_pos >= 0
  and time − time.offset(by: −1).defaults(to: time − 0.1) <= 0.1
  and time − time.offset(by: −1).defaults(to: time − 0.1) > 0.0
assume<a2> power > 0.0
  and power <= power.offset(by: −1).defaults(to: power)
  and fuel > 0.0 and fuel < fuel.offset(by: −1).defaults(to: fuel + 0.1)
  and (time_s = time_s.offset(by: −1).defaults(to: 0)
  and time_us > time_us.offset(by: −1).defaults(to: 0))
  and (time_s > time_s.offset(by: −1).defaults(to: 0)
       or time_us > time_us.offset(by: −1).defaults(to: 0))
```

```
// Frequency computations
output time := cast(time_s) + cast(time_us) / 1000000.0
output start_time := if time.offset(by: −1).defaults(to: −1.0) == −1.0 then time else
      start_time.offset(by: −1).defaults(to: −1.0)
output flight_time := time − start_time
output trace_pos @ time_s or time_us or vel_x or vel_y or vel_z or fuel or power or vel_r_x
      or vel_r_y or vel_r_z := trace_pos.offset(by:−1).defaults(to:−1) + 1
output frequency := 1.0 / ( time − time.offset(by: −1).defaults(to: time − 0.0001) ) // major
      improvement
output freq_sum := freq_sum.offset(by: −1).defaults(to: 0.0) + frequency
output freq_avg := freq_sum / cast(trace_pos+1)
output freq_max := if frequency > freq_max.offset(by: −1).defaults(to: frequency) then
      frequency else freq_max.offset(by: −1).defaults(to: frequency)
output freq_min := if frequency < freq_min.offset(by: −1).defaults(to: frequency) then
      frequency else freq_min.offset(by: −1).defaults(to: frequency)
// Exemplary phase detection
output velocity := sqrt( vel_x∗vel_x + vel_y∗vel_y + vel_z∗vel_z )
output velocity_max := if reset_max.offset(by: −1).defaults(to: false) then velocity else max(
      velocity, velocity_max.offset(by: −1).defaults(to: 0.0) )
output velocity_min := if reset_max.offset(by: −1).defaults(to: false) then velocity else min(
      velocity, velocity_min.offset(by: −1).defaults(to: 0.0) )
output dif_max := abs(velocity_max − velocity_min)
output reset_max := dif_max > 1.0
output reset_time := if reset_max or trace_pos = 0 then time else reset_time.offset(by:
      −1).defaults(to: 0.0)
output unchanged := if reset_max.offset(by: −1).defaults(to: false) then 0 else
      unchanged.offset(by: −1).defaults(to: 0) + 1
trigger unchanged = 150 "Phase detected!"
// Statistics
output vel_dev := abs(vel_r_x−vel_x) + abs(vel_r_y−vel_y) + abs(vel_r_z−vel_z)
output dev_sum := vel_dev + dev_sum.offset(by: −1).defaults(to: 0.0)
output vel_av := dev_sum / cast((trace_pos+1)∗3)
output worst_dev_pos := if worst_dev.offset(by: −1).defaults(to: vel_dev − 1.0) < vel_dev
      then trace_pos else worst_dev_pos.offset(by: −1).defaults(to: 0)
output worst_dev := if worst_dev.offset(by: −1).defaults(to: vel_dev − 1.0) < vel_dev then
      vel_dev else worst_dev.offset(by: −1).defaults(to: 0.0)
output start_fuel := start_fuel.offset(by: −1).defaults(to: fuel)
output fuel_level := 1 − ( start_fuel − fuel ) / start_fuel
output fuel_half := fuel_level < 0.50
output fuel_warning := fuel_level < 0.25
output fuel_danger := fuel_level < 0.10
output start_power := start_power.offset(by: −1).defaults(to: power)
output power_p_consumed := ( (start_power − power) / (start_power) )
trigger_once fuel_half "INFO: Fuel level is half reduced"
trigger_once fuel_warning "WARNING: Fuel level is below 25%"
trigger_once fuel_danger "DANGER: Fuel level is below 10%"
trigger_once power_p_consumed > 0.50 "Power below half capacity"
trigger_once power_p_consumed > 0.75 "Power below quarter capacity"
trigger_once power_p_consumed > 0.90 "Urgent: Recharge Power!"
// Assertions
assert <a1> time.offset(by: −1).defaults(to: −1.0) < time
  and start_time == start_time.offset(by: −1).defaults(to: start_time)
  and flight_time >= flight_time.offset(by: −1).defaults(to: 0.0)
assert<a2> reset_time >= 0.0
  and start_fuel >= fuel and start_power >= power
  and (!fuel_half.offset(by: −1).defaults(to: false) or fuel_half)
  and (!fuel_warning.offset(by: −1).defaults(to: false)or fuel_warning)
  and (!fuel_danger.offset(by: −1).defaults(to: false) or fuel_danger)
  and power_p_consumed >= power_p_consumed.offset(by: −1).defaults(to:
       power_p_consumed)
```

Specification: *mm_output*_1

```
import math
input stateID_SC: UInt64
// Assumptions
assume<a1> trace_pos >= 0
// Exemplary state transition analysis
output trace_pos @ stateID_SC := trace_pos.offset(by: −1).defaults(to: −1) + 1
output change_state := if trace_pos = 0 then false
  else stateID_SC != stateID_SC.offset(by: −1).defaults(to: 0)
output transitions := if stateID_SC.offset(by:−1).defaults(to: 0) = 0 then stateID_SC == 1
  else if stateID_SC.offset(by: −1).defaults(to: 0) == 1 then stateID_SC == 1 or stateID_SC
       == 2
  else if stateID_SC.offset(by: −1).defaults(to: 0) == 2 then stateID_SC == 1 or stateID_SC
       == 3
  else if stateID_SC.offset(by: −1).defaults(to: 0) == 3 then stateID_SC == 3
  else false
output invalid_transitions := change_state and !transitions
trigger invalid_transitions "Invalid state transition"
// Assertions
assert <a1> invalid_transitions or
!( stateID_SC.offset(by: −1).defaults(to: 0) != 0 and stateID_SC = 0 )
assert <a2> (stateID_SC == 1 or stateID_SC == 2 or stateID_SC == 3 )
  or !( stateID_SC.offset(by: −2).defaults(to: 0) = 1
   and transitions.offset(by: −1).defaults(to: false) and transitions )
```

Specification: *mm_output*_2

```
import math
input time_s: UInt64
input time_us: UInt64
input stateID_SC: Int64
input OnGround: UInt64
// Assumptions
assume <a1> trace_pos >= 0
  and time − time.offset(by: −1).defaults(to: time − 0.1) <= 0.1
  and time − time.offset(by: −1).defaults(to: time − 0.1) > 0.0
assume <a2> (time_s = time_s.offset(by: −1).defaults(to: 0)
  and time_us > time_us.offset(by: −1).defaults(to: 0))
  and (time_s > time_s.offset(by: −1).defaults(to: 0)
       or time_us > time_us.offset(by: −1).defaults(to: 0))
// Frequency computations
output time := cast(time_s) + cast(time_us) / 1000000.0
output start_time := if time.offset(by: −1).defaults(to: −1.0) == −1.0 then time else
    start_time.offset(by: −1).defaults(to: −1.0)
output flight_time := time − start_time
output trace_pos @ time_s or time_us or stateID_SC or OnGround := trace_pos.offset(by:
    −1).defaults(to: −1) + 1
output frequency := 1.0 / ( time − time.offset(by: −1).defaults(to: time − 0.0001) )
output freq_sum := freq_sum.offset(by: −1).defaults(to: 0.0) + frequency
output freq_avg := freq_sum / cast(trace_pos+1)
// Phase Statistics
output change_state := if trace_pos = 0 then false
       else stateID_SC != stateID_SC.offset(by: −1).defaults(to: 0)
trigger change_state
output entrance_time := if change_state then time
         else entrance_time.offset(by: −1).defaults(to: time)
output hover_end := change_state and stateID_SC.offset(by: −1).defaults(to: −1) = 4
output hover_cur_time := if hover_end then
time − entrance_time.offset(by: −1).defaults(to: 0.0)else 0.0
output hover_sum_time := hover_sum_time.offset(by: −1).defaults(to: 0.0) +
    hover_cur_time
output hover_num_times := hover_num_times.offset(by: −1).defaults(to: 0) + if hover_end
    then 1 else 0
output hover_max_time := max ( hover_max_time.offset(by: −1).defaults(to: 0.0),
    hover_cur_time )
output hover_avg_time := if hover_num_times != 0 then hover_sum_time /
    cast(hover_num_times) else 0.0
output landing_info := if change_state and stateID_SC = 5 then 0.0 else time −
    entrance_time.offset(by: −1).defaults(to: time)
output landing_error := stateID_SC = 5 and OnGround != 1 and landing_info > 20.0
// Assertions
assert <a1> time.offset(by: −1).defaults(to: −1.0) < time
  and start_time == start_time.offset(by: −1).defaults(to: start_time)
  and flight_time >= flight_time.offset(by: −1).defaults(to: 0.0)
assert <a2> time >= entrance_time and start_time <= entrance_time
  and hover_cur_time >= 0.0 and hover_max_time <= flight_time
assert <a3> !(landing_error and hover_end)
```

```
  and (!landing_error or landing_info > 0.0)
```

Specification: *contingency_output*

```
input avgDist_laser: Float64
input actual_laser: Float64
input static_laser: Float64
input avgDist_optical:Float64
input actual_optical: Float64
input static_optical: Float64
// Assumptions
assume <a1> avgDist_laser >= 0.0 and actual_laser >= 0.0
  and static_laser >= 0.0 and avgDist_optical >= 0.0
  and actual_optical >= 0.0 and static_optical >= 0.0
  and (avgDist_laser + actual_laser + static_laser > 0.0)
  and (avgDist_optical + actual_optical + static_optical > 0.0)
// Trust computations
output rating_laser := 0.2 * static_laser + 0.4 * actual_laser
     + 0.4 * avgDist_laser
output rating_optical := 0.2 * static_optical + 0.4 * actual_optical + 0.4 * avgDist_optical
output trust_laser := rating_laser / ( rating_laser + rating_optical)
output trust_optical := 1.0 − trust_laser
trigger trust_laser >= 0.5 "Trust in laser"
trigger trust_optical > 0.5 "Trust in optical sensor"
// Assertions
assert <a1> trust_laser ≠ trust_optical
```

Specification: *health_output*

```
import math
// average distance to the measured ostacle (range of sight) using laser
input avgDist_laser: Float64
// average distance to the measured ostacle (range of sight) using camera
input avgDist_optical: Float64
input vel: Float64
// Assumption
assume <a1> avgDist_laser <= 100.0 and avgDist_laser >= 0.0 // both in m
  and avgDist_optical <= 50.0 and avgDist_optical >= 0.0 // both in m
  and abs(vel) < 5.5 // in m/s
// Line of sight
output avgDst_dif := min( avgDist_laser, avgDist_optical ) − abs(vel)
trigger avgDst_dif < 5.0 "WARNING: Dynamic Velocity Limit reached"
trigger avgDst_dif < 2.0 "ERROR: Abort mission."
// Assertions
assert <a1> avgDst_dif < 54.5 and avgDst_dif > −5.5
```

A.2. Trace Segmentation

A.2.1. Proof of Theorem 1

We prove the Theorem 1 by structural induction on φ. We show that "For any trace σ and STL formula φ, we have $\sigma \models \varphi$ iff $\rho(\varphi, \sigma) \geq 0$". →Thm. 1, p. 37

Proof.

Base Cases:

Case $\varphi : p_i$ where $p_i \in AP$ and $|\sigma| > 0$,

$$\begin{aligned} & \sigma \models \varphi \\ \overset{\text{Def. 3.2}}{\iff} \; & f_i(\sigma(1)) \geq 0 \\ \overset{\text{Def. 3.3}}{\iff} \; & \rho(p_i, \sigma) \geq 0 \end{aligned}$$

Case $\varphi : p_i$ where $p_i \in AP$ and $|\sigma| = 0$,

$$\begin{aligned} & \sigma \not\models \varphi \\ \overset{\text{Def. 3.2}}{\iff} \; & f_i(\sigma(1)) < 0 \\ \overset{\text{Def. 3.3}}{\iff} \; & \rho(p_i, \sigma) < 0 \end{aligned}$$

Induction Step:

Case $\varphi : \neg\varphi_1$,

$$\begin{aligned} & \sigma \models \neg\varphi_1 \\ \overset{\text{Def. 3.2}}{\iff} \; & \sigma \not\models \varphi_1 \\ \overset{\text{IH}}{\iff} \; & \rho(\varphi_1, \sigma) < 0 \\ \overset{\text{Def. 3.3}}{\iff} \; & \rho(\neg\varphi_1, \sigma) > 0 \end{aligned}$$

Case $\varphi : \varphi_1 \wedge \varphi_2$,

$$\begin{aligned} & \sigma \models \varphi_1 \wedge \varphi_2 \\ \overset{\text{Def. 3.2}}{\iff} \; & \sigma \models \varphi_1 \wedge \sigma \models \varphi_2 \\ \overset{\text{IH}}{\iff} \; & \rho(\varphi_1, \sigma) \geq 0 \wedge \rho(\varphi_2, \sigma) \geq 0 \\ \iff \; & \min(\rho(\varphi_1, \sigma), \rho(\varphi_2, \sigma)) \geq 0 \\ \overset{\text{Def. 3.3}}{\iff} \; & \rho(\varphi_1 \wedge \varphi_2, \sigma) \geq 0 \end{aligned}$$

Case $\varphi : \varphi_1 \vee \varphi_2$,

$$\begin{aligned}
& \sigma \models \varphi_1 \vee \varphi_2 \\
\overset{\text{Def. 3.2}}{\iff}\; & \sigma \models \varphi_1 \;\vee\; \sigma \models \varphi_2 \\
\overset{\text{IH}}{\iff}\; & \rho(\varphi_1, \sigma) \geq 0 \;\vee\; \rho(\varphi_2, \sigma) \geq 0 \\
\iff\; & \max(\rho(\varphi_1, \sigma), \rho(\varphi_2, \sigma)) \geq 0 \\
\overset{\text{Def. 3.3}}{\iff}\; & \rho(\varphi_1 \vee \varphi_2, \sigma) \geq 0
\end{aligned}$$

Case $\varphi : \Diamond_{[l,u]}(\varphi_1)$,

$$\begin{aligned}
& \sigma \models \Diamond_{[l,u]}(\varphi_1) \\
\overset{\text{Def. 3.2}}{\iff}\; & \exists i \in [l,u],\; \sigma[i:] \models \varphi_1 \\
\overset{\text{IH}}{\iff}\; & \exists i \in [l,u],\; \rho(\varphi_1, \sigma[i:]) \geq 0 \\
\iff\; & \max_{i \in [l,u]} (\rho(\varphi_1, \sigma[i:])) \geq 0 \\
\overset{\text{Def. 3.3}}{\iff}\; & \rho(\Diamond_{[l,u]}(\varphi_1), \sigma) \geq 0
\end{aligned}$$

Case $\varphi : \Box_{[l,u]}(\varphi_1)$,

$$\begin{aligned}
& \sigma \models \Box_{[l,u]}(\varphi_1) \\
\overset{\text{Def. 3.2}}{\iff}\; & \forall i \in [l,u],\; \sigma[i:] \models \varphi_1 \\
\overset{\text{IH}}{\iff}\; & \forall i \in [l,u],\; \rho(\varphi_1, \sigma[i:]) \geq 0 \\
\iff\; & \min_{i \in [l,u]} (\rho(\varphi_1, \sigma[i:])) \geq 0 \\
\overset{\text{Def. 3.3}}{\iff}\; & \rho(\Box_{[l,u]}(\varphi_1), \sigma) \geq 0
\end{aligned}$$

Case $\varphi : \varphi_1 \,\mathcal{U}_{[l,u]}\, \varphi_2$,

$$\begin{aligned}
& \sigma \models \varphi_1 \,\mathcal{U}_{[l,u]}\, \varphi_2 \\
\overset{\text{Def. 3.2}}{\iff}\; & \exists i \in [l,u], (\forall j \in [0, i-1], \sigma[j:] \models \varphi_1) \wedge \sigma[i:] \models \varphi_2 \\
\overset{\text{IH}}{\iff}\; & \exists i \in [l,u], (\forall j \in [0, i-1], \rho(\varphi_1, \sigma[j:]) \geq 0) \\
& \qquad\qquad\qquad\qquad \wedge \rho(\varphi_2, \sigma[i:]) \geq 0 \\
\iff\; & \max_{i \in [l,u]} \min(\rho(\varphi_2, \sigma[i:]), \min_{j \in [0, i-1]} \rho(\varphi_1, \sigma[j:])) \geq 0 \\
\overset{\text{Def. 3.3}}{\iff}\; & \rho(\varphi_1 \,\mathcal{U}_{[l,u]}\, \varphi_2, \sigma) \geq 0
\end{aligned}$$

$\square$

A.2.2. Proof of Theorem 2

We prove the theorem by structural induction on $\mathcal{T}$. We show that "For any trace σ and TBT $\mathcal{T}$, $\sigma \models \mathcal{T}$ iff $\rho(\mathcal{T}, \sigma) \geq 0$".

Proof.

Base Case $\mathcal{T}$: $\mathsf{Leaf}(\varphi)$

$$\begin{aligned}
& \sigma \models \mathsf{Leaf}(\varphi) \\
\overset{\text{Def. 3.9}}{\iff} \quad & \sigma \models \varphi \\
\overset{\text{Thm. 1}}{\iff} \quad & \rho(\varphi, \sigma) \geq 0
\end{aligned}$$

Induction Step:

Case $\mathcal{T}$: $\mathsf{Fback}([\mathcal{T}_1, \dots, \mathcal{T}_k])$,

$$\begin{aligned}
& \sigma \models \mathsf{Fback}([\mathcal{T}_1, \dots, \mathcal{T}_k]) \\
\overset{\text{Def. 3.9}}{\iff} \quad & \exists j \in \{1, \dots, k\}, \exists i \in [0, |\sigma| - 1], \sigma[i:] \models \mathcal{T}_j \\
\overset{\text{IH}}{\iff} \quad & \exists j \in \{1, \dots, k\}, \exists i \in [0, |\sigma| - 1], \rho(\mathcal{T}_j, \sigma[i:]) \geq 0 \\
\iff \quad & (\max_{j \in [1,k]} \max_{i \in [0, |\sigma|-1]} \rho(\mathcal{T}_j, \sigma[i:])) \geq 0 \\
\overset{\text{Def. 3.10}}{\iff} \quad & \rho(\mathsf{Fback}([T_1, \dots, T_k]), \sigma) \geq 0
\end{aligned}$$

Case $\mathcal{T}$: $\mathsf{Par}_M([\mathcal{T}_1, \dots, \mathcal{T}_k])$,

$$\begin{aligned}
& \sigma \models \mathsf{Par}_M([\mathcal{T}_1, \dots, \mathcal{T}_k]) \\
\overset{\text{Def. 3.9}}{\iff} \quad & \exists i_1, \dots, i_M \in \{1, \dots, k\}, \sigma \models \mathcal{T}_{i_1}, \dots, \sigma \models \mathcal{T}_{i_M} \\
\overset{\text{IH}}{\iff} \quad & \exists i_1, \dots, i_M \in \{1, \dots, k\}, \rho(\mathcal{T}_{i_1}, \sigma) \geq 0, \dots, \rho(\mathcal{T}_{i_M}, \sigma) \geq 0 \\
\iff \quad & max_M(\rho(\mathcal{T}_1, \sigma), \dots, \rho(\mathcal{T}_k, \sigma)) \geq 0 \\
\overset{\text{Def. 3.10}}{\iff} \quad & \rho(\mathsf{Par}_M([\mathcal{T}_1, \dots, \mathcal{T}_k]), \sigma) \geq 0
\end{aligned}$$

Case $\mathcal{T}$: $\mathsf{Seq}([\mathcal{T}_1, \mathcal{T}_2])$,

$$\begin{aligned}
& \sigma \models \mathsf{Seq}([\mathcal{T}_1, \mathcal{T}_2]) \\
\overset{\text{Def. 3.9}}{\iff} \quad & \exists i \in [0, |\sigma| - 1], \sigma[:i] \models \mathcal{T}_1 \wedge \sigma[i+1:] \models \mathcal{T}_2 \\
\overset{\text{IH}}{\iff} \quad & \exists i \in [0, |\sigma| - 1], \rho(\mathcal{T}_1, \sigma[:i]) \geq 0 \wedge \rho(\mathcal{T}_2, \sigma[i+1:]) \geq 0 \\
\iff \quad & \max_{i \in [0, |\sigma|-1]} \min(\rho(\mathcal{T}_1, \sigma[:i]), \rho(\mathcal{T}_2, \sigma[i+1:])) \geq 0 \\
\overset{\text{Def. 3.10}}{\iff} \quad & \rho(\mathsf{Seq}([\mathcal{T}_1, \mathcal{T}_2]), \sigma) \geq 0
\end{aligned}$$

Case $\mathcal{T}$: $\mathsf{Tout}_t(\mathcal{T}_1)$

$$\begin{aligned}
&\sigma \models \mathsf{Tout}_t(\mathcal{T}_1)\\
\overset{\text{Def. 3.9}}{\iff}\ &|\sigma| \le t \ \wedge\ \exists t'.\, 0 \le t' < |\sigma| \ \wedge\ \sigma[:t'] \models \mathcal{T}\\
\overset{\text{IH}}{\iff}\ &|\sigma| \le t \ \wedge\ \exists t'.\, 0 \le t' < |\sigma| \ \wedge\ \rho(\mathcal{T}, \sigma[:t']) \ge 0\\
\iff\ &|\sigma| \le t \ \wedge \max_{t' \in [0,|\sigma|-1]} \rho(\mathcal{T}, \sigma[:t']) \ge 0\\
\overset{\text{Def. 3.10}}{\iff}\ &\rho(\mathsf{Tout}_t(\mathcal{T}_1), \sigma) \ge 0
\end{aligned}$$

Case $\mathcal{T}$: $\bigstar_n(\mathcal{T}_1)$,

$$\begin{aligned}
&\sigma \models \bigstar_n(\mathcal{T}_1)\\
\overset{\text{Def. 3.9}}{\iff}\ &\exists i \in [0, |\sigma|-1].\sigma[:i] \models \mathcal{T}_1 \wedge \sigma[i+1:] \models \bigstar_{n-1}(\mathcal{T}_1)\\
\overset{\text{IH}}{\iff}\ &\exists i \in [0, |\sigma|-1].\rho(\mathcal{T}_1, \sigma[:i]) \ge 0 \wedge\\
&\qquad\qquad \rho(\bigstar_{n-1}(\mathcal{T}_1), \sigma[i+1:]) \ge 0\\
\iff\ &\max_{i \in [0,|\sigma|-1]} \min(\rho(\mathcal{T}_1, \sigma[:i]),\\
&\qquad\qquad \rho(\bigstar_{n-1}(\mathcal{T}_1), \sigma[i+1:])) \ge 0\\
\overset{\text{Def. 3.10}}{\iff}\ &\rho(\mathsf{Seq}([\mathcal{T}_1, \bigstar_{n-1}(\mathcal{T}_1)]), \sigma) \ge 0\\
\overset{\text{Def. 3.10}}{\iff}\ &\rho(\bigstar_n(\mathcal{T}_1), \sigma) \ge 0
\end{aligned}$$

$\square$

A.2.3. Proof of Theorem 8

Let σ be a δ-stuttering trace for a given $\delta \ge 1$. Let $\mathcal{T}$ be a δ preserving TBT. Recall that (a) every occurrence of $\mathcal{U}_{[l,u]}$ has $l = 0$ and u divisible by δ; (b) every occurrence of $\diamondsuit_{[l,u]}$ and $\square_{[l,u]}$ has l, u divisible by δ and (c) every occurrence of Tout_t has t divisible by δ.

Let $\mathcal{T}'$ be the TBT with each occurrence of $\bowtie_{[l,u]}$ replaced with $\bowtie_{[l/\delta, u/\delta]}$ for $\bowtie \in \{\mathcal{U}, \diamondsuit, \square\}$ and every occurrence of Tout_t replaced by $\mathsf{Tout}_{t/\delta}$.

Let σ' be the subsampled trace where $\sigma'(i+1) = \sigma(i\delta + 1)$ for $i \in \{0, \dots, \frac{|\sigma|}{\delta} - 1\}$.

We say that a set of consecutive trace positions in σ forms a block B_i : $\{i\delta + 1, \dots, (i+1)\delta\}$. Thus, we can partition the indices of trace σ into $m = \frac{|\sigma|}{\delta}$ contiguous blocks $B_0, \dots, B_{m-1}$.

First, we will prove for every TBT $\mathcal{T}$ that is δ-preserving and of the form $\mathsf{Leaf}(\varphi)$ that if φ holds at the first position of the block then it holds everywhere in the block (and vice-versa).

Lemma 19. *For a δ preserving TBT $\mathcal{T}$ of the form* Leaf(φ)*, a δ-stuttering trace σ and index $i \geq 0$ such that $i\delta + 1 \leq |\sigma|$,*

$$\sigma[i\delta :] \models \varphi \textit{ iff } \sigma[i\delta + k :] \models \varphi\,,$$

for all $k \in \{0, \dots, \delta - 1\}$.

Note that the converse direction holds trivially since we can set $k = 0$.

Proof. Proof is by induction on the structure of the formula φ. Assume $0 \leq i < m$ and $0 \leq k \leq \delta - 1$.
Case φ is an atomic proposition p: True by definition of a δ-stuttering trace (Definition 4.4). → Def. 4.4, p. 60
Case $\varphi : \varphi_1 \circ \varphi_2$ where $\circ \in \{\wedge, \vee\}$. Since we assume the result by induction on the subformulas φ_1, φ_2, the proof follows directly from that.
Case $\varphi : \neg\varphi_1$. By induction, we have

$$\sigma[i\delta :] \models \varphi_1 \text{ iff } \sigma[i\delta + k :] \models \varphi_1\,,$$

Therefore,

$$\sigma[i\delta :] \not\models \varphi_1 \text{ iff } \sigma[i\delta + k :] \not\models \varphi_1\,,$$

It follows that

$$\sigma[i\delta :] \models \neg\varphi_1 \text{ iff } \sigma[i\delta + k :] \models \neg\varphi_1\,,$$

Case $\varphi : \Diamond_{[l,u]}\varphi_1$.

$$\sigma[i\delta :] \models \Diamond_{[l,u]}\varphi_1 \text{ iff } \exists j \in [i\delta + l, i\delta + u]\ \sigma[j :] \models \varphi_1$$

Assume $\sigma[i\delta :] \models \Diamond_{[l,u]}\varphi_1$. Let $j + 1 \in B_r$ for some block $B_r : \{r\delta + 1, \dots, (r + 1)\delta\}$. The reason we consider $j + 1$ is that $\sigma[j :]$ by convention begins at state $\sigma(j + 1)$. Therefore, $j = r\delta + \hat{r}$ for $\hat{r} \in \{0, \dots, \delta - 1\}$

By induction hypothesis: $\sigma[r\delta + k :] \models \varphi_1$ for $k \in \{0, \dots, \delta - 1\}$. and $j - i\delta \in [l, u]$. Therefore, $\hat{r} + r\delta - i\delta \in [l, u]$. However, since l and u are divisible by δ and $\hat{r} < \delta$, we obtain

$$\frac{\hat{r}}{\delta} + r - i \in [\frac{l}{\delta}, \frac{u}{\delta}] \text{ or, equivalently, } r - i \in [\frac{l}{\delta}, \frac{u}{\delta}]\,.$$

Therefore, $r\delta \in i\delta + [l, u]$. $\sigma[i\delta + k :] \models \Diamond_{[l,u]}\varphi_1$ for all k since $\sigma[r\delta + k :] \models \varphi_1$ and $(r\delta + k) - (i\delta + k) \in [l, u]$.

Conversely, if $\sigma[i\delta + k :] \models \Diamond_{[l,u]}\varphi_1$ for all k, then so does $\sigma[i\delta :]$.
Case $\varphi : \Box_{[l,u]}\varphi_1$. Proof is similar to the $\Diamond_{[l,u]}$ case.

Let $\sigma[i\delta :] \models \Box_{[l,u]}\varphi$. Therefore, for all $j \in i\delta + [l,u]$ we have

$$\sigma[j :] \models \varphi_1 .$$

Consider block $B_i : \{i\delta + 1, \ldots, (i+1)\delta\}$. Let $l = l'\delta$ and $u = u'\delta$. Then, we note by induction that all indices in the blocks:

$$B_{i+l'}, \ldots, B_{i+u'}$$

satisfy φ_1.

Therefore, for all $k \in \{0, \ldots, \delta - 1\}$ for all $\hat{j} \in \{(i+l')\delta + 1 + k, \ldots, (i+u')\delta + 1 + k\}$, we have

$$\sigma[\hat{j} :] \models \varphi_1 .$$

Therefore, $\sigma[i\delta + k :] \models \Box_{[l,u]}\varphi_1$.

The converse holds trivially.

Case $\varphi : \ \varphi_1 \mathcal{U}_{[0,u]} \varphi_2$.

If $u = 0$ then φ is logically equivalent to φ_2 and the result immediately holds. Assume $u > 0$.

Let $\sigma[i\delta :] \models \varphi$. There exists $j \in [i\delta, i\delta + u]$ such that

$$\sigma[j :] \models \varphi_2$$

and for all $j' \in [i\delta, j-1]$

$$\sigma[j' :] \models \varphi_1$$

Once again let $B_r : \{r\delta + 1, \ldots, (r+1)\delta\}$ be the block containing $j+1$. Once again by induction, we have all indices in B_r satisfy φ_2. Also, all indices in the range $j' \in [i\delta, r\delta)$ satisfy $\sigma[j' :] \models \varphi_1$.

If B_r is the same block as B_i then we note that

$$\sigma[i\delta + k :] \models \varphi_2 \text{ and thus } \sigma[i\delta + k :] \models \varphi_1 \mathcal{U}_{[0,u]} \varphi_2 .$$

Otherwise, we have $r > i$. Since $j - i\delta \leq u$ we have $j = r\delta + \hat{r}$ and $r\delta - i\delta + \hat{r} \leq u'\delta$. Dividing by δ, we get $r - i + \frac{\hat{r}}{\delta} \leq u'$ or $r - i \leq u'$. Therefore, $(r\delta - i\delta) \leq u$. Therefore, for all $k \in \{0, \ldots, \delta - 1\}$, we have

$$r\delta - (i\delta + k) \leq u - k \leq u .$$

Therefore, we conclude that

$$\sigma[i\delta + k :] \models \varphi_1 \mathcal{U}_{[l,u]} \varphi_2$$

since $\sigma[r\delta] \models \varphi_2$, $\sigma[j' :] \models \varphi_1$ for all $j' \in [i\delta, r\delta)$ and $r\delta \in i\delta + k + [0,u]$.

□

To establish the main Theorem 8, we will need to establish the following lemma. Let φ be an STL formula that is δ preserving and φ' be obtained by transforming every interval $[l, u]$ associated with $\Diamond, \Box, \mathcal{U}$ into $[\frac{l}{\delta}, \frac{u}{\delta}]$. →Thm. 8, p. 61

Lemma 20. *For all i such that $i\delta < |\sigma|$, if $\sigma'[i:] \models \varphi'$ then $\sigma[i\delta:] \models \varphi$.*

Proof. Proof is once again by induction on the structure of the formula φ.
Case φ is an atomic proposition p: we have $\varphi' = \varphi$. The rest holds by definition of subsampling.
Case $\varphi \in \{\neg\varphi_1, \varphi_1 \vee \varphi_2, \varphi_1 \wedge \varphi_2\}$: holds by induction on the subformulas.
Case $\varphi : \Diamond_{[l\delta, u\delta]}\varphi_1$: We have $\varphi' : \Diamond_{[l,u]}\varphi'_1$. Therefore, $\sigma'[i:] \models \varphi'$ iff $\exists j \in [i+l, i+u]\ \sigma'[j:] \models \varphi'_1$. By Ind. Hyp., $\sigma[j\delta:] \models \varphi_1$. Also, $j\delta \in i\delta + [l\delta, u\delta]$. Therefore, $\sigma[i\delta:] \models \Diamond_{[i\delta, j\delta]}\varphi_1$.
Case $\varphi : \Box_{[l\delta, u\delta]}\varphi_1$: We have $\varphi' : \Box_{[l,u]}\varphi'_1$. Therefore, $\sigma'[i:] \models \varphi'$ iff $\forall j \in [i+l, i+u]\ \sigma'[j:] \models \varphi'_1$. By Ind. Hyp., $\sigma[j\delta:] \models \varphi_1$ for all $j \in i + [l, u]$. Using Lemma 19, we conclude that $\sigma[j\delta + k:] \models \varphi_1$ for all $j \in i + [l, u]$ and for all $k \in [0, \delta - 1]$. Therefore, we conclude that $\sigma[i\delta:] \models \Box_{[l\delta, u\delta]}\varphi_1$.
Case $\varphi : \varphi_1 \mathcal{U}_{[0,u\delta]}\varphi_2$: We have $\varphi' : \varphi'_1 \mathcal{U}_{[0,u]}\varphi'_2$. Therefore, $\sigma'[i:] \models \varphi'$ iff $\exists j \in [i, i+u]\ \sigma'[j:] \models \varphi'_2$ and $\sigma'[j':] \models \varphi'_1$ for $j' \in [i, j)$. By Ind. Hyp., $\sigma[j\delta:] \models \varphi_2$ and $\sigma[j'\delta:] \models \varphi_1$. Lemma 19, we conclude that $\sigma[j'\delta + k:] \models \varphi_1$ for all $j' \in i + [0, j)$ and for all $k \in [0, \delta - 1]$. Therefore, we conclude that $\sigma[i\delta:] \models \varphi_1 \mathcal{U}_{[l\delta, u\delta]}\varphi_2$.

□

Proof of Theorem 8: Now we will prove a stronger version from which the required result follows directly.

Lemma 21. *If $\sigma'[i:j] \models \mathcal{T}'$ then $\sigma[i\delta : j\delta + \delta - 1] \models \mathcal{T}$*

Proof. Let $\sigma'[i:j] \models \mathcal{T}'$. If $\sigma'[i:j]$ is empty then so is $\sigma[i\delta : j\delta + \delta - 1]$ and the statement holds trivially.

Proof is on the structure of the $\mathcal{T}$.
Case $\mathcal{T} = \mathsf{Leaf}(\varphi)$: Follows directly from Lemma 20 applied to the sub-trace $\sigma[i\delta : j\delta + \delta - 1]$ which is also a delta-stuttering trace.
Case $\mathcal{T} : \mathsf{Seq}([\mathcal{T}_1, \mathcal{T}_2])$.

$$\sigma'[i:] \models \mathsf{Seq}([\mathcal{T}'_1, \mathcal{T}'_2]) \Rightarrow \quad \exists\, u \geq i,\ \sigma[i:u] \models \mathcal{T}'_1,\ \sigma[u+1:] \models \mathcal{T}'_2$$

Note that $\sigma[i\delta : u\delta + \delta - 1]$ and $\sigma[(u+1)\delta:]$ are δ stuttering traces and by induction hypothesis, we have

$$\sigma[i\delta : u\delta + \delta - 1] \models \mathcal{T}_1 \text{ and } \sigma[(u+1)\delta:] \models \mathcal{T}_2\,.$$

Therefore, $\sigma \models \mathsf{Seq}(\mathcal{T}_1, \mathcal{T}_2)$.
Case $\mathcal{T} : \mathsf{Fback}([\mathcal{T}_1, \ldots, \mathcal{T}_k])$.

$\sigma'[i:j] \models \mathsf{Fback}([\mathcal{T}_1', \ldots, \mathcal{T}_k'])$. It follows that there exists $r \geq i$ and $l \in [1,k]$ such that

$$\sigma'[r:j] \models \mathcal{T}_l' .$$

By induction, $\sigma[r\delta : j\delta + \delta - 1] \models \mathcal{T}_l$. Therefore, $\sigma[i\delta : j\delta + \delta - 1] \models \mathsf{Fback}([\mathcal{T}_1, \ldots, \mathcal{T}_k])$.
Case $\mathcal{T} : \mathsf{Par}_M([\mathcal{T}_1, \ldots, \mathcal{T}_k])$.
We have $\sigma'[i:j] \models \mathsf{Par}_M([\mathcal{T}_1', \ldots, \mathcal{T}_k'])$ iff there are M subtrees $\mathcal{T}_{i_1}', \ldots, \mathcal{T}_{i_M}'$ such that

$$\sigma'[i:j] \models \mathcal{T}_{i_j}' .$$

By induction $\sigma[i\delta : j\delta + \delta - 1] \models \mathcal{T}_{i_j}$ and thus

$$\sigma[i\delta : j\delta + \delta - 1] \models \mathsf{Par}_M([\mathcal{T}_1, \ldots, \mathcal{T}_k]) .$$

Case $\mathcal{T} : \mathsf{Tout}_t(\mathcal{T}_1)$.
We have $\sigma'[i:j] \models \mathsf{Tout}_t \mathcal{T}_1'$. Assume that $j - i + 1 \geq t$.
$\sigma'[i : i + t - 1] \models \mathcal{T}_1'$. By induction, $\sigma[i\delta : i\delta + (t-1)\delta + \delta - 1] \models \mathcal{T}_1$. Therefore, $\sigma[i\delta : t\delta - 1] \models \mathsf{Tout}_{t\delta}(\mathcal{T}_1)$.
Case $\mathcal{T} : \bigstar_n(\mathcal{T}_1)$.
Let $\sigma'[i:j] \models \bigstar_n(\mathcal{T}_1)$. There exists indices $-1 = i_1, \ldots, i_k = j$ for $k \leq n+1$ such that $\sigma'[i_1 + 1 : i_2] \models \mathcal{T}_1'$, $\sigma[i_2 + 1 : i_3] \models \mathcal{T}_1'$, $\cdots$, $\sigma[i_{k-1} + 1 : i_k] \models \mathcal{T}_1'$. Therefore, for each l, $\sigma[(i - l + 1)\delta : i_{l+1}\delta + \delta - 1] \models \mathcal{T}_1$. Therefore, $\sigma[i\delta : j\delta + \delta - 1] \models \bigstar_n(\mathcal{T}_1)$.

□

→Thm. 8, p. 61 Theorem 8 follows directly.

Bibliography

[1] Evan Ackerman and Michael Koziol. “The blood is here: Zipline’s medical delivery drones are changing the game in Rwanda”. In: *IEEE spectrum* 56.5 (2019), pp. 24–31.

[2] David Harel and Amir Pnueli. “On the Development of Reactive Systems”. In: *Logics and Models of Concurrent Systems - Conference proceedings, Colle-sur-Loup (near Nice), France, 8-19 October 1984*. Ed. by Krzysztof R. Apt. Vol. 13. NATO ASI Series. Springer, 1984, pp. 477–498. DOI: 10.1007/978-3-642-82453-1_17. URL: https://doi.org/10.1007/978-3-642-82453-1%5C_17.

[3] Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. MIT press, 2016.

[4] Rajeev Alur. *Principles of cyber-physical systems*. MIT press, 2015.

[5] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136.

[6] Mark J Willis. “Proportional-integral-derivative control”. In: *Dept. of Chemical and Process Engineering University of Newcastle* 6 (1999), p. 28.

[7] Eduardo D Sontag. *Mathematical control theory: deterministic finite dimensional systems*. Vol. 6. Springer Science & Business Media, 2013.

[8] Gino Brunner, Bence Szebedy, Simon Tanner, and Roger Wattenhofer. “The Urban Last Mile Problem: Autonomous Drone Delivery to Your Balcony”. In: *CoRR* abs/1809.08022 (2018). arXiv: 1809.08022. URL: http://arxiv.org/abs/1809.08022.

[9] Niall O’ Mahony, Sean Campbell, Anderson Carvalho, Suman Harapanahalli, Gustavo Adolfo Velasco-Hernández, Lenka Krpalkova, Daniel Riordan, and Joseph Walsh. “Deep Learning vs. Traditional Computer Vision”. In: *CoRR*

abs/1910.13796 (2019). arXiv: 1910.13796. URL: http://arxiv.org/abs/1910.13796.

[10] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. "Mastering the game of Go with deep neural networks and tree search". In: *Nat.* 529.7587 (2016), pp. 484–489. DOI: 10.1038/NATURE16961. URL: https://doi.org/10.1038/nature16961.

[11] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. "Mastering the game of Go without human knowledge". In: *Nat.* 550.7676 (2017), pp. 354–359. DOI: 10.1038/NATURE24270. URL: https://doi.org/10.1038/nature24270.

[12] Elia Kaufmann, Leonard Bauersfeld, Antonio Loquercio, Matthias Müller, Vladlen Koltun, and Davide Scaramuzza. "Champion-level drone racing using deep reinforcement learning". In: *Nat.* 620.7976 (2023), pp. 982–987. DOI: 10.1038/S41586-023-06419-4. URL: https://doi.org/10.1038/s41586-023-06419-4.

[13] Sayan Mitra. *Verifying cyber-physical systems: A path to safe autonomy*. MIT Press, 2021.

[14] Matt Luckcuck, Marie Farrell, Louise A. Dennis, Clare Dixon, and Michael Fisher. "Formal Specification and Verification of Autonomous Robotic Systems: A Survey". In: *ACM Comput. Surv.* 52.5 (Sept. 2019). ISSN: 0360-0300. DOI: 10.1145/3342355. URL: https://doi.org/10.1145/3342355.

[15] Martin Leucker and Christian Schallhart. "A brief account of runtime verification". In: *J. Log. Algebraic Methods Program.* 78.5 (2009), pp. 293–303. DOI: 10.1016/J.JLAP.2008.08.004. URL: https://doi.org/10.1016/j.jlap.2008.08.004.

[16] Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. "A taxonomy for classifying runtime verification tools". In: *Int. J. Softw. Tools Technol. Transf.* 23.2 (2021), pp. 255–284. DOI: 10.1007/S10009-021-00609-Z. URL: https://doi.org/10.1007/s10009-021-00609-z.

[17] Ezio Bartocci, Jyotirmoy V. Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Nickovic, and Sriram Sankaranarayanan. "Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications". In: *Lectures on Runtime Verification - Introductory and Advanced Topics*. Ed. by Ezio Bartocci and Yliès Falcone. Vol. 10457. Lecture Notes in Computer

Science. Springer, 2018, pp. 135–175. DOI: 10.1007/978-3-319-75632-5_5. URL: https://doi.org/10.1007/978-3-319-75632-5%5C_5.

[18] C. Watterson and Donal Heffernan. "Runtime verification and monitoring of embedded systems". In: *IET Softw.* 1.5 (2007), pp. 172–179. DOI: 10.1049/IET-SEN:20060076. URL: https://doi.org/10.1049/iet-sen:20060076.

[19] Thomas Reinbacher, Kristin Yvonne Rozier, and Johann Schumann. "Temporal-Logic Based Runtime Observer Pairs for System Health Management of Real-Time Systems". In: *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. Ed. by Erika Ábrahám and Klaus Havelund. Vol. 8413. Lecture Notes in Computer Science. Springer, 2014, pp. 357–372. DOI: 10.1007/978-3-642-54862-8_24. URL: https://doi.org/10.1007/978-3-642-54862-8%5C_24.

[20] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. "Copilot: A Hard Real-Time Runtime Monitor". In: *Runtime Verification*. Ed. by Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Roşu, Oleg Sokolsky, and Nikolai Tillmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 345–359. ISBN: 978-3-642-16612-9.

[21] Jan Baumeister, Bernd Finkbeiner, Sebastian Schirmer, Maximilian Schwenger, and Christoph Torens. "RTLola Cleared for Take-Off: Monitoring Autonomous Aircraft". In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12225. Lecture Notes in Computer Science. Springer, 2020, pp. 28–39. DOI: 10.1007/978-3-030-53291-8_3. URL: https://doi.org/10.1007/978-3-030-53291-8%5C_3.

[22] Donal Heffernan, Ciaran MacNamee, and Padraig Fogarty. "Runtime verification monitoring for automotive embedded systems using the ISO 26262 functional safety standard as a guide for the definition of the monitored properties". In: *IET Softw.* 8.5 (2014), pp. 193–203. DOI: 10.1049/IET-SEN.2013.0236. URL: https://doi.org/10.1049/iet-sen.2013.0236.

[23] Aaron Kane, Omar Chowdhury, Anupam Datta, and Philip Koopman. "A Case Study on Runtime Monitoring of an Autonomous Research Vehicle (ARV) System". In: *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*. Ed. by Ezio Bartocci and Rupak Majumdar. Vol. 9333. Lecture Notes in Computer Science. Springer, 2015, pp. 102–117. DOI: 10.1007/978-3-319-23820-3_7. URL: https://doi.org/10.1007/978-3-319-23820-3%5C_7.

[24] Sebastian Biewer, Bernd Finkbeiner, Holger Hermanns, Maximilian A. Köhl, Yannik Schnitzer, and Maximilian Schwenger. "On the road with RTLola". In: *Int. J. Softw. Tools Technol. Transf.* 25.2 (2023), pp. 205–218. DOI: `10.1007/S10009-022-00689-5`. URL: `https://doi.org/10.1007/s10009-022-00689-5`.

[25] Xian Li, Ming Chai, Lin Zhao, Tao Tang, and Tianhua Xu. "Safety Monitoring for ETCS with 4-valued LTL". In: *10th International Symposium on Autonomous Decentralized Systems, ISADS 2011, Tokyo & Hiroshima, Japan, March 23-17, 2011*. IEEE Computer Society, 2011, pp. 86–91. DOI: `10.1109/ISADS.2011.18`. URL: `https://doi.org/10.1109/ISADS.2011.18`.

[26] Jeff Huang, Cansu Erdogan, Yi Zhang, Brandon M. Moore, Qingzhou Luo, Aravind Sundaresan, and Grigore Rosu. "ROSRV: Runtime Verification for Robots". In: *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*. Ed. by Borzoo Bonakdarpour and Scott A. Smolka. Vol. 8734. Lecture Notes in Computer Science. Springer, 2014, pp. 247–254. DOI: `10.1007/978-3-319-11164-3\_20`. URL: `https://doi.org/10.1007/978-3-319-11164-3%5C_20`.

[27] Ankush Desai, Tommaso Dreossi, and Sanjit A. Seshia. "Combining Model Checking and Runtime Verification for Safe Robotics". In: *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings*. Ed. by Shuvendu K. Lahiri and Giles Reger. Vol. 10548. Lecture Notes in Computer Science. Springer, 2017, pp. 172–189. DOI: `10.1007/978-3-319-67531-2\_11`. URL: `https://doi.org/10.1007/978-3-319-67531-2%5C_11`.

[28] Chi Hu, Wei Dong, Yonghui Yang, Hao Shi, and Ge Zhou. "Runtime Verification on Hierarchical Properties of ROS-Based Robot Swarms". In: *IEEE Trans. Reliab.* 69.2 (2020), pp. 674–689. DOI: `10.1109/TR.2019.2923681`. URL: `https://doi.org/10.1109/TR.2019.2923681`.

[29] Yu Jiang, Houbing Song, Rui Wang, Ming Gu, Jiaguang Sun, and Lui Sha. "Data-Centered Runtime Verification of Wireless Medical Cyber-Physical System". In: *IEEE Trans. Ind. Informatics* 13.4 (2017), pp. 1900–1909. DOI: `10.1109/TII.2016.2573762`. URL: `https://doi.org/10.1109/TII.2016.2573762`.

[30] Gianluca Torta, Gianluca Aguzzi, Ferruccio Damiani, and Mirko Viroli. "Aggregate Runtime Verification for Emergency Healthcare in Crowded Events". In: *2025 IEEE 5th International Conference on Human-Machine Systems (ICHMS)*. 2025, pp. 315–318. DOI: `10.1109/ICHMS65439.2025.11154242`.

[31] Fraser Cameron, Georgios Fainekos, David M. Maahs, and Sriram Sankaranarayanan. "Towards a Verified Artificial Pancreas: Challenges and Solutions for Runtime Verification". In: *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*. Ed. by Ezio Bartocci and Rupak Majumdar. Vol. 9333. Lecture Notes in Computer Science. Springer,

2015, pp. 3–17. DOI: 10.1007/978-3-319-23820-3_1. URL: https://doi.org/10.1007/978-3-319-23820-3%5C_1.

[32] Maurizio Leotta, Davide Ancona, Luca Franceschini, Dario Olianas, Marina Ribaudo, and Filippo Ricca. "Towards a Runtime Verification Approach for Internet of Things Systems". In: *Current Trends in Web Engineering - ICWE 2018 International Workshops, MATWEP, EnWot, KD-WEB, WEOD, TourismKG, Cáceres, Spain, June 5, 2018, Revised Selected Papers*. Ed. by Cesare Pautasso, Fernando Sánchez-Figueroa, Kari Systä, and Juan Manuel Murillo Rodríguez. Vol. 11153. Lecture Notes in Computer Science. Springer, 2018, pp. 83–96. DOI: 10.1007/978-3-030-03056-8_8. URL: https://doi.org/10.1007/978-3-030-03056-8%5C_8.

[33] Christos Tsigkanos, Marcello M. Bersani, Pantelis A. Frangoudis, and Schahram Dustdar. "Edge-Based Runtime Verification for the Internet of Things". In: *IEEE World Congress on Services, SERVICES 2021, Chicago, IL, USA, September 5-10, 2021*. Ed. by Nimanthi L. Atukorala, Carl K. Chang, Rong N. Chang, Ernesto Damiani, Ian T. Foster, Zhongjie Wang, and Jia Zhang. IEEE, 2021, p. 16. DOI: 10.1109/SERVICES51467.2021.00027. URL: https://doi.org/10.1109/SERVICES51467.2021.00027.

[34] Lei Hang and Dohyeun Kim. "Reliable Task Management Based on a Smart Contract for Runtime Verification of Sensing and Actuating Tasks in IoT Environments". In: *Sensors* 20.4 (2020), p. 1207. DOI: 10.3390/S20041207. URL: https://doi.org/10.3390/s20041207.

[35] Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. "A Stream-Based Specification Language for Network Monitoring". In: *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*. Ed. by Yliès Falcone and César Sánchez. Vol. 10012. Lecture Notes in Computer Science. Springer, 2016, pp. 152–168. DOI: 10.1007/978-3-319-46982-9_10. URL: https://doi.org/10.1007/978-3-319-46982-9%5C_10.

[36] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. "Monitoring hyperproperties". In: *Formal Methods Syst. Des.* 54.3 (2019), pp. 336–363. DOI: 10.1007/S10703-019-00334-Z. URL: https://doi.org/10.1007/s10703-019-00334-z.

[37] Jan Baumeister, Bernd Finkbeiner, Frederik Scheerer, Julian Siber, and Tobias Wagenpfeil. "Stream-Based Monitoring of Algorithmic Fairness". In: *Tools and Algorithms for the Construction and Analysis of Systems - 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part I*. Ed. by Arie Gurfinkel and Marijn Heule. Vol. 15696. Lecture Notes in Computer Science. Springer, 2025, pp. 60–81. DOI: 10.1007/978-3-031-90643-5_4. URL: https://doi.org/10.1007/978-3-031-90643-5%5C_4.

[38] Thomas A. Henzinger, Mahyar Karimi, Konstantin Kueffner, and Kaushik Mallik. "Monitoring Algorithmic Fairness". In: *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*. Ed. by Constantin Enea and Akash Lal. Vol. 13965. Lecture Notes in Computer Science. Springer, 2023, pp. 358–382. DOI: `10.1007/978-3-031-37703-7\_17`. URL: `https://doi.org/10.1007/978-3-031-37703-7%5C_17`.

[39] César Sánchez. "Online and Offline Stream Runtime Verification of Synchronous Systems". In: *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*. Ed. by Christian Colombo and Martin Leucker. Vol. 11237. Lecture Notes in Computer Science. Springer, 2018, pp. 138–163. DOI: `10.1007/978-3-030-03769-7\_9`. URL: `https://doi.org/10.1007/978-3-030-03769-7%5C_9`.

[40] Uma D. Ferrell and Alfred H. Andy Anderegg. "Holistic Assurance Case for System-of-Systems". In: *2022 IEEE/AIAA 41st Digital Avionics Systems Conference (DASC)*. 2022, pp. 1–9. DOI: `10.1109/DASC55683.2022.9925789`.

[41] D. Seto, B. Krogh, L. Sha, and A. Chutinan. "The Simplex architecture for safe online control system upgrades". In: *Proceedings of the 1998 American Control Conference. ACC (IEEE Cat. No.98CH36207)*. Vol. 6. 1998, 3504–3508 vol.6. DOI: `10.1109/ACC.1998.703255`.

[42] Pranav Nagarajan, Suresh K. Kannan, Christoph Torens, Mike E. Vukas, and George F. Wilber. "ASTM F3269 - An Industry Standard on Run Time Assurance for Aircraft Systems". In: *AIAA Scitech 2021 Forum*. DOI: `10.2514/6.2021-0525`. eprint: `https://arc.aiaa.org/doi/pdf/10.2514/6.2021-0525`. URL: `https://arc.aiaa.org/doi/abs/10.2514/6.2021-0525`.

[43] Kent L. Beck. *Test-driven Development - by example*. The Addison-Wesley signature series. Addison-Wesley, 2003. ISBN: 978-0-321-14653-3.

[44] John Ferguson Smart and Jan Molak. *BDD in Action: Behavior-driven development for the whole software lifecycle*. Simon and Schuster, 2023.

[45] Sebastian Schirmer, Philipp Schitz, Johann C. Dauer, Bernd Finkbeiner, and Sriram Sankaranarayanan. *Trace Repair for Temporal Behavior Trees*. 2025. arXiv: `2509.08610 [cs.LO]`. URL: `https://arxiv.org/abs/2509.08610`.

[46] Jan Baumeister, Bernd Finkbeiner, Franz Jünger, Florian Kohn, Sebastian Schirmer, and Christoph Torens. "A ROS Adapter for RTLola". In: *Runtime Verification*. Ed. by Bettina Könighofer and Hazem Torfah. Cham: Springer Nature Switzerland, 2026, pp. 92–100. ISBN: 978-3-032-05435-7.

[47] Christoph Torens, Franz Jünger, Sebastian Schirmer, Pranav Nagarajan, Simon Schopferer, Dmytro Zhukov, and Johann Dauer. "Runtime monitoring of operational design domain to safeguard machine learning components". In: *CEAS Aeronautical Journal* (Sept. 11, 2025). ISSN: 1869-5590. DOI: `10.1007/s13272-025-00883-6`. URL: `https://doi.org/10.1007/s13272-025-00883-6`.

[48] Jan Baumeister, Bernd Finkbeiner, Florian Kohn, Florian Löhr, Guido Manfredi, Sebastian Schirmer, and Christoph Torens. "Monitoring Unmanned Aircraft: Specification, Integration, and Lessons-Learned". In: *Computer Aided Verification*. Ed. by Arie Gurfinkel and Vijay Ganesh. Cham: Springer Nature Switzerland, 2024, pp. 207–218. ISBN: 978-3-031-65630-9.

[49] Sebastian Schirmer, Jasdeep Singh, Emily Jensen, Johann Dauer, Bernd Finkbeiner, and Sriram Sankaranarayanan. "Temporal Behavior Trees: Robustness and Segmentation". In: *Proceedings of the 27th ACM International Conference on Hybrid Systems: Computation and Control*. HSCC '24. Hong Kong SAR, China: Association for Computing Machinery, 2024. ISBN: 9798400705229. DOI: `10.1145/3641513.3650180`. URL: `https://doi.org/10.1145/3641513.3650180`.

[50] Jan Baumeister, Johann C. Dauer, Bernd Finkbeiner, and Sebastian Schirmer. "Monitoring with verified guarantees". In: *Int. J. Softw. Tools Technol. Transf.* 25.4 (2023), pp. 593–616. DOI: `10.1007/S10009-023-00712-3`. URL: `https://doi.org/10.1007/s10009-023-00712-3`.

[51] Sebastian Schirmer, Christoph Torens, Johann C Dauer, Jan Baumeister, Bernd Finkbeiner, and Kristin Y Rozier. "A hierarchy of monitoring properties for autonomous systems". In: *AIAA SCITECH 2023 Forum*. 2023, p. 2588.

[52] Sebastian Schirmer and Christoph Torens. "Safe Operation Monitoring for Specific Category Unmanned Aircraft". In: *Automated Low-Altitude Air Delivery: Towards Autonomous Cargo Transportation with Drones*. Ed. by Johann C. Dauer. Cham: Springer International Publishing, 2022, pp. 393–419. ISBN: 978-3-030-83144-8. DOI: `10.1007/978-3-030-83144-8_16`. URL: `https://doi.org/10.1007/978-3-030-83144-8_16`.

[53] Baumeister. "Tracing Correctness: A Practical Approach to Traceable Runtime Monitoring". Master Thesis. Saarland University, 2020.

[54] Maximilian Schwenger. "Statically-analyzed stream monitoring for cyber-physical Systems". PhD thesis. 2022. DOI: `http://dx.doi.org/10.22028/D291-37014`.

[55] Jan Baumeister. "tbd". PhD thesis. 2026.

[56] Matteo Iovino, Edvards Scukins, Jonathan Styrud, Petter Ögren, and Christian Smith. "A survey of behavior trees in robotics and ai". In: *Robotics and Autonomous Systems* 154 (2022), p. 104096.

[57] Rahib H. Abiyev, Nurullah Akkaya, and Ersin Aytac. “Control of soccer robots using behaviour trees”. In: *9th Asian Control Conference, ASCC 2013, Istanbul, Turkey, June 23-26, 2013*. IEEE, 2013, pp. 1–6. DOI: `10.1109/ASCC.2013.6606326`. URL: `https://doi.org/10.1109/ASCC.2013.6606326`.

[58] Kirk Y. W. Scheper, Sjoerd Tijmons, Cornelis C. de Visser, and Guido C. H. E. de Croon. “Behavior Trees for Evolutionary Robotics†”. In: *Artificial Life* 22.1 (2016), pp. 23–48. ISSN: 1064-5462. DOI: `10.1162/ARTL_a_00192`. eprint: `https://direct.mit.edu/artl/article-pdf/22/1/23/1665258/artl\_a\_00192.pdf`. URL: `https://doi.org/10.1162/ARTL%5C_a%5C_00192`.

[59] Razan Ghzouli, Thorsten Berger, Einar Broch Johnsen, Swaib Dragule, and Andrzej Wąsowski. “Behavior Trees in Action: A Study of Robotics Applications”. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2020. Virtual, USA: Association for Computing Machinery, 2020, pp. 196–209. ISBN: 9781450381765. DOI: `10.1145/3426425.3426942`. URL: `https://doi.org/10.1145/3426425.3426942`.

[60] Zhuochao He, Xuyang Zhang, Simon Jones, Sabine Hauert, Dandan Zhang, and Nathan F. Lepora. “TacMMs: Tactile Mobile Manipulators for Warehouse Automation”. In: *IEEE Robotics and Automation Letters* 8.8 (2023), pp. 4729–4736. DOI: `10.1109/LRA.2023.3287363`.

[61] Aleksandr Sidorenko, Jesko Hermann, and Martin Ruskowski. “Using Behavior Trees for Coordination of Skills in Modular Reconfigurable CPPMs”. In: *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2022, pp. 1–8. DOI: `10.1109/ETFA52439.2022.9921558`.

[62] Hao Hu, Xiaoliang Jia, Kuo Liu, and Bingyang Sun. “Self-Adaptive Traffic Control Model With Behavior Trees and Reinforcement Learning for AGV in Industry 4.0”. In: *IEEE Transactions on Industrial Informatics* 17.12 (2021), pp. 7968–7979. DOI: `10.1109/TII.2021.3059676`.

[63] Danying Hu, Yuanzheng Gong, Blake Hannaford, and Eric J Seibel. “Semi-autonomous simulated brain tumor ablation with Raven II surgical robot using behavior trees”. In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2015, pp. 3868–3875.

[64] Anja Johansson and Pierangelo Dell’Acqua. “Emotional behavior trees”. In: *2012 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE. 2012, pp. 355–362.

[65] M. Mateas and A. Stern. “A behavior language for story-based believable agents”. In: *IEEE Intelligent Systems* 17.4 (2002), pp. 39–47. DOI: `10.1109/MIS.2002.1024751`.

[66] A. v. Perger, P. Gamper, and R. Witzmann. “Behavior Trees for Smart Grid Control”. In: *IFAC-PapersOnLine* 55.9 (2022). 11th IFAC Symposium on Control of Power and Energy Systems CPES 2022, pp. 122–127. ISSN: 2405-8963. DOI: `https://doi.org/10.1016/j.ifacol.2022.07.022`. URL: `https://www.sciencedirect.com/science/article/pii/S2405896322004074`.

[67] Andreas Bauer, Martin Leucker, and Christian Schallhart. “The Good, the Bad, and the Ugly, But How Ugly Is Ugly?” In: *Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers*. Ed. by Oleg Sokolsky and Serdar Tasiran. Vol. 4839. Lecture Notes in Computer Science. Springer, 2007, pp. 126–138. DOI: `10.1007/978-3-540-77395-5\_11`. URL: `https://doi.org/10.1007/978-3-540-77395-5%5C_11`.

[68] Giuseppe De Giacomo and Moshe Y. Vardi. “Linear Temporal Logic and Linear Dynamic Logic on Finite Traces”. In: *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*. IJCAI ’13. Beijing, China: AAAI Press, 2013, pp. 854–860. ISBN: 9781577356332.

[69] Georgios E Fainekos and George J Pappas. “Robustness of temporal logic specifications”. In: *International Workshop on Formal Approaches to Software Testing*. Springer. 2006, pp. 178–192.

[70] Alexandre Donzé and Oded Maler. “Robust satisfaction of temporal logic over real-valued signals”. In: *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer. 2010, pp. 92–106.

[71] Aurélien Rizk, Grégory Batt, François Fages, and Sylvain Soliman. “On a Continuous Degree of Satisfaction of Temporal Logic Formulae with Applications to Systems Biology”. In: *Computational Methods in Systems Biology*. Springer Berlin Heidelberg, 2008, pp. 251–268.

[72] Jyotirmoy V Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A Seshia. “Robust online monitoring of signal temporal logic”. In: *Formal Methods in System Design* 51 (2017), pp. 5–30.

[73] Martin Leucker and César Sánchez. “Regular linear temporal logic”. In: *International colloquium on theoretical aspects of computing*. Springer. 2007, pp. 291–305.

[74] Antonio Cau, Ben Moszkowski, and Hussein Zedan. “Interval temporal logic”. In: *URL: http://www. cms. dmu. ac. uk/˜ cau/itlhomepage/itlhomepage. html* (2006).

[75] Joseph Halpern, Zohar Manna, and Ben Moszkowski. “A hardware semantics based on temporal intervals”. In: *Automata, Languages and Programming*. Ed. by Josep Diaz. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 278–291. ISBN: 978-3-540-40038-7.

[76] D. Harel and D. Peleg. "Process logic with regular formulas". In: *Theoretical Computer Science* 38 (1985). ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(85)90225-7. URL: https://www.sciencedirect.com/science/article/pii/0304397585902257.

[77] Roni Rosner and Amir Pnueli. "A Choppy Logic". In: *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science (LICS 1986)*. Cambridge, MA, USA: IEEE Computer Society Press, 1986, pp. 306–313.

[78] B. C. Moszkowski. "Compositional Reasoning using Interval Temporal Logic and Tempura". In: *Compositionality: The Significant Difference*. Ed. by Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 439–464. ISBN: 978-3-540-49213-9.

[79] Dogan Ulus, Thomas Ferrère, Eugene Asarin, and Oded Maler. "Timed Pattern Matching". In: *Formal Modeling and Analysis of Timed Systems*. Ed. by Axel Legay and Marius Bozga. Cham: Springer International Publishing, 2014, pp. 222–236. ISBN: 978-3-319-10512-3.

[80] Robert A. Wagner. "Order-n Correction for Regular Languages". In: *Commun. ACM* 17.5 (1974), pp. 265–268. DOI: 10.1145/360980.360995. URL: https://doi.org/10.1145/360980.360995.

[81] Michael Benedikt, Gabriele Puppis, and Cristian Riveros. "Regular Repair of Specifications". In: *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*. IEEE Computer Society, 2011, pp. 335–344. DOI: 10.1109/LICS.2011.43. URL: https://doi.org/10.1109/LICS.2011.43.

[82] Vasumathi Raman, Mehdi Maasoumy, and Alexandre Donzé. "Model predictive control from signal temporal logic specifications: A case study". In: *Proceedings of the 4th ACM SIGBED International Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems*. 2014, pp. 52–55.

[83] Parv Kapoor, Eunsuk Kang, and Rômulo Meira-Góes. "Safe Planning Through Incremental Decomposition of Signal Temporal Logic Specifications". In: *NASA Formal Methods Symposium*. Springer. 2024, pp. 377–396.

[84] Christel Baier, Norine Coenen, Bernd Finkbeiner, Florian Funke, Simon Jantsch, and Julian Siber. "Causality-Based Game Solving". In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12759. Lecture Notes in Computer Science. Springer, 2021, pp. 894–917. DOI: 10.1007/978-3-030-81685-8_42. URL: https://doi.org/10.1007/978-3-030-81685-8%5C_42.

[85] Jörg Hoffmann, Julie Porteous, and Laura Sebastia. "Ordered Landmarks in Planning". In: *CoRR* abs/1107.0052 (2011). arXiv: 1107.0052. URL: http://arxiv.org/abs/1107.0052.

[86] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. "S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems". In: *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. Ed. by Parosh Aziz Abdulla and K. Rustan M. Leino. Vol. 6605. Lecture Notes in Computer Science. Springer, 2011, pp. 254–257. DOI: 10.1007/978-3-642-19835-9_21. URL: https://doi.org/10.1007/978-3-642-19835-9%5C_21.

[87] Sota Sato, Jie An, Zhenya Zhang, and Ichiro Hasuo. "Optimization-Based Model Checking and Trace Synthesis for Complex STL Specifications". In: *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part III*. Ed. by Arie Gurfinkel and Vijay Ganesh. Vol. 14683. Lecture Notes in Computer Science. Springer, 2024, pp. 282–306. DOI: 10.1007/978-3-031-65633-0_13. URL: https://doi.org/10.1007/978-3-031-65633-0%5C_13.

[88] Russell V Gilabert, Evan T Dill, Kelly J Hayhurst, and Steven D Young. "SAFEGUARD: Progress and test results for a reliable independent on-board safety net for UAS". In: *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*. IEEE. 2017, pp. 1–9.

[89] Evan T Dill, Russell V Gilabert, and Seth S Young. "Safeguard". In: *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*. IEEE. 2018, pp. 1–8.

[90] Kristin Yvonne Rozier. "On the Evaluation and Comparison of Runtime Verification Tools for Hardware and Cyber-Physical Systems." In: *RV-CuBES*. 2017, pp. 123–137.

[91] Mark A. Skoog, Loyd R. Hook, and Wes Ryan. "Leveraging ASTM Industry Standard F3269-17 for Providing Safe Operations of a Highly Autonomous Aircraft". In: *2020 IEEE Aerospace Conference*. 2020, pp. 1–7. DOI: 10.1109/AERO47225.2020.9172434.

[92] Kyle Ellis, Paul Krois, Misty D Davirs, and John Koelling. *In-time system-wide safety assurance (ISSA) concept of operations*. Tech. rep. 2019.

[93] Mia N. Stevens and Ella M. Atkins. "Layered Geofences in Complex Airspace Environments". In: *2018 Aviation Technology, Integration, and Operations Conference*. DOI: 10.2514/6.2018-3348. eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2018-3348. URL: https://arc.aiaa.org/doi/abs/10.2514/6.2018-3348.

[94] Mia N. Stevens, Hossein Rastgoftar, and Ella M. Atkins. "Geofence Boundary Violation Detection in 3D Using Triangle Weight Characterization with Adjacency". In: *J. Intell. Robotic Syst.* 95.1 (2019), pp. 239–250. DOI: 10.1007/S10846-018-0930-5. URL: https://doi.org/10.1007/s10846-018-0930-5.

[95] Sebastian Schirmer, Christoph Torens, and Florian Adolf. "Formal Monitoring of Risk-based Geofences". In: *2018 AIAA Information Systems-AIAA Infotech @ Aerospace*. DOI: `10.2514/6.2018-1986`. eprint: `https://arc.aiaa.org/doi/pdf/10.2514/6.2018-1986`. URL: `https://arc.aiaa.org/doi/abs/10.2514/6.2018-1986`.

[96] Dimitry Solet, Jean-Luc Béchennec, Mikaöl Briday, Sébastien Faucou, and Sébastien Pillement. "Hardware runtime verification of embedded software in SoPC". In: *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*. 2016, pp. 1–6. DOI: `10.1109/SIES.2016.7509425`.

[97] Thang Nguyen, Ezio Bartocci, Dejan Ničković, Radu Grosu, Stefan Jaksic, and Konstantin Selyunin. "The HARMONIA Project: Hardware Monitoring for Automotive Systems-of-Systems". In: *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer International Publishing, 2016, pp. 371–379. ISBN: 978-3-319-47169-3.

[98] Joseph Stamenkovich, Lakshman Maalolan, and Cameron Patterson. "Formal Assurances for Autonomous Systems Without Verifying Application Software". In: *2019 Workshop on Research, Education and Development of Unmanned Aerial Systems (RED UAS)*. 2019, pp. 60–69. DOI: `10.1109/REDUAS47371.2019.8999690`.

[99] N. Koenig and A. Howard. "Design and use paradigms for Gazebo, an open-source multi-robot simulator". In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*. Vol. 3. 2004, 2149–2154 vol.3. DOI: `10.1109/IROS.2004.1389727`.

[100] Alexey Dosovitskiy, Germán Ros, Felipe Codevilla, Antonio M. López, and Vladlen Koltun. "CARLA: An Open Urban Driving Simulator". In: *1st Annual Conference on Robot Learning, CoRL 2017, Mountain View, California, USA, November 13-15, 2017, Proceedings*. Vol. 78. Proceedings of Machine Learning Research. PMLR, 2017, pp. 1–16. URL: `http://proceedings.mlr.press/v78/dosovitskiy17a.html`.

[101] Tomoya Yamaguchi, Bardh Hoxha, and Dejan Ničković. "RTAMT – Runtime Robustness Monitors with Application to CPS and Robotics". In: *International Journal on Software Tools for Technology Transfer* 26.1 (2023), pp. 79–99. ISSN: 1433-2787. DOI: `10.1007/s10009-023-00720-3`. URL: `http://dx.doi.org/10.1007/S10009-023-00720-3`.

[102] Ivan Perez, Anastasia Mavridou, Thomas Pressburger, Alexander Will, and Patrick J. Martin. "Monitoring ROS2: from Requirements to Autonomous Robots". In: *Proceedings Fourth International Workshop on Formal Methods for Autonomous Systems (FMAS) and Fourth International Workshop on Automated and verifiable Software sYstem DEvelopment (ASYDE), FMAS/ASYDE@SEFM 2022,*

and Fourth International Workshop on Automated and verifiable Software sYstem DEvelopment (ASYDE)Berlin, Germany, 26th and 27th of September 2022. Ed. by Matt Luckcuck and Marie Farrell. Vol. 371. EPTCS. 2022, pp. 208–216. DOI: `10.4204/EPTCS.371.15`. URL: `https://doi.org/10.4204/EPTCS.371.15`.

[103] Marian Johannes Begemann, Hannes Kallwies, Martin Leucker, and Malte Schmitz. "TeSSLa-ROS-Bridge - Runtime Verification of Robotic Systems". In: *Theoretical Aspects of Computing - ICTAC 2023 - 20th International Colloquium, Lima, Peru, December 4-8, 2023, Proceedings*. Ed. by Erika Ábrahám, Clemens Dubslaff, and Silvia Lizeth Tapia Tarifa. Vol. 14446. Lecture Notes in Computer Science. Springer, 2023, pp. 388–398. DOI: `10.1007/978-3-031-47963-2\_23`. URL: `https://doi.org/10.1007/978-3-031-47963-2%5C_23`.

[104] Maryam Ghaffari Saadat, Angelo Ferrando, Louise A. Dennis, and Michael Fisher. "ROSMonitoring 2.0: Extending ROS Runtime Verification to Services and Ordered Topics". In: *Proceedings Sixth International Workshop on Formal Methods for Autonomous Systems, FMAS@iFM 2024, Manchester, UK, 11th and 12th of November 2024*. Ed. by Matt Luckcuck and Mengwei Xu. Vol. 411. EPTCS. 2024, pp. 38–55. DOI: `10.4204/EPTCS.411.3`. URL: `https://doi.org/10.4204/EPTCS.411.3`.

[105] Bernd Finkbeiner, Stefan Oswald, Noemi Passing, and Maximilian Schwenger. "Verified Rust Monitors for Lola Specifications". In: *Runtime Verification*. Ed. by Jyotirmoy Deshmukh and Dejan Ničković. Cham: Springer International Publishing, 2020, pp. 431–450. ISBN: 978-3-030-60508-7.

[106] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 41–62. ISBN: 978-3-662-49122-5.

[107] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580. ISSN: 0001-0782. DOI: `10.1145/363235.363259`. URL: `https://doi.org/10.1145/363235.363259`.

[108] Robert W. Floyd. "Assigning Meanings to Programs". In: *Program Verification: Fundamental Issues in Computer Science*. Ed. by Timothy R. Colburn, James H. Fetzer, and Terry L. Rankin. Dordrecht: Springer Netherlands, 1993, pp. 65–81. ISBN: 978-94-011-1793-7. URL: `https://doi.org/10.1007/978-94-011-1793-7_4`.

[109] Bernhard Beckert, Reiner Hähnle, and Peter H Schmitt, eds. *Verification of Object-Oriented Software. The KeY Approach*. Vol. 4334. LNCS 4334. Springer-Verlag, 2007. ISBN: 978-3-540-68977-5. DOI: `10.1007/978-3-540-69061-0`. arXiv: `11.1669v3`.

[110] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. "Let's verify this with Why3". In: *International Journal on Software Tools for Technology Transfer* 17 (2015), pp. 709–727.

[111] K. Rustan M. Leino. “Dafny: An Automatic Program Verifier for Functional Correctness”. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Edmund M. Clarke and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370. ISBN: 978-3-642-17511-4.

[112] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. “The synchronous data flow programming language LUSTRE”. In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1320. DOI: `10.1109/5.97300`.

[113] Gérard Berry. “The Foundations of Esterel”. In: *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. Cambridge, MA, USA: MIT Press, 2000, pp. 425–454. ISBN: 0262161885.

[114] Thierry Gautier, Paul Le Guernic, and Loïc Besnard. “SIGNAL: A declarative language for synchronous programming of real-time systems”. In: *Functional Programming Languages and Computer Architecture*. Ed. by Gilles Kahn. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 257–277. ISBN: 978-3-540-47879-9.

[115] Lalita Jategaonkar Jagadeesan, Carlos Puchol, and James E. Von Olnhausen. “Safety property verification of Esterel programs and applications to telecommunications software”. In: *Computer Aided Verification*. Ed. by Pierre Wolper. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 127–140. ISBN: 978-3-540-49413-3.

[116] G. Hagen and C. Tinelli. “Scaling Up the Formal Verification of Lustre Programs with SMT-Based Techniques”. In: *2008 Formal Methods in Computer-Aided Design*. 2008, pp. 1–9. DOI: `10.1109/FMCAD.2008.ECP.19`.

[117] Yahui Song and Wei-Ngan Chin. “A Synchronous Effects Logic for Temporal Verification of Pure Esterel”. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Fritz Henglein, Sharon Shoham, and Yakir Vizel. Cham: Springer International Publishing, 2021, pp. 417–440. ISBN: 978-3-030-67067-2.

[118] Sebastian Schirmer, Christoph Torens, Florian Nikodem, and Johann Dauer. “Considerations of Artificial Intelligence Safety Engineering for Unmanned Aircraft”. In: *Computer Safety, Reliability, and Security*. Ed. by Barbara Gallina, Amund Skavhaug, Erwin Schoitsch, and Friedemann Bitsch. Cham: Springer International Publishing, 2018, pp. 465–472. ISBN: 978-3-319-99229-7.

[119] Ricardo A. Arteaga, Nickolas Demidovich, John Dinofrio, Chuck Greenlow, Timothy Williams, and James Nelson. “Flight Testing ADS B and GPS LPV Approaches on a NASA ER-2 Research Airplane”. In: *AIAA SCITECH 2024 Forum*. DOI: `10.2514/6.2024-0307`. eprint: `https://arc.aiaa.org/doi/pdf/10.2514/6.2024-0307`. URL: `https://arc.aiaa.org/doi/abs/10.2514/6.2024-0307`.

[120] Bowen Alpern and Fred B. Schneider. “Recognizing Safety and Liveness”. In: *Distributed Comput.* 2.3 (1987), pp. 117–126. DOI: `10.1007/BF01782772`. URL: `https://doi.org/10.1007/BF01782772`.

[121] Bowen Alpern and Fred B. Schneider. “Defining Liveness”. In: *Inf. Process. Lett.* 21.4 (1985), pp. 181–185. DOI: `10.1016/0020-0190(85)90056-0`. URL: `https://doi.org/10.1016/0020-0190(85)90056-0`.

[122] Zohar Manna and Amir Pnueli. “A Hierarchy of Temporal Properties”. In: *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing, Quebec City, Quebec, Canada, August 22-24, 1990*. Ed. by Cynthia Dwork. ACM, 1990, pp. 377–410. DOI: `10.1145/93385.93442`. URL: `https://doi.org/10.1145/93385.93442`.

[123] Jianwen Li, Moshe Y. Vardi, and Kristin Y. Rozier. “Satisfiability Checking for Mission-Time LTL”. In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11562. Lecture Notes in Computer Science. Springer, 2019, pp. 3–22. DOI: `10.1007/978-3-030-25543-5\_1`. URL: `https://doi.org/10.1007/978-3-030-25543-5%5C_1`.

[124] Andreas Bauer, Martin Leucker, and Christian Schallhart. “Runtime verification for LTL and TLTL”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20.4 (2011), pp. 1–64.

[125] Jan Baumeister, Bernd Finkbeiner, Florian Kohn, and Frederik Scheerer. “A Tutorial on Stream-based Monitoring”. In: *CoRR* abs/2501.15913 (2025). DOI: `10.48550/ARXIV.2501.15913`. arXiv: `2501.15913`. URL: `https://doi.org/10.48550/arXiv.2501.15913`.

[126] Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Hazem Torfah. “StreamLAB: Stream-based Monitoring of Cyber-Physical Systems”. In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11561. Lecture Notes in Computer Science. Springer, 2019, pp. 421–431. DOI: `10.1007/978-3-030-25540-4\_24`. URL: `https://doi.org/10.1007/978-3-030-25540-4%5C_24`.

[127] Ezio Bartocci, Luca Bortolussi, Michele Loreti, Laura Nenzi, and Simone Silvetti. “MoonLight: A Lightweight Tool for Monitoring Spatio-Temporal Properties”. In: *Runtime Verification*. Ed. by Jyotirmoy Deshmukh and Dejan Ničković. Cham: Springer International Publishing, 2020, pp. 417–428. ISBN: 978-3-030-60508-7.

[128] M Loreti, Luca Bortolussi, E Bartocci, and L Nenzi. “A Logic for Monitoring Dynamic Networks of Spatially-distributed Cyber-Physical Systems”. In: *Logical Methods in Computer Science* 18 (2022).

[129] Oded Maler and Dejan Nickovic. "Monitoring temporal properties of continuous signals". In: *International symposium on formal techniques in real-time and fault-tolerant systems*. Springer. 2004, pp. 152–166.

[130] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. "LOLA: Runtime Monitoring of Synchronous Systems". In: *12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA*. IEEE Computer Society, 2005, pp. 166–174. DOI: 10.1109/TIME.2005.26. URL: https://doi.org/10.1109/TIME.2005.26.

[131] Bernd Finkbeiner and Henny Sipma. "Checking finite traces using alternating automata". In: *Formal Methods in System Design* 24 (2004), pp. 101–127.

[132] Maximilian Schwenger. "Statically-analyzed stream monitoring for cyber-physical Systems". PhD thesis. Saarland University, Saarbrücken, Germany, 2022. URL: https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/33890.

[133] SAE International. *Validation and Verification Process Steps for Monitors Development in Complex Flight Control and Related Systems*. ARP Standard ARP6539. 2022. URL: https://doi.org/10.4271/ARP6539.

[134] Michele Colledanchise and Petter Ögren. *Behavior trees in robotics and AI: An introduction*. CRC Press, 2018.

[135] Pierre Wolper. "Temporal logic can be more expressive". In: *Information and control* 56.1-2 (1983), pp. 72–99.

[136] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques & Tools*. Pearson Education, 2007, pp. 159–163.

[137] Ryan Matheu, Aniruddh G. Puranic, John S. Baras, and Calin Belta. "OMTBT: Online Monitoring of Temporal Behavior Trees with Applications to Closed-Loop Learning". In: *2025 European Control Conference (ECC)*. 2025, pp. 2129–2135. DOI: 10.23919/ECC65951.2025.11187275.

[138] Ana Lúcia De Moura and Roberto Ierusalimschy. "Revisiting Coroutines". In: *ACM Trans. Program. Lang. Syst.* 31.2 (2009). ISSN: 0164-0925. DOI: 10.1145/1462166.1462167. URL: https://doi.org/10.1145/1462166.1462167.

[139] Christopher D Marlin. "Coroutines: A Programming Methodology, a Language Design, and an Implementation". PhD thesis. University of Adelaide, Department of Computing Science, 1979.

[140] Sooyung Byeon, Joonwon Choi, Yutong Zhang, and Inseok Hwang. "Stochastic-Skill-Level-Based Shared Control for Human Training in Urban Air Mobility Scenario". In: *J. Hum.-Robot Interact.* (2023). Just Accepted. DOI: 10.1145/3603194. URL: https://doi.org/10.1145/3603194.

[141] Bianca Isabella Schuchardt, Thomas Dautermann, Alexander Donkels, Stefan Krause, Niklas Peinecke, and Gunnar Schwoch. "Maritime operation of an unmanned rotorcraft with tethered ship deck landing system". In: *CEAS Aeronautical Journal*. CEAS Aeronautical Journal 12.1 (2020), pp. 1–9. URL: `https://elib.dlr.de/140951/`.

[142] T. Schmelz and R. Lantzsch. "Abschlussbericht: F&T Studie - Pilotenassistenz für Schiffsdecklandungen (PiloDeck)[Final report: F&T Study - Pilot assitance for ship deck landing (PiloDeck)]," in: *Technical Note AHD-TN-ESPE-302-18* (2018).

[143] Alberto Bemporad, Manfred Morari, Vivek Dua, and Efstratios N. Pistikopoulos. "The explicit linear quadratic regulator for constrained systems". In: *Automatica* 38.1 (2002), pp. 3–20. ISSN: 0005-1098. DOI: `https://doi.org/10.1016/S0005-1098(01)00174-1`. URL: `https://www.sciencedirect.com/science/article/pii/S0005109801001741`.

[144] Hilary P. Williams. *Model Building in Mathematical Programming*. 1st. New York Academy of Sciences Series. John Wiley & Sons, Incorporated, 2013.

[145] Robert J. Vanderbei. *Linear Programming: Foundations and Extensions*. 5th. International Series in Operations Research & Management Science. Springer International Publishing, 2020.

[146] Sebastian Schirmer, Jasdeep Singh, Emily Jensen, Johann Dauer, Bernd Finkbeiner, and Sriram Sankaranarayanan. *Temporal Behavior Trees*. Version v1.0.0. 2024. DOI: `10.5281/zenodo.13807484`. URL: `https://doi.org/10.5281/zenodo.13807484`.

[147] T.J. Koo and S. Sastry. "Output Tracking Control Design of a Helicopter Model Based on Approximate Linearization". In: *Proceedings of the 37th IEEE Conference on Decision and Control*. Vol. 4. Tampa, FL, USA: IEEE, 1998, pp. 3635–3640. DOI: `10.1109/CDC.1998.761745`.

[148] Joint Authorities for Rulemaking on Unmanned Systems (JARUS). *JARUS guidelines on Specific Operations Risk Assessment (SORA)*. Version 2.5; Public release. 2024. URL: `http://jarus-rpas.org/wp-content/uploads/2024/06/SORA-v2.5-Main-Body-Release-JAR_doc_25.pdf`.

[149] Sebastian Schirmer. "Runtime Monitoring with Lola". MA thesis. Saarland University, 2016.

[150] Abdel Zaro, Ardalan Tajbakhsh, and Aaron M Johnson. "Collision detection for multi-robot motion planning with efficient quad-tree update and skipping". In: *arXiv preprint arXiv:2307.07602* (2023).

List of Figures

List of Tables