POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Master's degree in Automation and Control Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria

SMT-based Trace Checking of CPS Properties

Supervisor: Prof. Matteo Rossi
Co-supervisors: Dr. Ing. Claudio Menghi
Prof. Domenico Bianculli
Prof. Lionel C. Briand

Master's thesis of:
Enrico Viganò, Matr. 905384

Academic Year 2019-2020

Alle mie nonne,
quelle meravigliose
malefiche vecchiette

Sommario

Al giorno d'oggi, componenti informatiche sono sempre più utilizzate per sostituire l'uomo in attività complesse, dove è necessario che interagiscano con l'ambiente fisico in cui sono collocati. Forme di trasporto automatizzato, applicazioni di domotica, sistemi medicali di nuova generazione e la nascita del cosiddetto "Internet of Things" sono solo esempi del fatto che è ormai quasi impossibile trovare un dominio della tecnica in cui l'informatica non sia penetrata, aumentando notevolmente le capacità, ma di pari passo anche la complessità, dei macchinari che ci circondano. In sintesi, la maggior parte dei nostri sistemi stanno diventando, o sono già diventati, ciber-fisici, e sono spesso impiegati in situazioni in cui la sicurezza e l'affidabilità sono aspetti critici. Ne consegue che lo sviluppo di tecniche che consentano un'analisi rigorosa di questo tipo di sistemi è diventato di fondamentale importanza. Tuttavia, analizzare i Sistemi Ciber-Fisici significa essere in grado di descrivere quello che spesso risulta essere un amalgama complesso di due dinamiche molto diverse tra loro: l'evoluzione delle variabili fisiche, tipicamente rappresentate come segnali continui, e il comportamento delle componenti software, solitamente rappresentato da una sequenza di cambiamenti di stato e modellato attraverso sistemi ad eventi discreti. Inoltre, la maggior parte di questi sistemi tendono ad essere notevolmente complessi e presentano un numero elevato di variabili e parametri da tenere sotto osservazione per ottenere una descrizione sufficientemente esaustiva del loro comportamento.

Questa quantità sempre crescente di dati da organizzare e interpretare ha portato allo sviluppo di tecniche di verifica automatizzate ad essi dedicate. Il Trace-checking è una di queste tecniche. Essa mira a produrre un verdetto riguardante la conformità di un sistema ad un certo requisito, confrontando la traccia di esecuzione di un sistema, ovvero la misurazione di un insieme di variabili che ne descrivono lo stato durante le sue operazioni, con il suo comportamento ideale, espresso attraverso formule logiche scritte in un adeguato linguaggio di specificazione. Può essere applicata sia a sistemi software che

a sistemi hardware, purché sia possibile registrare una traccia, e più recentemente si è iniziato ad usarla anche per i sistemi ciber-fisici.

Ci sono due categorie principali di linguaggi usati per specificare i requisiti logici: i linguaggi *time-based*, ovvero basati sul tempo e quelli basati sul concetto di sequnza di eventi (*sequence-based*). I linguaggi basati sul tempo, che interpretano le variabili come segnali in un dominio temporale, sono adatti ad esprimere i requisiti relativi alle grandezze fisiche, ma non sono facilmente adattabili a specificare i requisiti relativi ai componenti software. Al contrario, nei linguaggi sequence-based, le tracce sono definite tramite sequenze ordinate di eventi consecutivi (come quelle prodotte dagli automi a stati finiti) che risultano adeguate per descrivere la dinamica delle componenti informatiche, ma non ottimali per rappresentare l'evoluzione di grandezze del mondo fisico. Ad oggi sono stati sviluppati anche linguaggi ibridi che tentano di supportare la specificazione di entrambi i tipi di comportamento, ma sono solitamente estensioni ad hoc di linguaggi esistenti, focalizzati su contesti specifici e che spesso ereditano alcune delle limitazioni del linguaggio di partenza.

Oltretutto, esiste generalmente un rapporto di proporzionalità inversa tra l'espressività del linguaggio di specifica e l'efficienza della procedura di trace-checking; di conseguenza, occorre trovare un compromesso tra la potenza espressiva del linguaggio e l'immediatezza della risposta, affinché sia possibile avvalersi della tecnica sviluppata nella pratica. Questa è una delle principali sfide da affrontare quando si progetta un approccio di trace-checking per uso industriale.

In questo contesto, considerando le sfide e le necessità dello sviluppo e del testing dei sistemi ciber-fisici, presentiamo *Hybrid Logic of Signals*, in breve HLS, un nuovo linguaggio di specifica pensato per questa classe di sistemi, e *ThEodorE* (*Logic-based TracE checkEr for HLS*), un algoritmo di trace-checking automatico svluppato per le proprietà espresse in HLS.

HLS supporta l'espressione di requisiti riferiti sia ai *timestamps*, ovvero ai tempi di acquisizione dei record della traccia, che agli indici numerici dei record. Diventa quindi possibile catturare il comportamento delle componenti sia informatiche che fisiche e le loro interazioni, utilizzando gli indici per esprimere i cambiamenti discontinui di stato e i timestamp per catturare le relazioni temporali degli aspetti continui del sistema.

ThEodorE riduce il problema del controllo delle proprietà HLS in un problema di *Satisfiability Modulo Theories*, che puo essere risolto da numerosi solver già esistenti, i quali incorporano procedure decisionali efficienti per diverse teorie logiche, rendendo così possibile verificare se la fomula che rappresenta l'unione della proprietà espressa e del comportamento della traccia è soddisfatta o meno.

ThEodorE è essenzialmente un *Domain Specific Language* (Linguaggio di Dominio Specifico) sviluppato per esprimere le proprietà logiche in HSL e assegnarle alla traccia a cui si riferiscono, dotato di un generatore di codice per tradurre le strutture dati create in questo modo nella logica SMT di destinazione.

Abbiamo implementato ThEodorE usando Eclipse Xtext, uno strumento per sviluppare linguaggi di programmazione, e Xtend, un dialetto di Java pensato per essere usato assieme a Xtext, e lo abbiamo finalizzato nella forma di un plugin per Eclipse IDE.

Abbiamo valutato l'espressività di HSL e l'applicabilità di ThEodorE attraverso un caso di studio fornitoci dal nostro partner industriale, LuxSpace.

LuxSpace è un azienda del settore aerospaziale che ha sviluppato un satellite per raccogliere informazioni di tracciamento dalle navi e per trasmettere questi dati a terra. Rappresenta un caso emblematico di sistema ciber-fisco composto da componenti software complessi che interagiscono con un gran numero di attuatori e sensori e con l'ambiente fisico circostante. Visto il contesto di impiego, il settore industriale e l'utilizzo previsto per il satellite, esso è oggetto di molti requisiti tecnici e di funzionamento che deve soddisfare.

Abbiamo preso in considerazione 212 di questi requisiti e abbiamo cercato di esprimerli utilizzando HSL e due linguaggi di specifica all'avanguardia, ovvero SB-TemPsy-DSL e STL, entrambi supportati, come HSL, da strumenti di trace-checking.

In primo luogo, abbiamo valutato la misura in cui i requisiti erano esprimibili da ogni linguagggio. HSL è stato in grado di esprimere il 100% dei requisiti, mentre SB-TemPsy-DSL e STL sono stati in grado di esprimerne rispettivamente solo il 68% e il 48%.

LuxSpace ci ha anche fornito 20 tracce, ottenute simulando il comportamento del satellite in diversi scenari e per tempi di simulazione che vanno dalle quattro alle sei ore. La loro dimensione varia da 41844 a 1202241 voci.

L'applicabilità di ThEodorE è stata quindi testata su questo dataset contro SB-TemPsy-Check e Breach, i due algoritmi di trace-checking relativi ai suddetti linguaggi. Applicando i 212 requisiti alle tracce di simulazione abbiamo ottenuto 747 combinazioni di tracce e requisiti. ThEodorE è riuscito a calcolare un verdetto definitivo entro un'ora per il 74,5% del totale delle combinazioni. In particolare, è riuscito a valutare il 67,9% delle 337 combinazioni che i linguaggi concorrenti non hanno potuto esprimere.

Quando i requisiti sono esprimibili in SB-TemPsy-DSL e STL, SB-TemPsy-Check e Breach sono più veloci di ThEodorE. Tuttavia, dato lo scenario di utilizzo (controllo delle tracce offline), la differenza nei tempi di esecuzione non ha alcuna conseguenza pratica, poiché il tempo medio di controllo delle tracce, nell'ordine dei minuti, è in ogni caso significativamente inferiore alle ore necessarie per raccogliere le tracce.

Essendo in grado di esprimere un insieme di proprpietà molto più ampio rispetto ad altri sistemi e di verificare le suddette tracce entro limiti di tempo pratici, ThEodorE rappresenta un migliore compromesso tra espressività e performance. Inoltre, si prevede che l'efficienza e l'efficacia di ThEodorE miglioreranno in futuro insieme alla tecnologia SMT sottostante, in quanto i solver in questione sono tuttora oggetto di ricerca e sviluppo continuo.

Abstract

Software is no longer designed only to support humans in complex calculations, but it is more and more used to replace humans in complex activities, where it performs its tasks interacting with a physical environment through hardware systems like sensors and actuators.

Automated transportation, domotics, automated medical systems, and the Internet of Things are just examples of the fact that is now almost impossible to find a technical field in which software has not penetrated, greatly increasing the capabilities, but also the complexity, of our machinery. In synthesis, most of our systems are becoming, or have already become, Cyber-Physical, and they are often deployed in situations where safety and reliability are critical. This means that developing techniques that enable rigorous analysis of these types of systems is of paramount importance. However, analyzing Cyber-Physical Systems means being able to describe the complex amalgamation of two very different dynamics: the evolution of physical variables, typically represented as continuous signals, and the behavior of the software components, usually represented by a sequence of state changes and modeled through discrete event systems. Moreover, most of these systems tend to be fairly complex and present a high number of variables and parameters that need to be kept under observation in order to have a useful insight into their behaviour. This ever-increasing amount of data to organize and interpret requires the development of automated verification techniques aimed at complex systems. Trace-checking is a technique of runtime verification that produces a verdict on the conformity of a system to a certain requirement by comparing the execution trace of a system with its ideal behavior, expressed through a suitable specification language. It can be applied to both software systems and hardware systems, as long as a trace can be made available, and more recently it has also been applied to CPS.

There are two main categories of languages used for specifying CPS requirements: time-based and sequence-based languages. Time-based languages,

which interpret variables as signals over a time domain, are suitable to express CPS requirements related to physical quantities but are not easily amenable to specifying requirements related to software components. On the contrary, in sequence-based languages, traces are defined as sequences of consecutive records of events, which is ideal to describe the dynamics of software components, but not to represent the evolution of physical quantities. Hybrid languages exist to support the specification of both kinds of behaviors, but they are usually ad-hoc extensions of existing languages, focused on specific contexts.

Typically, there is a trade-off between the expressiveness of the specification language and the efficiency of the trace-checking procedure; consequently, there is a delicate balance to strike between the expressive power of the specification language and the immediacy of the response, and this is one of the main challenges to face when designing a trace-checking approach for industrial use.

In this context, considering the challenges and necessities of CPS development and testing, we present the Hybrid Logic of Signals (HLS), a new specification language tailored to specifying CPS requirements, and ThEodorE (Logic-based TracE checkEr for HLS), an efficient trace-checking approach for properties expressed in HLS.

HLS supports the expression of CPS requirements as properties referring both to the time-stamps and the indices of the records of CPS traces, extending existing time and sequence-based languages. The behaviour of both cyber and physical components and their interactions can be captured, using the indices to express discontinuous changes of state, and the timestamps to capture the time relations of the continuous aspects of the system.

ThEodorE translates the problem of trace-checking HLS properties to a satisfiability problem, which can be solved using existing Satisfiability Modulo Theories (SMT) solvers, which incorporate efficient decision procedures for several background theories, thus making it possible to check whether a formula expressed in first-order logic is satisfiable.

ThEodorE is essentially a Domain Specific Language developed to express logic properties in HSL and assign them to the trace they refer to, augmented with a code generator to translate the data structures created this way into the target SMT logic.

We implemented ThEodorE as an Eclipse IDE plugin using Eclipse Xtext, a tool for developing programming languages and DSLs, and Xtend, a Java

dialect used in tandem with Xtext.

We assessed the expressiveness of HSL and the applicability of ThEodorE through a case study provided by our industrial partner, LuxSpace.

LuxSpace developed a satellite to collect tracking information from ships and to transmit those data to the ground, which is an emblematic case of a CPS made of complex software components interacting with many actuators and sensors and the surrounding physical environment, with critical requirements to satisfy regarding all these aspects.

We considered 212 of these requirements and we attempted to express them using HSL and two state-of-the-art specification languages: SB-TemPsy-DSL and STL, both supported by publicly available trace checking tools.

First, we evaluated the extent to which the requirements were expressible in each language. HSL was able to express 100% of the requirements, while SB-TemPsy-DSL and STL were able to express respectively only 68% and 48% of the requirements.

LuxSpace also provided us with 20 traces, obtained by simulating the behavior of the satellite in different scenarios and for simulation times ranging from four to six hours. Their size ranges from 41844 to 1202241 entries.

The applicability of ThEodorE was then tested on this dataset against the aforementioned tools, SB-TemPsy-Check and Breach.

Applying the 212 requirements to the simulation traces we obtained 747 trace-requirement combinations. ThEodorE could compute a definitive verdict within one hour for 74.5% of the total combinations and in particular for 67.9% of the 337 trace-requirement combinations that could not be checked by the other tools due to language limitations.

When the requirements are expressible in SB-TemPsy-DSL and STL, SB-TemPsy-Check and Breach are faster than ThEodorE. However, given the usage scenario (offline trace checking), the difference in execution times does not have any practical consequences since the average trace-checking time is significantly lower than the time required to collect the traces.

Being able to support a much wider range of properties than other trace-checkers and to verify them within practical time limits, our approach achieves a better trade-off between expressiveness and performances. Furthermore, the efficiency and effectiveness of ThEodorE are expected to improve in the future along with the underlying SMT technology.

Acknowledgements

First and foremost, I would like to thank Prof. Matteo Rossi who offered me the opportunity to develop this thesis.

My sincere gratitude goes to Prof. Lionel C. Briand and the Interdisciplinary Centre for Security, Reliability, and Trust of the University of Luxembourg, for making this thesis possible.

I am deeply indebted to Prof. Domenico Bianculli, for the precious advice and continuous support, and to Claudio Menghi, for the guidance, the help, and, most of all, the patience he showed me throughout this journey.

Special thanks to Yago Isasi Parache and LuxSpace, for providing a practical foundation to this project.

I am also very grateful to Giuseppe and Nicolò, for never letting me down when I needed to clear my head.

Finally, thanks should also go to my family, whose support never faltered even in these complicated times, to Federica, for always tolerating me and encouraging me, and to the friends who stuck by my side along these years: Fabiola, Francesco, Francesca, Gabriele, Lorenzo, Luca, Matteo, Miriam, Nicolò, Paolo, Silvia, Sebastiano, Tommaso, and Valentina.

Contents

List of Figures

List of Tables

LuxSpace

Chapter 1

Introduction

This Chapter provides an overview of this thesis. First (Section 1.1) we describe the research context and we present the research problem addressed by this thesis (Section 1.2). Then we describe the contribution of this work (Section 1.3). Finally, we briefly go over the structure of the thesis (Section 1.4).

1.1 Research Context

Software systems are becoming more and more ubiquitous, and one of the reasons is that they are no longer just conceived to support humans in complex calculations, but they are more and more used to replace humans in complex, tedious, time-consuming, or even dangerous tasks. In many of these applications, the software is required to autonomously sense and act on its physical environment. Some examples are autonomous transportation systems, smart electrical grids, domotic systems, and new-generation medical systems. Nowadays is become quite difficult to find a technical field in which software is not employed in one way or another. Software systems guarantee high levels of controllability, by enabling systems to monitor up to hundreds or even thousands of variables, to process the data contained in those variables, and to take the best course of action following a variety of different algorithms and decision procedures. In one word, our systems are now *Cyber-Physical*. As such, they are constantly monitoring their environment, analyzing their own status, and acting accordingly.

Cyber-Physical Systems (CPSs) *monitor* their environment using sensors to detect changes in the surrounding conditions. Clear and accurate assessment of the environmental variables is crucial for the software decision processes and control algorithms to work properly. Environmental conditions may influence the state of the system, prompting changes in the set of variables that describe its behavior. These changes can act as disturbances and very well affect the performance of the system. That is why CPSs *analyze* their status, gathering and interpreting pieces of information about the conditions of their components, their performances, and their eventual malfunctioning and fail-states.

One example of CPSs is our case-study (see Chapter 4), a satellite developed by the european space-system contractor LuxSpace [7]. Satellites usually rely on a set of magnetometers to monitor the status of the magnetic field which changes as they move on their orbits. Earth's magnetic field changes over time, interacting with the solar wind, a stream of charged particles generated by the Sun and can cause disturbances to various functions of the machine, for example, communications. For a system as complex and isolated as a satellite, the ability to collect critical pieces of information about its state, act on them when needed, and relay them to ground is extremely important. For example, it needs to monitor the light exposure on its solar panels to ensure they provide enough energy to sufficiently charge the batteries to keep working when passing into Earth's shadow. Solar trackers orient the panels to the most convenient position, acting on inputs from autonomous subsystems that follow optimized procedures.

Therefore, CPSs must be able to *act* autonomously, influencing themselves and the environment in which they are deployed using actuators, that enable decision taken by software components to reflect on the physical domain. The attitude control on the satellite from our case study uses reaction wheels to act on the position and orientation of the vessel, causing changes in the values of various environmental variables depending on them such as the magnetic field. Another, more subtle but equally important way for CPSs to interact with their environment is by exchanging information with other systems and with operators.

This monitor-analyze-act cycle creates a system in which the software and its physical environment are deeply intertwined and subjected to feedback loop dynamics that may be very difficult to fully comprehend. Moreover, to analyze CPS systems, engineers need to rely on very different formalisms to describe the software and the physical components. Software components are usually executed with a certain frequency and their behavior is regulated by a clock and captured by discrete sequences of events, such as the one produced by finite state machines. Differently, physical variables are subjected to a continuous evolution, which is better captured by mathematical formalisms like differential equations. CPSs, combining both continuous and discrete aspects into their structure, exhibit hybrid dynamics. This adds another layer of difficulty to the challenge of predicting and verifying the correctness of their behaviour. However, cyber-physical systems are often deployed in situations where safety and reliability are critical which makes developing techniques to enable the rigorous analysis of these types of systems of paramount importance.

This thesis was conducted in collaboration with the Interdisciplinary Centre for Security, Reliability, and Trust [56] (SnT) of the University of Luxembourg [55] and LuxSpace [7], a European space systems contractor based in Luxembourg. Specifically, this thesis was developed within the context of the H2020-EU project "Testing the Untestable: Model Testing of Complex Software-Intensive Systems" [3] (TUNE). The TUNE project aims at developing Verification and Validation (V&V) techniques for software-intensive systems. Specifically, it aims at proposing novel test solutions for systems that are untestable [29], meaning that traditional testing methods are highly expensive, time-consuming, or infeasible. The main goal of TUNE is to enable, propose, and develop scalable, solutions for test automation. The final goal is shifting towards well-defined engineering approaches with the intent of bringing early and cost-effective automation to the testing of many critical systems.

TUNE aims at supporting a large spectrum of testing activities. It includes the development of schedulability analysis for real-time systems [54], procedures to automatically derive the assumptions under which software components work properly [44], approaches to enable an efficient model-based simulation of satellite systems [69], and solutions to reduce the manual effort required by automating the generation of test cases from requirements specifications [71]. This thesis was conducted within the project "Trace-Checking CPS Properties".

1.2 Research Problem

Engineers collect traces (i.e., logs) describing the behavior of a CPS both when the CPS is simulated and, using instrumentation and logging mechanisms, also during the actual execution of its tasks. A trace is a sequence of records that contain some information about the execution (or the simulation) of the various components of the system (e.g., the state of the system variables). Trace records are usually labeled with time-stamps representing the time instants at which the recorded information was obtained. These traces are analyzed to check whether they conform to the system's requirements specifications; this activity can be automated by employing trace-checking tools. Specification-driven trace-checking tools usually take as input a trace to be analyzed and a requirement specification; they yield a boolean verdict indicating whether the trace satisfies the specification. The algorithms implemented by trace-checking tools are typically language-specific.

Problem 1.

In the context of trace checking, there exist two main categories of languages used for specifying CPS requirements: *time-based* and *sequence-based* languages. However, CPSs asks for more expressive languages able to express properties that are related both to the cyber and the physical components.

Time-based languages used within run time-verification frameworks (e.g., Signal Temporal Logic [57], Restricted Signals First-Order Logic [58], Signal First Order Logic [17], and SB-TemPsy-DSL [27]) interpret the records of the cyber and physical components as signals over a *time domain*. Specifications, written in a time-based language, express time relations over the occurrence of events. Such languages are suitable to express CPS requirements related to physical quantities; an example of such requirement is P1: "between 2 s and 10 s (measured starting from the origin of the trace) the speed of the satellite is lower than 10 m/s". However, (usually) time-based languages are not *easily* amenable to specifying requirements related to software components by users with a limited background on temporal logic. As an example, let us consider the requirement P2: "*whenever the satellite changes its mode from safe to normal*, the speed of the satellite decreases". To express the first part of this requirement (marked in italics), one should specify that 1) in the trace there are two *consecutive* records; 2) the first record captures that the satellite is in "safe mode"; and 3) the second record captures that the satellite is in "normal mode". This requirement cannot be easily expressed in time-based languages since they cannot specify the first condition, i.e., that a record immediately follows another one in the trace. Indeed, expressing such a condition requires the specification language to provide access to the indices (i.e., positions in the trace) of the different records.

On the other hand, in sequence-based languages—such as Linear Temporal Logic [37] (and domain-specific languages based on one of its extensions, like the one used for the SpeAR tool [42], FRETISH [46], and CoCoSpec [30])—

traces are sequences of consecutive records, whose temporal model is represented by the sequence of *discrete indices* of the records. This class of languages interprets the records of the CPS software and physical components as discrete-time signals. Specifications in these languages constrain the indices in which events can occur; such specifications are used to express properties that mostly refer to the CPS software components, such as the first part of the aforementioned P2 property. However, these languages cannot express time relations over the occurrence of events, such as "the satellite angular rate shall reach a value lower than $1.5\,^{\circ}/\mathrm{s}$ within $10\,\mathrm{s}$".

A third class of specification languages is *hybrid* languages (e.g., STL-MX [41], HyLTL [28], HRELTL [31], Differential Dynamic Logic [63], Hybrid Temporal Logic [52]), which supports the specification of both continuous and discrete behaviors. However, these languages typically extend existing languages (e.g., LTL) to support the specification of hybrid behaviors in specific contexts (e.g., using signal derivatives). Therefore, they provide ad-hoc solutions that inherit some of the intrinsic limitations of the base language, thus hindering the expressiveness of the resulting hybrid language. For example, a hybrid language based on LTL cannot support metric operators to constrain the time distance between events.

Since Cyber-physical systems combine cyber and physical characteristics [64], trace-checking tools should support languages that allow engineers to express properties that refer to both the cyber and the physical components.

Problem 2.

We define a trace as the recording of a set of variables that describe the behaviour of a system, or the model of said system, during a single run. In the case of complex Cyber-Physical Systems, it could mean following the evolution of hundreds of variables representing both software states and physical quantities, sometimes for extended periods. Therefore, manually checking if a trace representing the behaviour of the system under observation satisfies a requirement of interest would be an extremely time-consuming endeavour, and prone to many errors.

Automated trace-checking procedures are developed to reduce the time and effort requested for this kind of procedure while increasing their reliability. To perform a trace-checking procedure, once the requirement has been articulated in a sufficiently expressive specification language and a trace of the system has been obtained, they need to be checked against each other to determine if the trace respects the requirement through a method that ideally must be as efficient and as robust as possible to be of practical use.

Typically, there is a trade-off between the expressiveness of the specification language and the efficiency of the trace-checking procedure; consequently, there is a delicate balance to strike between the expressive power of the specification language and the immediacy of the response, and this is one of the main challenges to face when designing a trace-checking approach for industrial use.

Since their purpose is reducing the time and effort required to analyze the behavior of complex CPS, trace-checking tools should support automated procedures that guarantee reliable results without being excessively resource-intensive and time-consuming.

1.3 Contribution of the Thesis

The contribution of this work is an automated trace-checking tool for CPS. This thesis solves the problems identified in Section 1.2 as follows.

- We present the **Hybrid Logic of Signals** (HLS). HLS is a new specification language tailored to specifying CPS requirements. HLS allows engineers to express CPS requirements as properties (i.e., specifications) referring both to the time-stamps and the indices of the records of CPS traces. In this way, HLS specifications can easily express the behavior of both cyber and physical components, as well as their interactions, exploiting the indices to express changes of state and the timestamps for describing the continuous aspects of the system.

- We present **ThEodorE (Logic-based TracE checkEr for HLS)**, an efficient trace-checking approach for properties expressed in HLS. ThEodorE reduces the problem of checking an HLS property on a trace to a satisfiability problem, which can be solved using off-the-shelf Satisfiability Modulo Theories (SMT) solvers. The latter have efficient decision procedures for several background theories, thus making it possible to check whether the requirements expressed over the execution traces are satisfiable in the light of sound and time-proven mathematical and logical paradigms.

The thesis was evaluated using an industrial case study in the satellite domain, in collaboration with the engineers who developed the satellite's on-board system. We evaluated the support provided by the ThEodorE trace-checker by assessing its applicability on 20 large traces provided by LuxSpace and obtained by simulating the behaviour of the satellite across different scenarios, representative of its working conditions.

- We assessed the *expressiveness* of HLS by checking whether it could express the 212 requirements of our case study that were directly derived from LuxSpace's technical documentation. Our results show that HLS could fully express all these requirements. We also compared HLS with SB-TemPsy-DSL [27] and STL [57], two specification languages proposed in the literature, and for which trace-checking tools are available. The results show that HLS is significantly more expressive than SB-TemPsy-DSL and STL, which could only express 145 and 102 requirements, respectively.
- We evaluated the trace-checking support provided by ThEodorE by assessing its *applicability* on 20 large traces provided by our industrial partner, LuxSpace, and obtained by simulating the behavior of the satellite across representative, different scenarios. We ran the ThEodorE trace-checker on 747 trace- requirement combinations. The ThEodorE trace-checker completed the verification in 74.5% of the cases within one hour, a reasonable time-out considering typical CPS development contexts. ThEodorE yielded a verdict for 67.9% of the 337 trace-requirement combinations containing a requirement that cannot be verified by any of the other trace-checkers. We compared the applicability of ThEodorE with SB-TemPsy-Check [27] and Breach [35], for the trace-requirement combinations containing requirements expressible in SB-TemPsy-DSL and STL. For these combinations, SB-TemPsy-Check and Breach were 21.9% and 4.9% more often applicable than ThEodorE, respectively. SB- TemPsy-Check and Breach were also more efficient, but not to a point where it had practical implications.

Our results show that ThEodorE is broadly applicable as it allows engineers to specify a large variety of requirements while providing an efficient trace-checking procedure.

1.4 Structure of the Thesis

This thesis is organized as follows:

- **Chapter 2: Background.** It describes background concepts and notations necessary to understand this work. We provide a high-level description of specification-driven trace-checking. We introduced Xtext, the tool used to implement the specification language of ThEodorE and describe some of its features. We present Satisfiability Modulo Theory (SMT), a decision problem for logical formulas that allow considering theories including classical first-order logic.

- **Chapter 3: State of the Art.** It reports on related work. It presents 1. Alternative languages that allow engineers to specify the requirements of interest. We discuss how these languages express the properties of CPS, and why we decided to introduce a new language. 2. Alternative trace-checking tools that allow engineers to check whether a trace is compliant with its requirements. We discuss the design of alternative trace-checking tools proposed in the literature. We discuss how these tools solved the trace-checking problem.
- **Chapter 4: Case Study.** It presents our case study: a maritime satellite designed to collect tracking information from vessels operating on Earth and to relay those data to the ground. We show a fragment of an execution trace from our case study and an exemplar requirement. Finally, we show the limitations of current languages and motivate the need for an expressive language for specifying hybrid behaviors of CPSs.
- **Chapter 5: Hybrid Logic of Signals.** It first introduces a discussion on the design goals of the language. Then, the mathematical model of the traces considered in this work is defined. Finally, it presents the syntax and the semantics of the Hybrid Logic of Signals, the language we introduced.
- **Chapter 6: Theodore.** It introduces ThEodorE, a trace-checker for HLS that will be described and analyzed from a theoretical point of view. We show how ThEodorE reduces the problem of checking an HLS property on a trace to a satisfiability problem, which can be solved using off-the-shelf SMT solvers.
- **Chapter 7: Implementation.** It provides a general overview of the main components of ThEodorE. It describes how ThEodorE's grammar was written using Eclipse Xtext and how a code generator was implemented to perform the translation of the traces and their relative properties to Satisfiability Modulo Theory using Xtend, a Java dialect developed to be used in tandem with Xtext.
- **Chapter 8: Evaluation.** It reports on the evaluation of our contributions. It evaluates the expressiveness of HLS, and compare it with

state-of-the-art specification languages. Additionally, it evaluates the applicability of the ThEodorE trace checker, and compare it to state-of-the-art tools.

- **Chapter 9: Conclusions.** Summarizes our theoretical contributions and our practical findings. We draw conclusions and briefly touch on future works.

Chapter 2

Background

This chapter aims to clarify and describe the concepts and tools necessary for the comprehension of this thesis. First, we define the trace-checking problem (Section 2.1). Then (Section 2.2), we give an overview of Xtext, the tool used for developing ThEodorE. Finally, we describe Satisfiability Modulo Theories and their role in the verification process performed by ThEodorEs (Section 2.3).

2.1 Trace-Checking

Ensuring that a system behaves as expected in different environmental conditionss is critical for Cyber-Physical Systems, which often tend to operate autonomously, without constant human supervision, even on critical tasks. However, due to the complexity of most of the systems in question and the unpredictability of their surroundings, it is not reasonable nor feasible to manually check the compliancy of every single part of them to its expected standards, leading to the introduction of multiple kinds of automated analysis techniques

For the same reasons, even when using automated tools, it is nearly impossible to explore all the possible scenarios, since they are, for all intents and purposes, practically infinite. A common solution is to use lightweight techniques to explore a finite set of use cases that provides good coverage on the expected working conditions to which the system will be subjected.

Trace-checking is one of these techniques, belonging to the field of Runtime Verification [21]. Runtime Verification is a verification approach that consists of obtaining information from a system executing its task, extrapolating its behaviour, and checking if it satisfies or violates certain properties, expressed through some kind of formal specifications.

It can be applied to both software systems and hardware systems, as long as an execution trace is available, and more recently it has also been applied to CPS and hybrid systems in general [20, 27, 58, 35, 61, 16]. It can also be applied to the model of a given system, for example, a Simulink® file, which is very useful during model-based design processes.

A trace is an ordered sequence of records that contains information about the state of the system during a period of activity. In the case of CPSs, it is often composed of multiple records following the evolution of different variables during the execution time. These variables may belong to a physical part of the system or to a software component and their behaviour can be vastly different even in the context of the same system. Some of them may present discontinuous changes of state and others may be characterized by continuous evolution, depending on the nature of the component from which they originated, being it cybernetic or physical.

In property-driven trace checking, the formal requirements to verify over the traces can be expressed in different ways, such as regular expressions, state machines, and, most commonly for CPSs and hybrid systems, logical propositions.

Most of the logical languages employed for this kind of formalizations have their roots in temporal logic [21]. Linear Temporal Logic, for example, was one of the first logics adopted for formal verification of software systems, in 1977 [66]. Some of its extensions are specifically targeted at CPSs and hybrid systems (see Chapter 3)

Properties and traces are then interpreted, put together, and compared by software tools with a variety of methods to produce boolean verdicts over the satisfaction of the requisites under scrutiny.

While it does not produce a comprehensive analysis of the system and all its failure states, trace checking can be a valuable tool to verify if critical requirements are respected in a range of plausible scenarios and to identify the causes of failure states by analyzing which properties were violated and which variables did not conform to the expected behavior.

2.2 Xtext

The HLS specification language is the core of ThEodorE, our trace-checking tool, where was used as a base for a domain-specific language focused on expressing logical requirements and verifying their satisfiability over the traces produced by Cyber-Physical Systems.

A Domain-Specific Language (DSL) is a programming language dedicated to a particular application, in opposition to a General Purpose Language, which can be used across multiple domains. The advantage of a DSL is often its ease of use; these languages are accessible also to users that do not have extensive programming skills because they are very lean and focused, with a smaller number of keywords and functions.

Xtext [11] is a framework for developing programming languages and domain-specific languages. It is an open-source software maintained as part of the Eclipse Project [43], specifically of the Eclipse Modeling Framework [48]. It is designed for easy integration with the Eclipse Integrated Development Environment.

The first step in defining a DSL with Xtext is writing a grammar file with the dedicated grammar language. The Xtext grammar language is itself a DSL [12], designed to describe textual languages. It expresses the syntax of the language and how it will be mapped to the semantic model by the parser. Listing 2.1 is an example of a simple Xtext grammar file: the various components of the language are described through the use of keywords and Backus-Naur form [45] expressions.

The keywords of the language are simple strings defined by the DSL creator so that a parser will use them as a guide to read and separate the code.

A parser is a software component that takes as input a string of text and then builds a data structure, such as an Abstract Syntax Tree. Xtext automatically generates a parser for the DSL under development using the rules expressed in the grammar.

Listing 2.1 provides an example grammar specified using Xtext functions.

- Lines 2-3: The rule *StudentBody* specifies that *StudentBody* can contain an arbitrary number of instances of the object *Student* stored in the *students* feature.

- Lines 6-7: The rule *Students* specifies that a *Students* can be either a *BachelorStudent* or a *MasterStudent*.

- Lines 10-11: *BachelorStudent* is defined, with keyword *'bachelor'* followed by the feature *name*, which take as input the terminal rule *ID*.
- Lines 14-15: *MasterStudent* is defined much in the same way as *BachelorStudent*, albeit with a different keyword.

Listing 2.1: A simple example of Xtext grammar

```
 1
 2 StudentBody:
 3     (students+=Student)*;
 4
 5
 6 Student:
 7     BachelorStudent | MasterStudent;
 8
 9
10 BachelorStudent:
11     'bachelor' name=ID;
12
13
14 MasterStudent:
15     'master' name=ID;
```

The terminal rule ID is defined this way with regular expressions in the Xtext documentation 2.2:

Listing 2.2: The terminal rule ID

```
terminal ID:
    ('^')?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_
        '|'0'..'9')*;
```

It says that a token ID starts with an optional "^" character, followed by a letter or underscore followed by any number of letters, underscores, and numbers.

Based on this grammar, the user can defined the code in Listing 2.3 defines:

1. a *BachelorStudent* with the feature *name* equal to *JonhSmith*
2. a *MasterStudent* with the feature *name* equal to *JaneDoe*

The objects are stored in the *students* feature of *StudentBody*, to be used by an interpreter or, as in ThEodorE's case, a code generator.

Listing 2.3: An example of code to interpret

```
bachelor JonhSmith
master JaneDoe
```

Xtext has the feature to easily integrate a code generator thanks to Xtend [10], a Java dialect designed for this purpose. ThEodorE code generator is actually a translator from HLs to the input language of Z3 Prover [33], a solver for Satisfiability Modulo Theories formulas.

2.3 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) is a deduction framework for checking the satisfiability of first-order logic formulas. The satisfiability problem consists of determining whether a logical formula admits a solution, also called a model, that is compatible with its constraints.

In an SMT problem, the logical symbols that make up the structure of the formulas are interpreted in the light of different background theories restricting their meaning. Such theories represent logical formalizations of a variety of different topics, from linear arithmetics to arrays and data structures, amongst others. In essence, a theory is a set of logical sentences that constraint the interpretation of symbols in a formula.

Given a formula ϕ and a theory T, if $T \cup \{\phi\}$ is satisfiable, then ϕ is *satisfiable modulo T* , which means that is satisfiable in the context of the underlying theory T.

For example, we can interpret the $a + b = b + a$ equality as true in light of the theory of linear arithmetics. Multiple theories can be applied to the interpretation of a single formula if required by the content of the latter; integrating different theories to make them work together is, in fact, one of the challenges when designing automated solutions for this kind of problem.

SMT-LIB [2] is an initiative that promotes and maintains a library of SMT background theories and their description, in addition to developing and promoting the standard input/output language for SMT solvers, the SMT-LIB Standard [18]. Many such solvers exist to check SMT problems and there is even an annual competition SMT-COMP [1] promoting their development.

ThEodorE was designed to reduce the problem of checking logical properties over CPS traces to an SMT problem. This approach has the merit of being able to choose amongst different SMT compatible solvers and take advantage of the continuous improvement in the field. As mentioned in Section 2.2, the solver chosen for this thesis is Z3 Theorem Prover [33], which was developed by Microsoft Research.

Chapter 3

State of the Art

In this chapter, we will analyze the contributions relative to the state of the art and highlight the similarities and differences with respect to our work. First, we present alternative specification languages that can be used to express properties of CPSs (Section 3.1). Then, we describe alternative trace-checking tools and compare them with ThEodorE (Section 3.2).

3.1 Specification Languages

A significant amount of work has been done by the software engineering and formal methods communities to develop logic-based languages for supporting the specification of the properties of systems. This section provides an overview of existing specification languages and summarizes their benefits and limitations.

Time-based Languages

Our work is focused on languages that allow specifying how signals should change over time. Signal-based logic languages are a subset of time-based languages proposed in the literature to specify properties on signal behaviors that change over time.

- *Signal Temporal Logic* (STL) [57]. STL represents signals as functions from real time points to values. This logic admits predicates over the signals' values, for example $\mathcal{G}_{[0,10]}(s_1 > 0)$, specifies that within the time interval $[0, 10]$ "globally", the value of the signal s_1 is grater than ten. Since the properties are defined over dense time, STL lacks the "next" temporal operator.

- *Mixed Time Signal Temporal Logic* (STL-MX) [41] extends STL to define properties both on discrete-time and on dense time. The language includes two layers, one based on LTL to express properties of discrete-time Boolean signals (sampled at a fixed sample rate), and another one based on STL, to express properties on dense-time real-valued signals. Time mapping operators define the conversion between dense-time and discrete-time signals and formulae. Compared to HLS, STL-MX restricts discrete-time Boolean signals to be sampled at a fixed sample rate and lacks first-order quantifiers on variable values.

- *Restricted Signals First-Order Logic* (RFOL) [58] is an extension of STL that enables to use of absolute bounds and arithmetic operators to define the boundaries of the intervals of the temporal operators. Similar to STL, it lacks the "next" temporal operator.

- *Signal First Order logic* (SFO) [17] is an extension of first order logic with continuous signal variables. Similarly to STL and RFOL, it represents signals as functions from real time points to values. Therefore, it does not allow to use universal and existential quantifiers on index variables, and can not express the "next" temporal operator.

- *SB-TemPsy-DSL* [27] is a pattern-based logic language that allows the specification of signal-based temporal properties covering the most frequent requirement types in CPS domains. It was defined in collaboration with engineers from the CPS field and supports the specification of recurrent and common types of signal-based temporal properties, identified in a taxonomy [26].

There are many other time-based languages in the literature. However, these logics are not explicitly tailored for specifying signal-based properties. A partial listing of would include *Metric Temporal Logic* (MTL) [53], an extension of temporal logic that integrates the classical *until*, *next*, *since* and *previous* temporal operators with time-constraints, and *SOLOIST* [24] a specification language for formalizing the interactions of service compositions.

Sequence-based Languages

In sequence-based languages traces are represent as sequences of consecutive records, whose temporal model is based on the sequence of *discrete indices* of the records. This class of languages interprets the records of the CPS software and physical components as discrete-time signals. A subset of sequence-based languages related to our work is reported in the following:

- *Linear Temporal Logic* (LTL) [37] is a modal temporal logic with modalities referring to time. LTL enables users to specify properties using the temporal operators (e.g., "globally", "eventually", "release", "until", "release"). However, this logic does not possess an "explicit notion" of time that can be used to specify the properties of interest. For example, it is not possible to specify that within the time interval $[0, 10]$ the value of the signal s_1 is greater than ten.

- *SpeAR* [42] is a tool that allows capturing and analyzing requirements in a domain-specific language designed to read like natural language. The language is built on the top of LTL, and its formal semantics is specified using Past LTL.

- FRETISH [46] is a structured natural language for specifying properties of Cyber-Physical Systems. FRETISH provides different constructs, such as scope, condition, timing, and response operators. The semantics of FRETISH is defined by relying on its equivalence with LTL.

- *CoCoSpec* [30] is an assume-guarantee-based contract language for embedded systems. It is an extension of the Lustre [49] language. It enables the specification of properties using past linear temporal logic (pLTL).

- Pattern bAsed Mission specifier (PsALM) [60, 59] is a pattern-based domain-specific language tailored for the specification of robotic missions. Similarly to other languages, it is based on LTL.

Hybrid Languages

Different works have tackled the challenge to express hybrid properties by extending the LTL logic, which presents a discrete concept of time, to be able to handle continuous variables and signals. Some of them are also supported by trace checking procedures and tools.

- *HyLTL* [28], extends LTL to with flow constraints. Flow constraints are conditions specified by us inequalities between arithmetic expressions among the value of the variables of the system. They enable the users to define sets of valuations, sets of trajectories, and jump relations. A property expressed in HyLTL can be converted to a hybrid automaton and then verified on the trace using a composition between the translated property (negated) and the automaton representing the system.

- *RELTL* [31], extends LTL with regular expressions. It expresses constraints on discrete and continuous behaviors by expressing constraints over, respectively, instantaneous transitions and derivatives. Being based on LTL, it does not support metric operators to constrain the time distance between events. Moreover, using the derivatives of the variables to specify hybrid behavior limits HRELTL's usefulness to specific contexts, and only a subset of HRELTL is the object of automated verification procedures.

- *Differential Dynamic Logic* [63] uses differential expression to represent continuous evolutions. It differs from HLS since it is designed for specifying properties of systems expressed using the hybrid system [15] modeling formalism. As such, its modal operators enable references to the states that are reachable after firing the transitions of the hybrid system model. An automated tool [65]has been proposed as a model checking theorem prover for hybrid systems based on DDL, but it requires extensive knowledge of the model of the system. HLS and ThEodorE, on the contrary, are optimized for trace-checking, a different procedure that can be applied both on data produced by a model and on data derived from the real system, with no additional knowledge needed save from the requirement to verify.

- *Constraint LTL over clocks (CLTLoc [23])* is a *quantifier-free* extension of LTL allowing variables behaving like clocks over real numbers. Clocks on real numbers can be used to represent, time-based properties. However, CLTLoc does not allow users to quantify on real-value variables.

- The SCR requirements model [51] (i.e., special tables that encode the desired behavior of the system) was extended [50] to support hybrid systems. It considers continuous variables and the specification of their timing and accuracy requirements, by adding some timing information to the events encoded within the SCR tables. As such, it cannot express most of the properties in our case study. Furthermore, the language is not supported by any trace-checking tool.

- Lola [32, 38] is a rule and stream-based specification language. It enables the specification of properties that specify correct behavior, and properties that specify statistical measures that allow profiling the system that produces the input streams. Striver [47] is another general language that allows to express other real-time monitoring languages.

3.2 Trace-checking Tools

In recent years, with the explosion in the complexity of the systems and therefore in the quantity of data to analyze, the need for automated trace-checking procedures has arisen. A large portion of these algorithms and tools is based on expression languages such as HLS. In this section, we present a subset of automated trace-checking tools developed for systems verification. We referred to recent surveys [67, 20], for a complete description of existing tools. In the following, we will briefly go over them.

- *AMT 2.0* [61] is a tool for the analysis of hybrid signals with both continuous and Boolean components, combining discrete events with numerical values. The specifications are expressed in extended Signal Temporal Logic (xSTL) an integration between Signal Temporal Logic and Timed Regular Expressions.

- *S-TaLiRo* [16] is a trace-checking tool for Metric Temporal Logic (MTL) properties, designed mainly to automatically produce inputs for and analyze the output of non-linear hybrid systems modeled in Simulink/Stateflow (TM). The output traces of the model are ranked with a robustness-based metric; a negative robustness score means that the system trace falsified the property.

- The approach of reducing the trace-checking problem to the verification of the satisfiability of a logical formula has been also used on *SOLOIST* [25], which is an extension of Metric Temporal Logic, tailored for service-based applications, through a field study on specification patterns in the context of that field.

- *SOCRaTes* [58] is an automated generator of online test oracles for CPS Simulink models with or without uncertainties. The oracles are derived from requirements declared in Restricted Signals First-Order Logic (RFOL), a fragment of Signal First Order logic, and are initialized as Simulink$^{®}$ blocks, that can determine if the model has passed or failed a given test. Although SOCRaTes is targeted at the same class of system as ThEodorE, it is geared more towards online monitoring of models than to trace-checking. This means that its specification language, RFOL, is constrained to be less expressive than HLS due to efficiency reasons.

- StreamLAB [39] is a monitoring framework for RTLola specifications. StreamLAB performs a static analysis of the specification. It identifies parts of the specification with unbounded memory consumption, and compute bounds for all other parts of the specification. It also includes the computes the worst-case memory consumption of the specification.

- Striver [47] is another tool that supports run-time verification when observations are described as output streams of data computed from input streams of data.
- MonPoly [68, 22] is a monitoring tool for metric first-order temporal logic (MFOTL). MFOTL can express complex dependencies between data values coming from different events in the stream.

Out of all the tools we considered, we choose two, based on their significant similarities with ThEodorE and their availability, to provide a benchmark for its performances (Chapter 8). These approaches are described below:

- *SB-TemPsy-Check* [27] is a tool designed to verify *SB-TemPsy-DSL*, designed to express Signal-based temporal properties (SBTPs), expressed with SBTPs are commonly used to characterize the behavior of a system with input and output interpretable as signals over time; amongst many DSL capable of expressing this kind of properties, SB-TemPsy distinguishes itself for supporting the specification of important types of properties such as spikes or oscillations. The trace checking procedure of *SB-TemPsy-Check* reduces the problem of checking an SBTP over an execution trace to the problem of evaluating an Object Constraint Language constraint on a model of the execution trace. SB-TemPsy was one of the tools selected for evaluating the expressiveness and efficiency of HLS and ThEodorE and while it represented a more efficient solution, SB-TemPsy-DSL proved itself to be significantly less expressive than HLS.
- BREACH [34] is a Matlab/C++ toolbox designed for the analysis of models, which was recently extended [35] with an algorithm that assigns robustness values to the satisfaction or violation of an STL formula by an execution trace. Based on the same language as *S-TaLiRo* [16], but more efficient, BREACH was one of the tools selected as benchmark for ThEodorE's applicability.

Chapter 4

Case Study

LuxSpace [7] developed, in collaboration with a large space agency, a maritime satellite to collect tracking information from vessels operating on Earth and to relay those data to the ground. This is a representative CPS made of complex software component interacting with many actuators and sensors and the physical environment where the satellite is to be deployed. This system should satisfy many varied requirements regarding the behavior of the software system itself but also its interactions with hardware and the satellite physical dynamics in space. Its development relies on technologies and practices typically seen in CPS contexts, e.g., Model-in-the-loop development with Simulink®.

Software engineers check the compliance of the satellite behavior to its requirements [9] both while the software is being developed and at run time. This is done by

- collecting execution traces of the system, and
- checking whether those traces satisfy the system requirements.

This kind of work can be extremely time-consuming if done manually, due to the vast amount of data involved, hence the need of automating the trace-checking procedure.

Figure 4.1 shows a fragment of an execution trace, which we will use to motivate this work. A trace is a sequence of records that contain some information about the execution of the system. In this example, the records include data about the *angular rate* (`ang-rate`) and the *(satellite) mode* (`mode`). The angular rate is a physical quantity represented by a real value measured by sensors; this record represents a continuous dynamic. The mode

`ang-rate`	20.1	22.2	23.3	20.4	21.1	3.2	1.1
`mode`	0	1	0	0	3	3	3
timestamp	0	0.2	0.9	1.8	3.0	4.9	5.7
index	0	1	2	3	4	5	6

Record r_3

Figure 4.1: A fragment of an execution trace of our case study.

is an enumeration of values that represent the state of the satellite software: their evolution in time is stricly discreet. There are four different modes: "Idle Mode", "Safe Spin Mode", "Normal Mode Coarse", and "Normal Mode Fine", which are represented in the trace by the values 0, 1, 2, and 3, respectively. In addition, each record is associated with a timestamp, representing the time instant at which the recorded information was obtained, and with a progressive index value.

The requirements to be checked on the system traces refer both to the software and to the physical dynamics of the satellite. For example, let us consider requirement $\mathcal{R}1$:

> Whenever **the satellite mode switches from "Idle Mode" to "Normal Mode Fine"**, the satellite angular rate shall reach a value lower than 1.5 °/s **within 10 s**. Moreover, the angular rate shall stabilize around **an arbitrary value** c lower than or equal to 1.5 °/s.

$\mathcal{R}1$ specifies a constraint on a physical quantity, i.e., the angular rate of the satellite, which shall be ensured as a reaction to a software change, i.e., the satellite switching its mode from "Idle" to "Normal Mode Fine". This is a perfect example of how the software states and physical dynamics of the satellite frequently influence one another, and must be taken both into account when trying to correctly describe this kind of systems.

One way to express that ***the mode of the satellite switches from "Idle Mode" to "Normal Mode Fine"***, is to specify that the trace contains:

1. two records with *consecutive* indices;
2. the first record captures that the satellite is in "Idle Mode";
3. the second record captures that the satellite is in "Normal Mode Fine".

This requirement cannot be expressed using time-based languages since they do not provide access to the indices of the different records. To compensate for this limitation when using time-based languages, engineers can apply ad-hoc solutions, such as adding a new Boolean flag to the trace records. In our example, such a flag would be true whenever the mode of the satellite switches from “Idle Mode” to “Normal Mode Fine”. In this way, the aforementioned requirement fragment would be rephrased as ***the flag*** *`switch-from-IDLE-to-NORMAL-MODE-FINE`* ***is true***. However, this is impractical in real scenarios because:

1. The number of flags to add in the trace records can quickly grow and become unmanageable. For example, given the four possible values for the satellite mode in our case study, to consider all possible combinations for switching satellite mode, engineers would need to add 16 values in each record (one for each mode switching combination).

2. The requirement is reformulated and its connection to the actual software component behavior is lost, making the expression of the properties less clear and the results harder to interpret.

Furthermore, requirement $\mathcal{R}1$ cannot be expressed using sequence-based languages because they do not support time relations over the occurrence of events. More specifically, expressing that “the [...] angular rate shall reach [...] **within 10 s**” requires to access the timestamps associated with the trace records (and compute a distance). This feature is not provided by sequence-based languages.

Moreover, to the best of our our knowledge, among the time-based and sequence-based languages mentioned in the previous chapters, SFO [17] is the only language that allows users to use quantified variables in specifications, (as in “(there exist) **an arbitrary value** c lower than or equal to 1.5 °/s around which [...] shall stabilize”. This type of requirements is extremely common in practical CPS applications, since engineers often want to check that the system stabilizes around a given value (e.g., the steady-state value). Although engineers know some properties of the steady-state value c (i.e., c shall be lower than or equal to 1.5 °/s), they generally do not know its exact value, which has to be indicated as a generic variable in the requirement specification. This example, extracted from our case study, shows the need for an expressive language for specifying hybrid behaviors of CPSs. In the next chapter, we will introduce a new specification language for CPSs, which overcomes the limitations—in terms of expressiveness—of state-of-the-art languages and is supported by an effective trace-checking procedure.

Chapter 5

Hybrid Logic of Signals

In this chapter, we indroduce the *Hybrid Logic of Signals* (HLS), a new specification language for CPS. We first explain the design goals of the language (section 5.1). Then, we describe the mathematical model of the traces considered in this work (section 5.2). Finally, we explain the syntax (section 5.3) and the semantics (section 5.4) of the language, and go over the grammar for some additional contructs (section 5.5).

5.1 Design goals

HLS was conceived as a language for specifying CPS properties in a way that would seamlessly merge the features of sequence-based and time-based languages, to express with accuracy the hybrid behaviour of CPS. For this reason, HLS extends existing time-based languages (e.g., STL [57], MTL [53], RFOL [58], and SFO [17]) and sequence-based languages (e.g., LTL [37], FRETISH [46], and CoCoSpec [30]) to allow *to refer both to trace indices and to timestamps* in the logical specifications and *to arbitrarily combine them* to define properties describing the behavior of a CPS. More specifically, HLS allows engineers to use first-order *existential* and *universal* quantifiers with:

- *timestamp variables*, to declare properties referring to specific time instants and to the interval between them, such as "*there exists a time instant t within* 10 s *from the current time instant [...]*";
- *(trace) index variables*, to declare properties that refer to the array-like indices of trace records, such as "*for every trace index i, such that the corresponding record captures that the satellite is in "Idle Mode", and the immediately following record (at trace index $i+1$) captures that the*

satellite is in "Normal Mode Fine" [...]";

- *real-valued variables*, to declare properties that refer to arbitrary signal values, such as "*there exists a value c lower than or equal to* 1.5 °/s *around which the signal* ang-rate *shall stabilize*".

Moreover, HLS supports specifications including:

- the value of a signal *at a certain timestamp* or associated with a record *at a certain index*;
- the timestamp associated with the record *at a certain index*;
- the index of the record *with a certain timestamp*;
- expressions including and combining time variables, trace indices, and real-valued variables, using arithmetic and relational operators.

5.2 Traces

Let $\mathbb{J} = \{0, 1, 2, \ldots, j, \ldots, m\}$, with elements $j \in \mathbb{N}$, be a set of indices. Let $\mathbb{T}$ be an interval of $\mathbb{R}$; $\mathbb{T}$ is then defined as a time domain. Let $S = \{s_1, s_2, \ldots, s_i, \ldots, s_n\}$ be a set of variables (hereafter called "signals") of the systems being monitored, with $s_i \in \mathbb{R}$. A trace π is a finite sequence of records $r_0, r_1, \ldots, r_j, \ldots, r_m$, with $j \in \mathbb{J}$.

Each record r_j is a tuple $\langle j, t, v_1, v_2, \ldots, v_n \rangle$, where $j \in \mathbb{J}$ is the index linked with the record, $t \in \mathbb{T}$ is the timestamp at which the recorded information was obtained, and $v_1, v_2, \ldots, v_n \in \mathbb{R}$ are the values relative to signals $s_1, s_2, \ldots, s_n$ in the record. For a trace π the array notation "$[j]$" stands for the j-th record of π, and we use the dot notation to denote an element of a record; we also introduce the notation t_j, short for $\pi[j].t$ for a given trace π. For example, let π^e be the fragment of the trace represented in Figure 4.1; it is composed of seven records. Record r_3 is denoted by $\pi^e[3]$; it is defined by the tuple $\langle 3, 1.8, 0, 20.4 \rangle$, where $\pi^e[3].t = t_3 = 1.8$ is the value of the timestamp, $\pi^e[3].\texttt{mode} = 0$ is the value of signal `mode`, and $\pi^e[3].\texttt{ang-rate} = 20.4$ is the value of signal `ang-rate`.

The values of the timestamps are monotonically increasing, i.e., $t_j < t_{j+1}$, since records refer to consecutive timestamps. We say that a trace has a *fixed sample rate sr* if, for every $j, 0 \leq j < m$, $t_{j+1} - t_j = sr$, where sr is a constant value; otherwise, we say that the trace presents a *variable sample rate*. For example, trace π^e in Figure 4.1 has a variable sample rate.

Additionally, we define the function $\iota_\pi \colon \mathbb{T} \to \mathbb{J}$: given a timestamp value t, $\iota_\pi(t)$ is the value of the index j of the record in π with the highest timestamp t_j such that $t_j <= t$; the trace subscript will be omitted when it is clear from the context. For example, for trace π^e in Figure 4.1, $\iota_{\pi^e}(2.5) = 3$. In this work, we consider two definitions of ι:

$$\begin{aligned}\iota^V(t) ::= {} & [t_0 \leq t] \cdot [t < t_1] \cdot 0 + [t_1 \leq t] \cdot [t < t_2] \cdot 1 + \\ & \ldots + [t_{m-1} \leq t] \cdot [t < t_m] \cdot (m-1) + [t_m = t] \cdot m \\ \iota^F(t) ::= {} & \left\lfloor \frac{t}{sr} \right\rfloor\end{aligned}$$

Definition $\iota^V(t)$ assumes that the trace shows a variable sample rate. Notice that the notation $[P]$, where P is a logical predicate, is the Iverson bracket; it evaluates to 1 if P is true, and to 0 otherwise. The resulting arithmetic formula find where the timestamp t, provided in input, is situated w.r.t. the timestamps of the trace (i.e., $t_0, t_1, \ldots, t_m$), and returns the value of the index of the record that presents the highest timestamp smaller than or equal to t. For example, if the value of t is greater than timestamp t_2 and lower than timestamp t_3, the only expression in $\iota^V(t)$ that does not evaluate to 0 is $[t_2 \leq t] \cdot [t < t_3] \cdot 2$; therefore the index returned will be 2.

Definition $\iota^F(t)$ is valid when the trace has a fixed sample rate. In such a case, the index associated with a timestamp can be simply retrieved by computing the floor of the ratio of the timestamp t over the sample rate sr.

All the variables are expected to be sampled at each timestamp. This is a necessary requirement to permit the evaluation of the satisfaction of the system requirements at each timestamp. For systems that do not sample all the variables at each timestamp, pre-processing can be used to interpolate the values to assign to variables for which the value is missing at certain timestamps. In this work, we consider two parallel pre-processing strategies:

$\mathcal{A}1$: In each record, an interpolation function (e.g., piece-wise constant, linear, cubic) specific to each signal, is used to produce values for unassigned variables. Notice that this strategy does not alter the original sample rate of the trace, since it retains the same records as the original trace and only generates (in each record) values for the *unassigned* variables.

$\mathcal{A}2$: If the trace has a variable sample rate, it is transformed into a trace with a fixed sample rate. This is done by creating new records with a fixed sample rate equal to the smallest sample rate (i.e., the minimum

Term	$\texttt{tm} ::=$	$\texttt{tt} \mid \texttt{vt} \mid \texttt{it}$
Time Term	$\texttt{tt} ::=$	$\tau \mid t \mid \texttt{i2t}(\texttt{it}) \mid f(\texttt{tt}_1, \texttt{tt}_2)$
Index Term	$\texttt{it} ::=$	$\sigma \mid j \mid \texttt{t2i}(\texttt{tt}) \mid f(\texttt{it}_1, \texttt{it}_2)$
Value Term	$\texttt{vt} ::=$	$\rho \mid x \mid (s \ \texttt{@i} \ \texttt{it}) \mid (s \ \texttt{@t} \ \texttt{tt}) \mid f(\texttt{vt}_1, \texttt{vt}_2)$
Formula	$\texttt{p} ::=$	$\texttt{tm}_1 < \texttt{tm}_2 \mid \texttt{not p} \mid \texttt{p}_1 \ \texttt{or} \ \texttt{p}_2$
		$\mid \texttt{exists} \ \tau \ \texttt{in} \ I_T \ \texttt{such that p}$
		$\mid \texttt{exists} \ \sigma \ \texttt{in} \ I_J\texttt{such that p}$
		$\mid \texttt{exists} \ \rho \ \texttt{such that p}$

$t \in \mathbb{T}, j \in \mathbb{J}, x \in \mathbb{R}, \tau \in TV, \sigma \in SV, \rho \in RV, s \in S$

Figure 5.1: Syntax of the Hybrid Logic of Signals.

time interval between two records) of the initial trace, and by using the interpolation functions (as in the case of strategy $\mathcal{A}1$) to generate the values of *all* variables.

As we will address in Chapter 8, the strategy used to generate the values of unassigned variables decides the accuracy of the trace. The latter impacts the trace checking verdict and may affect the correctness of the trace-checking procedure.

5.3 Syntax

We define an HLS formula according to the grammar in Figure 5.1, whose start symbol is `p`. In the grammar, the symbol f is used to represent a generic (binary) arithmetic function; the symbol $\mid$ separates alternatives. In the following, the various language constructs are explained. From now on, we will refer to the set $TV = \{\tau_0, \tau_1, \dots\}$ of timestamp variables over $\mathbb{T}$, the set $IV = \{\sigma_0, \sigma_1, \dots\}$ of index variables over $\mathbb{J}$, and the set $RV = \{\rho_0, \rho_1, \dots\}$ of real-valued variables over $\mathbb{R}$.

- A *term* (non-terminal `tm`) can be either a *time term*, an *index term*, or a *value term*.

- A *time term* (non-terminal `tt`) allows users to refer to timestamps in the specifications. A time term may be a timestamp variable $\tau \in TV$, a literal denoting a value $t \in \mathbb{T}$, the value returned by the operator `i2t`, or an arithmetic expression over these entities. The operator `i2t(it)` takes an index term as argument and returns the timestamp associated with the record at the (trace) index `it`. An example of time term is the expression $\tau_0 + 5.5 + \texttt{i2t}(2)$.

- An *index term* (non-terminal `it`) is used to refer to trace indices in the specifications. An index term can be an index variable $\sigma \in IV$, a literal denoting a value $j \in \mathbb{J}$, the value returned by the operator `t2i`, or an arithmetic expression over these entities. The operator `t2i`(`tt`) takes a time term as argument and returns the index j of the trace record with timestamp t_j, where t_j is the highest timestamp value for which $t_j \leq$ `tt`. An example of index term is the expression $\sigma_0 + 2 +$ `t2i`(3.3).

- A *value term* (non-terminal `vt`) allows users to refer to real values (e.g., signal values) in the specifications. A value term can be a real-valued variable $\rho \in RV$, a literal denoting a value $x \in \mathbb{R}$, the value of a signal returned by the operators `@i` ("at index") and `@t` ("at timestamp"), or an arithmetic expression over these entities.

  1. The `@i` operator is an infix operator that takes two arguments: a signal s and an index term `it`; it returns the value of signal s associated with the record at the (trace) index `it`.

  2. The `@t` operator is an infix operator that takes two arguments: a signal s and a time term `tt`; it returns the value of signal s associated with a record at timestamp t_j, where t_j is the highest timestamp value in the trace for which $t_j \leq$ `tt`.

  An example of value term is the expression $(s_1$`@i`$2)+(s_2$`@t`$3.3)+\rho_0+5.2$, where s_1 and s_2 are signals, 2 is an index term, 3.3 is a time term, ρ_0 is a real-valued variable, and 5.2 is a numeric literal.

- A *formula* (non-terminal `p`) is a relational expression over terms, a logical expression over other formulae defined by Boolean connectives, or an existentially quantified formula. As anticipated in section 5.1, we design HLS to be able to support three types of quantification:

  1. over timestamp variables, as in "`exists` τ `in` I_T $[\dots]$", where I_T is a time range with bounds in $\mathbb{T}$;

  2. over index variables, as in "`exists` σ `in` I_J $[\dots]$", where I_J is a range of index values with bounds in $\mathbb{J}$;

  3. over real-valued variables, as in "`exists` ρ $[\dots]$".

  For example, the formula `exists` σ_0 `in` $[3,5]$ `such that` $(s_1$`@i`$\sigma_0) < 2.5$ specifies that there exists a record with index greater than or equal to 3 and lower than or equal to 5, in which the value of signal s_1 is less than 2.5.

We further extended the language with additional relational operators and logical connectives (e.g., implication (`implies`), conjunction (`and`)), in addition to universal quantifiers (`forall`) on timestamp variables, index variables, and real-valued variables, using the standard logical conventions.

An application of HLS for the specification of one of the requirements in our case study is presented in the following.

Let us consider a fragment of requirement $\mathcal{R}1$:

Whenever the satellite mode switches
from "Idle Mode" to "Normal Mode Fine",
the satellite angular rate shall reach
a value lower than $1.5\,°/\mathrm{s}$ *within* $10\,\mathrm{s}$.

recalling that the satellite mode is represented by the signal `mode`, for which value 0 corresponds to "Idle Mode" and value 3 corresponds to "Normal Mode Fine"; also, the angular rate is represented by the signal `ang-rate`. This fragment can be specified in HLS as:

$$\begin{array}{l}
\texttt{forall}\ \sigma_0\ \texttt{in}\ [0,5]\ \texttt{such that} \\
\quad ((\texttt{mode @i}\ \sigma_0) = 0\ \texttt{and}\ (\texttt{mode @i}\ (\sigma_0 + 1)) = 3) \\
\quad \texttt{implies exists}\ \tau_0\ \texttt{in}\ [0\,\mathrm{s}, 10\,\mathrm{s}]\ \texttt{such that} \\
\quad (\texttt{ang-rate @t}\ (\tau_0 + \texttt{i2t}(\sigma_0)) < 1.5))
\end{array}$$

The sub-formula $((\texttt{mode @i}\ \sigma_0) = 0\ \texttt{and}\ (\texttt{mode @i}\ (\sigma_0 + 1)) = 3)$ detects when the satellite switches from "Idle Mode" to "Normal Mode Fine" over two consecutive records (notice the use of the "at index" operator to refer to the consecutive indices σ_0 and $\sigma_0 + 1$). This expression lies within the scope of the outer universal quantifier, which iterates over a range of values for the index variable σ_0. This range depends on the length of the trace and the use of σ_0 in the formula. In this case, since the requirement states "*whenever* [the satellite mode switches...]", in the specification the full length of the trace fragment π^e in Figure 4.1 has to be covered; its record index values span from 0 to 6. We achieve this by setting the lower bound to zero and the upper bound to five; in this way, the term $\texttt{mode @i}\ (\sigma_0 + 1)$ always refers to a record index of the example trace.

The inner quantification over the timestamp variable τ_0 checks whether the angular rate of the satellite reaches a value lower than $1.5\,°/\mathrm{s}$ in a $10\,\mathrm{s}$ timeframe. More specifically, the expression $(\texttt{ang-rate @t}\ (\tau_0 + \texttt{i2t}(\sigma_0)) < 1.5)$ represents the value of signal `ang-rate` at timestamp $\tau_0 + \texttt{i2t}(\sigma_0)$, where τ_0 is in the interval $[0\,\mathrm{s}, 10\,\mathrm{s}]$, which corresponds to the distance of $10\,\mathrm{s}$, and

Time Term Interpretation

$$[\![\tau]\!]_{\pi,\mu} = \mu^{TV}(\tau), \text{ for all } \tau \in TV; \qquad [\![t]\!]_{\pi,\mu} = t, \text{ for all } t \in \mathbb{T};$$
$$[\![\texttt{i2t}(\texttt{it})]\!]_{\pi,\mu} = \pi[[\![\texttt{it}]\!]_{\pi,\mu}].t;$$
$$[\![f(\texttt{tt}_1, \texttt{tt}_2)]\!]_{\pi,\mu} = [\![f]\!]_{\pi,\mu}([\![\texttt{tt}_1]\!]_{\pi,\mu}, [\![\texttt{tt}_2]\!]_{\pi,\mu});$$

Index Term Interpretation

$$[\![\sigma]\!]_{\pi,\mu} = \mu^{IV}(\sigma), \text{ for all } \sigma \in IV; \qquad [\![j]\!]_{\pi,\mu} = j, \text{ for all } j \in \mathbb{J};$$
$$[\![\texttt{t2i}(\texttt{tt})]\!]_{\pi,\mu} = \iota_\pi([\![\texttt{tt}]\!]_{\pi,\mu});$$
$$[\![f(\texttt{it}_1, \texttt{it}_2)]\!]_{\pi,\mu} = [\![f]\!]_{\pi,\mu}([\![\texttt{it}_1]\!]_{\pi,\mu}, [\![\texttt{it}_2]\!]_{\pi,\mu});$$

Value Term Interpretation

$$[\![\rho]\!]_{\pi,\mu} = \mu^{RV}(\rho), \text{ for all } \rho \in RV; \qquad [\![x]\!]_{\pi,\mu} = x, \text{ for all } x \in \mathbb{R};$$
$$[\![(s \texttt{ @i } \texttt{it})]\!]_{\pi,\mu} = \pi[[\![\texttt{it}]\!]_{\pi,\mu}].s; \quad [\![(s \texttt{ @t } \texttt{tt})]\!]_{\pi,\mu} = \pi[\iota_\pi([\![\texttt{tt}]\!]_{\pi,\mu})].s$$
$$[\![f(\texttt{vt}_1, \texttt{vt}_2)]\!]_{\pi,\mu} = [\![f]\!]_{\pi,\mu}([\![\texttt{vt}_1]\!]_{\pi,\mu}, [\![\texttt{vt}_2]\!]_{\pi,\mu});$$

Formula Satisfaction

$(\pi,\mu) \models \texttt{tm}_1 < \texttt{tm}_2$	iff	$[\![\texttt{tm}_1]\!]_{\pi,\mu} < [\![\texttt{tm}_2]\!]_{\pi,\mu}$
$(\pi,\mu) \models \texttt{not p}$	iff	$(\pi,\mu) \not\models \texttt{p}$
$(\pi,\mu) \models \texttt{p}_1 \texttt{ or } \texttt{p}_2$	iff	$(\pi,\mu) \models \texttt{p}_1$ or $(\pi,\mu) \models \texttt{p}_2$
$(\pi,\mu) \models \texttt{exists } \tau \texttt{ in } I_T$ `such that p`	iff	$(\pi,\mu) \models \texttt{p}[\tau \leftarrow t_j]$ for some $t_j \in I_T$
$(\pi,\mu) \models \texttt{exists } \sigma \texttt{ in } I_J$ `such that p`	iff	$(\pi,\mu) \models \texttt{p}[\sigma \leftarrow j]$ for some $j \in I_J$
$(\pi,\mu) \models \texttt{exists } \rho$ `such that p`	iff	$(\pi,\mu) \models \texttt{p}[\rho \leftarrow v]$ for some $v \in \mathbb{R}$

Figure 5.2: Semantics of the Hybrid Logic of Signals.

$\texttt{i2t}(\sigma_0)$ is the timestamp at which the satellite switches from "Idle Mode" to "Normal Mode Fine", i.e., the timestamp associated with the record at index σ_0.

5.4 Semantics

To evaluate whether an HLS formula is true or false over a trace π, it must first be defined how time, index, and value terms are interpreted and evaluated.

Let $\mu^{TV}, \mu^{IV}, \mu^{RV}$ be variable assignments, respectively, for timestamp, index, and real-valued variables; for example, μ^{TV} is a mapping from a timestamp variable in TV to a value in $\mathbb{T}$. Let μ denote, collectively, the family of variable assignment functions $\mu^{TV}, \mu^{IV}, \mu^{RV}$. A generic term $\texttt{tm}$ is evaluated on a trace π, using the variable assignment functions in μ, by means of an interpretation function $[\![\texttt{tm}]\!]_{\pi,\mu}$.

We represented the interpretation of HLS terms inductively at the top of figure 5.2.

- For all three term types, the interpretation of a literal is the value denoted by the literal itself;
- a variable is interpreted using the variable assignment function for the corresponding type;
- an arithmetic expression defined using a function f is interpreted applying the interpretation of the function symbol f to the interpretation of the corresponding arguments.
- The operators `i2t`, `t2i`, `@i`, and `@t` are interpreted according to the informal semantics provided in the previous section.

The semantics of an HLS formula ϕ is defined over a trace π and a variable assignment μ; the notation $(\pi, \mu) \models \phi$ indicates that trace π satisfies formula ϕ under variable assignment μ. The satisfiability relation of HLS formulae is defined inductively at the bottom of figure 5.2.

- The formula $\mathtt{tm}_1 < \mathtt{tm}_2$ is satisfied if and only if (iff) the interpretation of term $\mathtt{tm}_1$ is lower than the interpretation of term $\mathtt{tm}_2$.
- The semantics of the Boolean connectives `or` and `not` is the standard one.
- A formula with an existential quantifier over a timestamp variable, of the form `exists` τ `in` I_T `such that` `p`, is satisfied iff there exists a timestamp $t_j \in I_T$, such that when substituting timestamp t_j for τ in the formula `p` (denoted by $\mathtt{p}[\tau \leftarrow t_j]$), the resulting formula is satisfied.
- A formula with an existential quantifier over an index variable, of the form `exists` σ `in` I_J `such that` `p`, is satisfied iff there exists an index $j \in I_J$, such that when substituting index j for σ in the formula `p` (denoted by $\mathtt{p}[\sigma \leftarrow j]$), the resulting formula is satisfied.
- Finally, a formula with an existential quantifier over a real-valued variable, of the form `exists` ρ `such that` `p`, is satisfied iff there exists a value $v \in \mathbb{R}$, such that, when substituted for ρ in the formula `p` (denoted by $\mathtt{p}[\rho \leftarrow v]$), the formula is satisfied

.

Spike
p ::= `spike in interval` I_T `and signal` (s `@i tt`) `with width` ρ `and amplitude` ρ

Oscillation
p ::= `oscillation in interval` I_T `and signal` (s `@i tt`) `with p2pAmp` ρ `and period` ρ

Rise
p ::= `Signal` (s `@i tt`) `rises in interval` I_T `reaching` ρ

Rise
p ::= `Signal` (s `@i tt`) `rises monotonically in interval` I_T `reaching` ρ

Fall
p ::= `Signal` (s `@i tt`) `falls in interval` I_T `reaching` ρ

Monotonic Fall
p ::= `Signal` (s `@i tt`) `falls monotonically in interval` I_T `reaching` ρ

Overshoot
p ::= `Signal` (s `@i tt`) `overshoots in interval` I_T `value` ρ

Monotonic Overshoot
p ::= `Signal` (s `@i tt`) `overshoots monotonically in interval` I_T `value` ρ

Undershoot
p ::= `Signal` (s `@i tt`) `undershoots in interval` I_T `value` ρ

Monotonic Undershoot
p ::= `Signal` (s `@i tt`) `undershoots monotonically in interval` I_T `value` ρ

$$\tau \in TV, \sigma \in SV, \rho \in RV, s \in S$$

Figure 5.3: Additional construct representing common signal patterns

5.5 Additional Constructs

We extended the grammar in Figure 5.1 with additional constructs. These constracts represent recurrent patterns for signal-based properties included within the SB-TemPsy [27] language and identified in a recent taxonomy [26]. These constructs act as shortcuts to express properties, relative to the common behaviour of signals in industrial cases. They are automatically compiled into HLS specifications.

Figure 5.3 presents an overview of these patterns. These patterns are among the most commonly employed to describe the properties of signal, and they can be of major practical utility, especially when considering the behaviour of physical variables. For example, the oscillation pattern allows designers to easily express properties such that “The velocity of an object along one of its axes shall oscillate with a maximum amplitude of 10m/s and a maximum period of 30s”.

Chapter 6

ThEodorE

In this chapter, we introduce ThEodorE, our trace checker for HLS. We first provide a high-level overview of the trace-checking approach used by ThEodorE (Section 6.1). ThEodorE reduces the problem of checking an HLS property on a trace to a satisfiability problem, which can be handled using off-the-shelf SMT solvers. Then, we present how the trace (Section 6.2) and the property of interest (Section 6.3) are translated into the input language of the SMT solvers. Finally, we describe how existing SMT solvers are reused to solve the trace-checking problem (Section 6.4).

6.1 Trace Checking Approach

ThEodorE takes as input a property ϕ expressed in HSL and a trace π. Figure 6.1 provides a high-level view of the components of ThEodorE and the data flow among them: function $\mathfrak{t}$, function $\mathfrak{h}$, and the *Satisfiability Checking* module. The ThEodorE trace-checking approach follows two steps:

- `Step 1:` ThEodorE automatically translates property ϕ and trace π into formulae, expressed using a target logic $\mathcal{L}$. This translation relies on two translation functions $\mathfrak{t}$ (for traces, see Section 6.2) and $\mathfrak{h}$ (for HSL formulae, see Section 6.3) and guarantees, given a variable assignment μ, that

$$(\pi, \mu) \models \phi \textbf{ iff } \mathfrak{h}(\neg\phi) \wedge \mathfrak{t}(\pi) \text{ is not satisfiable.}$$

- `Step 2:` ThEodorE (Section 6.4) checks satisfiability of formula $\psi \equiv \mathfrak{h}(\neg\phi) \wedge \mathfrak{t}(\pi)$, expressed in the target logic $\mathcal{L}$ using an SMT solver.

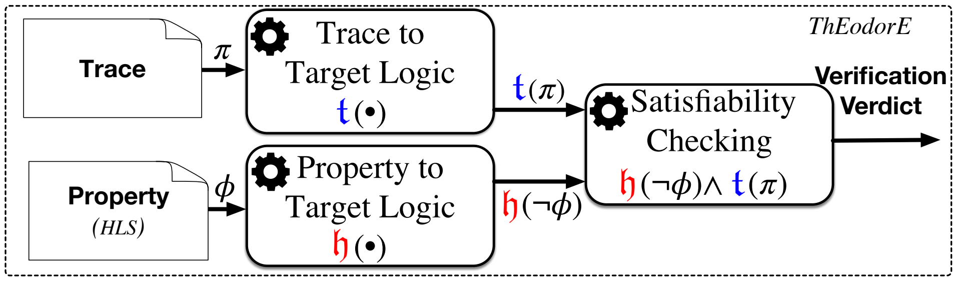

ThEodorE
Trace
π
Trace to Target Logic $\mathfrak{t}(\bullet)$
$\mathfrak{t}(\pi)$
Satisfiability Checking $\mathfrak{h}(\neg\phi) \wedge \mathfrak{t}(\pi)$
Verification Verdict
Property (HLS)
ϕ
Property to Target Logic $\mathfrak{h}(\bullet)$
$\mathfrak{h}(\neg\phi)$

Figure 6.1: ThEodorE: a trace checker for HLS.

The target logic $\mathcal{L}$ to be selected for trace checking of HSL properties in ThEodorE shall fulfill two goals:

G1: being sufficiently expressive to encode the logic-based representation of a trace π and the (semantics of an) HLS formula ϕ. This means that it should include linear real arithmetic (to support real-valued and timestamp terms), quantifiers (since HLS is a first-order logic), and arrays (since a trace can be seen as an array of records).

G2: being supported by an efficient solver, so that the trace checking procedure for HSL formulae can be completed within practical time limits.

We have identified the *AUFLIRA* (Closed linear formulae with free sort and function symbols over one- and two-dimensional arrays of integer indices and real values) fragment of the SMT-LIB (Satisfiability Modulo Theories LIBrary) logic [19] as a suitable target logic for ThEodorE. The theories used by AUFLIRA are identifiable through its name: A: arrays; UF: extension allowing free sort and function symbols; LIRA: linear integer and real arithmetics. Furthermore, AUFLIRA does not restrict the formulae to be quantifier-free. Based on the list of supported theories, AUFLIRA satisfies **G1**. It also satisfies **G2**, since it is included in the SMT-LIB logic, whose satisfiability can be verified using highly efficient and optimized solvers, as shown in the annual SMT competition [72].

For simplicity, and to better reflect ThEodorE's implementation, instead of describing a translation from HLS to the SMT-LIB language [19], we will present a translation from HLS to a specification expressed using the more compact and high-level syntax provided by the Z3 Python API [13] (which internally represents the specification in a form equivalent to the SMT-LIB language). Function $\mathfrak{t}$ translates a trace π into a logic formula expressed using the target logic $\mathcal{L}$.

6.2 Translating a Trace into the Target Logic

The sequence of timestamps in π is translated creating an array variable `t`; the type of the array indices (i.e., the domain of `t`) is $\mathbb{Z}$, whereas the type of the array values (i.e., the range of `t`) is $\mathbb{R}$. Then, a series of constraints on the values in `t` are enforced: the value of the array `t` at position i (denoted by `t[`i`]`) is constrained to be equal to the value of the timestamp contained in the record at index i of trace π.

Also, the translation procedure initializes an array variable *for each signal* whose values are used in the expression of the requirement; the variable name is obtained by concatenating `v_` with the name of the signal. For each signal, represented by an array variable, a series of constraints are asserted on the values of the array: the value of the array in position i is constrained to be equal to the value of the corresponding signal in the record at index i of trace π.

For example, Listing 6.1 illustrates the translation in our target logic of the trace depicted in Figure 4.1. Line 4 of Listing 6.1 declares the array variable `t`. Line 8 forces the timestamp in position 6 to be equal to 5.7. Line 9 declares the array variable associated to the signal `ang-rate`. Line 10 sets the value of the array variable `v_ang-rate` to be equal to 20.1 in position 0.

Listing 6.1: An excerpt of the formula generated for the trace depicted in Figure 4.1.

```
1
2  s=SolverFor(`AUFLIRA')
3  ...
4  t=Array(`t',IntSort(),RealSort());
5  s.add(t[0]==0)
6  s.add(t[1]==0.2)
7  ...
8  s.add(t[6]==5.7)
9  v_ang-rate=Array('ang-rate',IntSort(),RealSort());
10 s.add(v_ang-rate[0]==20.1)
11 s.add(v_ang-rate[1]==22.2)
12 ...
13 s.add(v_ang-rate[6]==1.1)
```

6.3 Translating an HLS Formula into the Target Logic

Function $\mathfrak{h}$ translates an HSL statement into a formula expressed using the target logic $\mathcal{L}$.

Firstly, the translation declares a new variable for each timestamp, each index, and each real-valued variable; the name of the new variable is created by concatenating `v_` with the name of the initial variable. The type of the new variables is `Real` for timestamps and real-valued variables, and `Int` for index variables.

Afterward, the translation recursively evaluates each node in the parse tree of the input formula, commencing from the root node; each node is translated using the rules presented in Figure 6.2.

The translation of time, index, and values term nodes are established as follows.

- Nodes referring to HSL variables are translated into the corresponding variables in the target logic formula.
- Literal nodes are mapped into literals in the target logic formula.
- Arithmetic expressions using a function f are translated by converting the function symbol into the equivalent in the target language, and then by applying it to the translation of its arguments.
- A time term node of the form `i2t`(`it`) is translated into an expression that accesses the element of the array `t` in position $\mathfrak{h}$(`it`).
- An index term node of the form `t2i`(`tt`) is translated into the application of the translation of function ι to $\mathfrak{h}$(`tt`).
- A value term of the form (s `@i` `it`) is translated into an expression that retrieves the value of variable `v_`s at index $\mathfrak{h}$(`it`).
- A value term of the form (s `@t` `tt`) is translated into an expression that retrieves the value of variable `v_`s at the index obtained through the evaluation of $\mathfrak{h}(\iota)(\mathfrak{h}($`tt`$))$.

The translation of function ι provides solutions for both definitions presented in section 5.2. It consists of a rewriting of the definition into the corrisponding syntax of the target logic. it is important to mention that the size of the arithmetic expression to compute $\mathfrak{h}(\iota^V)$ in the case of a variable sample rate is linear in the length of the trace and the number of timestamp variables.

Time Term
$\mathfrak{h}(\tau) = \texttt{v\_}\tau$, for all $\tau \in TV$;
$\mathfrak{h}(t) = t$, for all $t \in \mathbb{T}$;
$\mathfrak{h}(f(\texttt{tt}_1, \texttt{tt}_2)) = \mathfrak{h}(f)(\mathfrak{h}(\texttt{tt}_1), \mathfrak{h}(\texttt{tt}_2))$;
$\mathfrak{h}(\texttt{i2t}(\texttt{it})) = \texttt{t[}\mathfrak{h}(\texttt{it})\texttt{]}$;
Index Term
$\mathfrak{h}(\sigma) = \texttt{v\_}\sigma$, for all $\sigma \in IV$;
$\mathfrak{h}(j) = j$, for all $j \in \mathbb{J}$;
$\mathfrak{h}(f(\texttt{it}_1, \texttt{it}_2)) = \mathfrak{h}(f)(\mathfrak{h}(\texttt{it}_1), \mathfrak{h}(\texttt{it}_2))$;
$\mathfrak{h}(\texttt{t2i}(\texttt{tt})) = \mathfrak{h}(\iota)(\mathfrak{h}(\texttt{tt}))$;
Value Term
$\mathfrak{h}(\sigma) = \texttt{v\_}\sigma$, for all $\sigma \in RV$;
$\mathfrak{h}(x) = x$, for all $x \in \mathbb{R}$;
$\mathfrak{h}(f(\texttt{vt}_1, \texttt{vt}_2)) = \mathfrak{h}(f)(\mathfrak{h}(\texttt{vt}_1), \mathfrak{h}(\texttt{vt}_2))$
$\mathfrak{h}((s\ \texttt{@i}\ \texttt{it})) = \texttt{v\_s[}\mathfrak{h}(\texttt{it})\texttt{]}$;
$\mathfrak{h}((s\ \texttt{@t}\ \texttt{tt})) = \texttt{v\_s[}\mathfrak{h}(\iota)\,(\mathfrak{h}\,(\texttt{tt}))\texttt{]}$;
Formula (with $I_T = [t_a, t_b]$ and $I_J = [a, b]$)
$\mathfrak{h}(\texttt{tm}_1 < \texttt{tm}_2) = \mathfrak{h}(\texttt{tm}_1) < \mathfrak{h}(\texttt{tm}_2)$;
$\mathfrak{h}(\texttt{p}_1\ \texttt{or}\ \texttt{p}_2) = \texttt{Or}(\mathfrak{h}(\texttt{p}_1), \mathfrak{h}(\texttt{p}_2))$;
$\mathfrak{h}(\texttt{not p}) = \texttt{Not}(\mathfrak{h}(\texttt{p}))$;
$\mathfrak{h}(\texttt{exists}\ \tau\ \texttt{in}\ I_T\ \texttt{such that p}) = \texttt{Exists}\,(\texttt{v\_}\tau, \texttt{And}\,(\texttt{And}\,(t_a \leq \texttt{v\_}\tau, \texttt{v\_}\tau \leq t_b)\,, \mathfrak{h}\,(\texttt{p})))$
$\mathfrak{h}(\texttt{exists}\ \sigma\ \texttt{in}\ I_J\ \texttt{such that p}) = \texttt{Exists}\,(\texttt{v\_}\sigma, \texttt{And}\,(\texttt{And}\,(a \leq \texttt{v\_}\sigma, \texttt{v\_}\sigma \leq b)\,, \mathfrak{h}\,(\texttt{p})))$
$\mathfrak{h}(\texttt{exists}\ \rho\ \texttt{such that p}) = \texttt{Exists}(\texttt{v\_}\rho, \mathfrak{h}(\texttt{p}))$

Figure 6.2: Rules for translating HSL formulae into $\mathcal{L}$.

Evaluating the impact of our translation and the selection of the definition of function ι on the performance of the trace-checking procedure is part of our evaluation.

The translation of HSL formulae is then basically their rewriting into the equivalent syntax of the target logic, modulo the translation of the variables and the sub-formulae. For example, an expression of the kind:

$$\texttt{exists}\ \rho\ \texttt{such that p}$$

is rewritten as

$$\texttt{Exists}(\texttt{v\_}\rho, \mathfrak{h}(\texttt{p}))$$

where the target logic variable $\texttt{v\_}\rho$ corresponds to variable ρ in the HSL formula, and $\mathfrak{h}(\texttt{p})$ is the translation of sub-formulae $\texttt{p}$.

For example, the encoding of the portion of requirement $\mathcal{R}1$ introduced in

Section 5.3, considering the function ι' encoding for the variable sample-rate, is presented in Listing 6.2 as translated in our target logic.

Listing 6.2: The formula generated for the fragment of requirement $\mathcal{R}1$ presented in Section 5.3.

```
v_$\sigma_0$=Int(`v_$\sigma_0$')
v_$\tau_0$=Real(`v_$\tau_0$')
s.add(Not(Exists(v_$\sigma_0$,
  And(And(0 <= v_$\sigma_0$,v_$\sigma_0$ <= 5),
   Not(Exists(v_$\tau_0$,
    And(And(0 <= v_$\tau_0$ ,v_$\tau_0$ <= 10),
     v_ang-rate[v_$\tau_0$ +
     (((0 <= v_$\tau_0$ )*(v_$\tau_0$ <0.2))*0 +
       (0.2 <= v_$\tau_0$)*(v_$\tau_0$ <0.9))*1 + $\ldots$ +
       (5.7 <= v_$\tau_0$ )*6]
       ) < 1.5))))))
```

6.4 Satisfiability Checking

We verify the satisfiability of the formula $\psi = \mathfrak{h}(\neg p) \wedge f(\pi)$. The formula ϕ holds on the trace π, if the formula ψ is not satisfiable. Indeed, if ψ is not satisfiable the assignments of the trace π do not make $\neg p$ satisfiable. Viceversa, if ψ is satisfiable it is possible to perform an assignment that satisfies $\neg p$. Therefore, p does not hold on the trace π.

The final verdict yielded by ThEodorE can be "*satisfied*", "*violated*" or "*unknown*"; it is based on the answer of the solver. ThEodorE yields the *definitive verdicts* "satisfied" or "violated" when the solver returns "UNSAT" or "SAT", indicating, respectively, that ψ is unsatisfiable or satisfiable. However, the solver may return an "UNKNOWN" answer, since the satisfiability of the underlying target logic $\mathcal{L}$ is generally undecidable. In our case, this indicates that no conclusion is drawn on the satisfiability of formula ψ, resulting in an "unknown" verdict returned by ThEodorE. Assessing whether this is a frequent case in practical applications is part of our evaluation (Chapter 8).

ThEodorE ensures that $(\pi, \mu) \models \phi$ **iff** $\mathfrak{h}(\neg\phi) \wedge \mathfrak{t}(\pi)$ is not satisfiable. The correctness of our procedure is based on two arguments:

(i) $\mathfrak{t}$ translates the trace π into a set of array variables whose values are set according to the values of the original trace, and

(ii) $\mathfrak{h}$ rewrites the HLS formula into the target logic without applying any change (that could alter the semantics) to the structure of the formula.

Chapter 7

Implementation

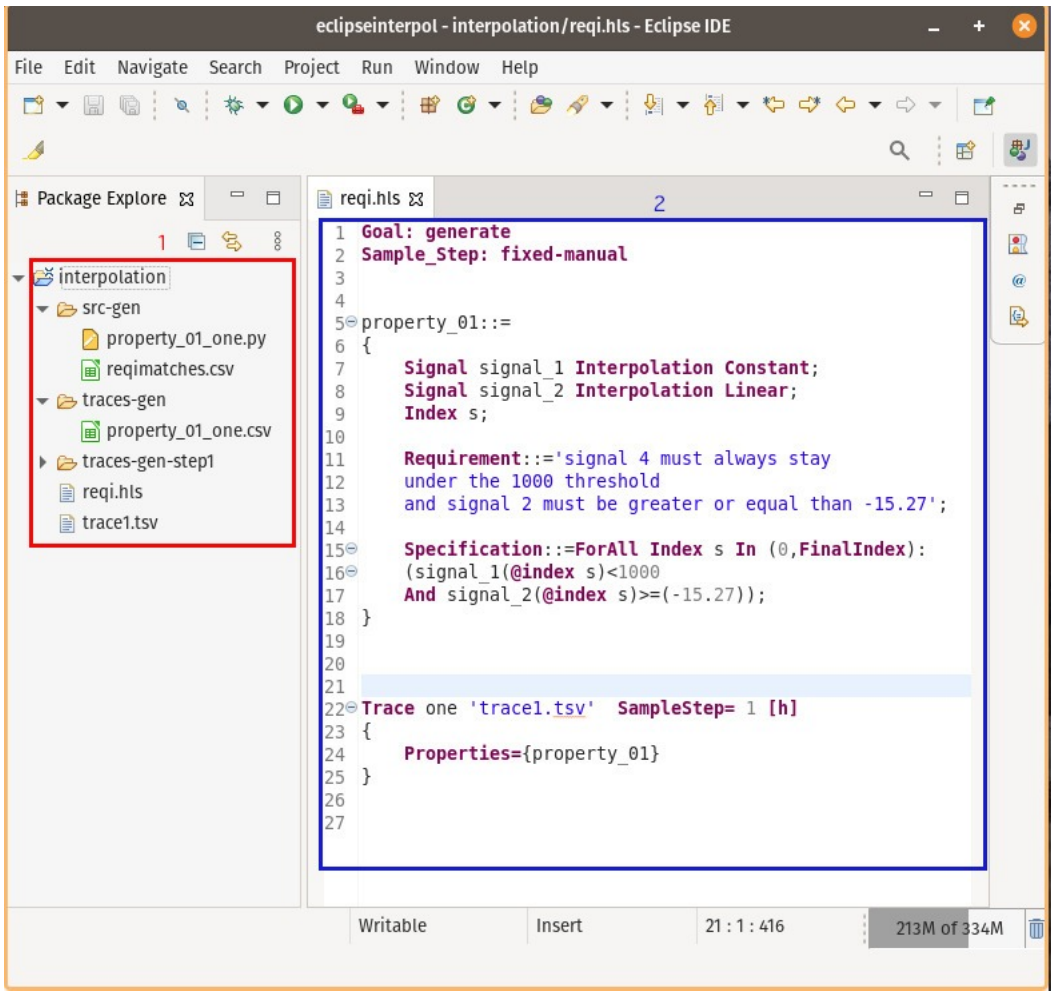
In this chapter, we describe the implementation of ThEodorE. First (Section 7.1), we provide an overview of ThEodorE, its components, how they were implemented, their inputs and their outputs. Then, we describe the grammar of ThEodorE's specification language (Section 7.2) and the code generator that translates the requirements into an SMT instance (Section 7.3). We describe some of the options provided by ThEodorE (Section 7.4) and the structure of ThEodorE's outputs (Section 7.5). Finally, we describe some options provided by ThEodorE (Section 7.6).

7.1 Overview

An overview of ThEodorE is provided in Figure 7.1. ThEodorE consists of three main components:

- A Domain-Specific Language (DSL) designed to express requirements in Hybrid Logic of Signals. This language was developed with Xtext [11] and it is defined by the grammar illustrated in Section 7.2. ThEodorE provides a GUI that enables users to write their specifications using this domain specific language. The GUI support provided by ThEodorE to write specifications is represented within the blue squared box labelled with (2) in Figure 7.1.
- A code generator developed in Xtend [10], a Java dialect created to be used with Xtext. It translates the declared requirements from HSL to the target Satisfiability Modulo Theories [2] logic and specify the values of every record of the signals contained in the traces into the same SMT formula as their relative requirements. It uses a pre-processing

algorithm for the system traces, also written in Xtend, that extracts the relevant values from the raw traces and can convert them from a variable-sample to a fixed-sample step if requested, interpolating missing values when needed (see Section 7.4). The outputs of the ThEodorE' code generator are represented within the red squared box labelled with (1) in Figure 7.1.



```
Goal: generate
Sample_Step: fixed-manual


property_01::=
{
    Signal signal_1 Interpolation Constant;
    Signal signal_2 Interpolation Linear;
    Index s;

    Requirement::='signal 4 must always stay
    under the 1000 threshold
    and signal 2 must be greater or equal than -15.27';

    Specification::=ForAll Index s In (0,FinalIndex):
    (signal_1(@index s)<1000
    And signal_2(@index s)>=(-15.27));
}



Trace one 'trace1.tsv'  SampleStep= 1 [h]
{
    Properties={property_01}
}
```

Figure 7.1: ThEodorE as a plugin installed on Eclipse

In practice, ThEodorE takes as inputs an ".hls" file written in the DSL (see Subsection 7.6), and the execution traces of the systems in ".tsv" or ".csv" format (more on their structure in Subsection 7.4). Its output is a Python [8] executable file (see Section 7.5) for every trace-requirement couple. Every file contains the expression of one requirement plus the relevant signals extrapolated from the trace's contents, declared in Z3py [14], the Python front-end of our chosen SMT compatible solver, Z3 [33], developed by Microsoft Research.

ThEodorE was packaged into a plugin for the Eclipse IDE [43], a very popular Java Integrated Development Environment, which was also used to write most of the source-code of ThEodorE.

Once installed, ThEodorE's interface will be that of a typical Eclipse editor, complete with syntax highlighting for HLS, as shown in Figure 7.1.

To create a ThEodorE project, it is sufficient to initialize a generic project on Eclipse and then create a file with the ".hls" extension inside the main folder.

A more detailed description of the structure and functions of the aforementioned file format will follow, in subsection 7.6.

7.2 Grammar

The underlying structure of ThEodorE's input files is described in the Xtext grammar language, which is itself a DSL [12] used to define textual languages. Xtext automatically generates a parser for the grammar, that will consume any ".hls" file and instantiate an Abstract Syntax Tree (AST) that ThEodorE's code generator (see Section 7.4) will translate into the target logic.

In Xtext, a language is described through a series of parser rules and terminal rules that act as instructions for the creation of a parse tree from the consumed text. A terminal rule will produce a single terminal token, a "leaf" in the tree, while a parser rule will act as a "node" that can be followed by both terminal and non-terminal tokens. The parse tree is also called "node model". The nodes of the tree, generated by the parser rules, are the building template for the EObjects **??** that form the AST.

One of the way used by the parser to differentiate between the various rules is the use of keywords, which are defined as reserved strings in the grammar.

In the following subsections, we will describe how the components of the language and the structure of the relative file format are represented in Xtext.

7.2.1 File Structure and Saving Options

Listing 7.1: The Xtext grammar file of ThEodorE (Structure)

```
grammar lu.svv.theodore.Hls with org.eclipse.xtext.common.
    Terminals

generate hls "http://www.theodore.svv.lu/theodore"

Hls:

  instructions=SavingInstructions
  samplestep=SampleStep
  requirements+=Requirement*
  traces += Trace*
  ;

SavingInstructions:
 "Goal:" instructions=("save"|"generate")
;

SampleStep:
  s="Sample_Step:" sample=('variable'|'fixed-manual'|'fixed-
      min')
;
```

The first rule in the grammar, `Hls`, (Listing 7.1) defines the structure and the components of the ".hls" file:

- One instance of the rule `SavingInstructions`, which describes the language component used to specify whether to run the code-generator or only edit is contained in the feature `instructions`;
- One instance of the rule `SampleStep`, that presents a similar structure, but is linked to the preprocessing phase is contained in the feature `samplestep`;
- An arbitrary number of (`Requirement`) rules are contained into `requirements`;
- An arbitrary number of traces (`Trace`) rules are contained into `traces`

The next lines describe the rules `SavingInstructions` and `SampleStep`: the components that they define allow the user to express parameters concerning the interpolation of the traces and the generated output file. More on their function will be explained in Section 7.6.

7.2.2 Traces and Requirements

Listing 7.2: The Xtext grammar file of ThEodorE (Trace objects)

```
Trace:
  "Trace" name=ID filePath=STRING ('SampleStep=' sampleStep=
      Value unit=('[h]'|'[min]'|'[s]'|'[ms]'|'[micros]'|'[
      nanos]'))?
  '{' ('Properties=' '{' requirementref+=[Requirement] (','
      requirementref+=[Requirement])* '}')?'}'
 ;
```

The structure of the rule `Trace` is shown in Listing 7.2. The language component defined by this rule allows the user to specify the path of the trace file and assign an arbitrary number of requirements to be verified on that trace.

Listing 7.3: The Xtext grammar file of ThEodorE (Requirement object)

```
VariableDefinition returns Variable:
  SampleVariable| TimeVariable | Signal |NumericVariable
;


SampleVariable returns SimpleVariable:
  'Index' {SampleVariable} name=ID ";"
;

TimeVariable returns SimpleVariable:
  'Timestamp' {TimeVariable} name=ID ";"
;

NumericVariable returns SimpleVariable:
  'Num' {NumericVariable} name=ID ";"
;

Signal returns Variable:
  'Signal' {Signal} name=ID ('Interpolation'
      interpolationType=('Linear'|'Constant'))?";"
;

Requirement:
  name=ID '::=' '{' variables+=VariableDefinition* ('
      Requirement' '::=' notes=STRING ';')? (spec=
      Specification)? '}'
;

Specification returns Specification:
  ('Specification' '::='expression=Expression ';')
;
```

Listing 7.3 shows the structure of a `Requirement` rule. `Requirement` define the language feature that contains the property to express and verify, plus some ancillary components:

- The `name` feature, defined by the terminal rule `ID` allows the requirement to be univocally identified;
- An arbitrary number of `VariableDefinition` rules are assigned to the feature `variables`;
- The `notes` feature, defined by the terminal rule `STRING` after the `Requirement::=` keyword;
- The `spec` feature is defined by the `Specification` rule.

`Specification` is then defined as the keyword `Specification::=` followed by the the feature `spec` that contains an `Expression` rule. `Expression` rules define the proper HLS grammar operators.

7.2.3 Expressions

The rest of the grammar is dedicated to the definition as Xtext rules of all the various logical and arithmetic operations that make up HLS. Everyone of them, while differentiated by the various operators acting as keyword, `return` the same rule type defined by the `Expression` label.

The rules describing logical expressions and first-order quantifiers are shown in Listing 7.4. The logical operators share with the arithmetic operations a binary structure, with a left-hand expression connected through an operator to the right hand one, while the quantifiers present a more complex structure.

Listing 7.4: The Xtext grammar file of ThEodorE (Logical operators)

```
TimeQuantifier returns Expression: (
  ({TimeQuantifier} op=("ForAll"|"Exists") 'Timestamp'
      function=[Variable]
  ("In" bracketdown=("["|"(") lower=TimeTermPlusOrMinus "," upper=TimeTermPlusOrMinus bracketup=("]"|")"))?
  (":" suchthat=Expression)));

SampleQuantifier returns Expression: (
  ({SampleQuantifier} op=("ForAll"|"Exists") 'Index' function=[Variable]
  ("In" bracketdown=("["|"(") lower=SampleTermPlusOrMinus "," upper=SampleTermPlusOrMinus bracketup=("]"|")"))?
  (":" suchthat=Expression)));

VariableQuantifier returns Expression: (
  ({VariableQuantifier} op=("ForAll"|"Exists") 'Value' function=[Variable]
  ("In" bracketdown=("["|"(") lower=ValueTermPlusOrMinus "," upper=ValueTermPlusOrMinus bracketup=("]"|")"))?
  (":" suchthat=Expression)));

Implication returns Expression:
   Or ({Implication.left=current} op=("->") right=Or)*;

Or returns Expression:
  And ({Or.left=current} op=("Or") right=And)*
;

And returns Expression:
  Negation ({And.left=current} op=("And") right=Negation)*;

Negation returns Expression:
  Primary | ({Neg} op="Not" '(' neg=Primary ')');

Primary returns Expression:
  TermRelation | '(' Expression ')'
;

TermRelation returns Expression:
   Term {TermRelation.left=current} op=(">="|"<="|">"|"<" | "=="|"!=")
    right=Term
```

Then rules describing the traditional arithmetic operations are defined over the different types of terms defined by the HLS grammar (see Chapter 5): time, sample, and value terms.

Listing 7.5 shows the defitions of the rules `TimeTermPlusOrMinus` and `TimeTermMulOrDiv` as an example.

Listing 7.5: The Xtext grammar file of ThEodorE (Arithmetic operations)

```
TimeTermPlusOrMinus returns Expression:
    TimeTermMulOrDiv (
    {TimeTermPlusOrMinus.left=current} op=('+'|'-')
    right=TimeTermMulOrDiv
  )* ;


TimeTermMulOrDiv returns Expression:
  TimeTermExponential (
    {TimeTermMulOrDiv.left=current} op=('*'|'/')
    right=TimeTermExponential
  )*
;
```

7.2.4 Singular Values

In this last section of the grammar (Listing 7.6), the terminal rules for the numeric values are defined, completing the definition of the rules regarding mathematical and logical operations.

Listing 7.6: The Xtext grammar file of ThEodorE (Numerals)

```
Value returns Expression:

'FinalIndex' {SampleTraceEnd}|
{IntNumber} value=INT (unit=('[h]'|'[min]'|'[s]'|'[ms]'|'[
    micros]'|'nanos'))? |
{DoubleNumber} upper=INT '.' lower=INT (unit=('[h]'|'[min]'|'
    [s]'|'[ms]'|'[micros]'|'nanos'))?|
{NegativeIntNumber} '(-' value=INT ')'(unit=('[h]'|'[min]'|'[
    s]'|'[ms]'|'[micros]'|'nanos'))?|
{NegativeDoubleNumber}'(-' upper=INT '.' lower=INT ')'(unit=(
    '[h]'|'[min]'|'[s]'|'[ms]'|'[micros]'|'nanos'))?|
'FinalTimestamp' {TimeTraceEnd}
```

```
;
```

7.3 Code Generator

ThEodorE's code generator produces the Python code of the output file. As specified in Section 7.1, we implemented the code generator using Xtend [10], a Java dialect developed by Eclipse to be used in tandem with Xtext.

We used Python as a translation language to have access to Z3py [14] a more readable front-end for the Z3 Solver [33]. We selected Z3 as SMT solver since it is an award-winning [4, 6], industry-strength tool and it is also compatible with *AUFLIRA*, the underlining logic chosen for the verification step (see Chapter 6).

Some of the notable components of the code generator are:

- `arithmethicRecursion`: a function that translates the arithmetic expressions contained in the `Specification` of every `Requirement` to the target logic: its output is a String that is appended to the output file.
- `formulaRecursion`: a function that translates the logic formulae contained in the `Specification` of every `Requirement` to the target logic: its output is a String that is appended to the output file.
- `processfile`: this function takes the pre-processed trace and writes the code for the initialization of the arrays representing the signals.

`arithmethicRecursion` and `formulaRecursion` together form the implementation of function $\mathfrak{h}$ (see Section 6.3), while `processfile` plays the role of function $\mathtt{t}$ (see Section 6.2).

An example of the code generator functioning is observable in Listing 7.7, which contains an excerpt of the function that handles the expressions shown in Listing 7.5.

Listing 7.7: The arithmethicRecursion function from ThEodorE's code generator

```
def String arithmethicRecursion(Expression expression) {
    var text = ""
    switch (expression) {
      [...]

      TimeTermMulOrDiv: {
        val left = arithmethicRecursion(expression.left)
        val right = arithmethicRecursion(expression.right)
        val op = expression.op
        text += left + op + right
      }

      [...]

      TimeTermPlusOrMinus: {
        val left = arithmethicRecursion(expression.left)
        val right = arithmethicRecursion(expression.right)
        val op = expression.op
        text += left + op + right
      }

      [...]
    }
    return text
  }
```

This is a recursive function that takes as input an `expression` object form the AST, extracts its features, translate them into strings of the target logic code, and appends them to the `text` string that is then returned. It iterates on both sides of every operator until the expressions are fully translated. The function `formulaRecursion` shares the same structure.

The function `processfile` is a function that opens the ".csv" file containing the preprocessed trace, saves the records for every signal into a list, and then proceeds to declare the corresponding arrays in a string of Z3py compatible code.

7.4 Trace Processing

The traces that ThEodorE accepts as inputs are written in the form of ".csv" or ".tsv" files, The files with the ".tsv" format are structured as the one shown in Listing 7.8. The ".tsv" and ".csv" formats were chosen due to being the output format for the logs of the satellite from our case-study (Chapter 4).

For every timestamp, listed in the first line of every block, there is a tabular structure with the name of the signal on the left and the respective value on the right; a line is allocated for every name-value couple.

To render the traces in a form that ThEodorE'code generator (see Section 7.3) can efficiently translate to the target logic, the traces are put trough a preprocessing phase in which:

- Only the signals that have been declared as variables for the current requirement are selected, to keep the preprocessed trace and, consequently, the executable file to a reasonable size.
- The trace can be converted to a fixed timestamp, if requested by the user, to allow for more efficient property verification since the function converting timestamps to index numbers and vice versa is less resource-intensive (see Chapter 8). This is especially useful when the size of the traces is particularly large.
- The values are rearranged in a more compact and simple structure and stored on a ".csv" file.
- The timestamps are changed to reflect the number of microseconds passed from the beginning of the trace recording.

Listing 7.8: Example of the structure of a "raw" trace

```

2020.102.11.39.01.521769
  signal_1 0
  signal_2 15
  signal_3 100
  signal_4 100

2020.102.13.39.02.753245
  signal_1 0
  signal_2 20
  signal_3 100
  signal_4 80

2020.102.18.39.02.753500
  signal_1 0
  signal_2 40
  signal_3 100
  signal_4 60

2020.102.19.00.02.753851
  signal_1 1
  signal_2 20
  signal_3 100
  signal_4 40

2020.102.20.39.02.754041
  signal_1 1
  signal_2 35
  signal_3 100
  signal_4 30

2020.102.23.00.03.455142
  signal_1 1
  signal_2 25
  signal_3 100
  signal_4 20

2020.103.01.15.03.455384
  signal_1 2
  signal_2 30
  signal_3 100
  signal_4 5

2020.103.02.39.03.457201
  signal_1 3
  signal_2 40
  signal_3 100
  signal_4 5

2020.103.05.43.44.186342
  signal_1 3
  signal_2 0
  signal_3 100
  signal_4 5

2020.103.07.43.55.198743
  signal_1 3
  signal_2 0
  signal_3 100
  signal_4 5
```

The trace represented in Listing 7.8 presents a total of ten timestamps, recorded with a variable sample-step. This raw trace was fed to ThEodorE to verify the property depicted in Listing 7.9, using the different options for the header value `Sample_Step` (Listing 7.14). This particular requirement was chosen to demonstrate an important detail of the preprocessing phase: it presents two signals with two different `Interpolation` methods selected: `Constant` and `Linear`.

Listing 7.9: Example of property

```

property_01::=
{
  Signal signal_4 Interpolation Constant;
  Signal signal_2 Interpolation Linear;
  Index s;
  Requirement::='signal 4 must always stay under the 1000
      threshold and signal 2 must be greater than or equal to
      -15.27';
  Specification::=ForAll Index s In (0,FinalIndex): (signal_4
      (@index s)<1000 And signal_2(@index s)>=(-15.27));
}
```

Figure 7.2 depicts the differences between the two methods:

- The `Constant` options are the most basic kind of interpolation and it is based on keeping the value of the signal constant until a new record comes along in the trace; this makes it more suited for signals from the digital domain such as status changes.
- The `Linear` options, as the name suggests, fills the missing data points with values taken from the hypothetical line connecting the two nearest known records. It is a more accurate representation of the continuous behavior of physical systems.

In the following sections, we will describe how a raw trace differs after being preprocessed according to the different choices available to the user: variable sample-step (Section 7.4.1), fixed sample-step using the shortest interval in the original trace (Section 7.4.2), and fixed sample step selected by the user (Section 7.4.2).

The trace shown in Listing 7.8 was fed to ThEodorE to demonstrate the results of the pre-processing phase with a practical example.

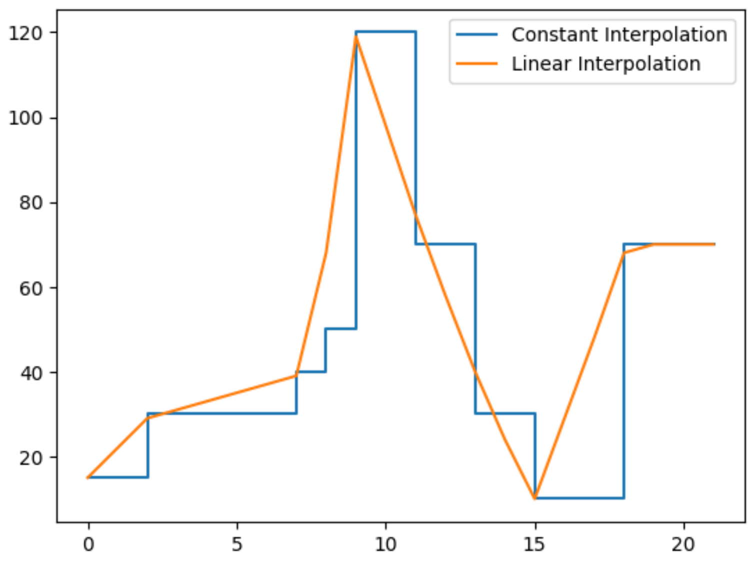


Figure 7.2: An example of Constant vs Linear Interpolation of the same trace.

7.4.1 Variable Sample-Step

In cases where is necessary to adhere as much as possible to the original trace, the user may elicit to keep a variable sample step.

This solution is guaranteed to leave the processed trace with:

- The same amount of records as the original trace.
- The exact same values for every signal at every timestamp, without interpolation unless the value of a signal is missing for one or more timestamps.

The processed trace (Listing 7.10), as expected, presents the same number of records and the same values as the original, with no signs of interpolation; only `Signal_2` and `Signal_4` are contained in the trace since they were the ones initialized in the property.

Listing 7.10: Trace after preprocessing mantaining the variable sample step of the raw trace

```

0,signal_4,100.0,signal_2,15.0
7201232000,signal_4,100.0,signal_2,20.0
25201232000,signal_4,80.0,signal_2,40.0
26461232000,signal_4,60.0,signal_2,20.0
32401233000,signal_4,40.0,signal_2,35.0
```

```
40861934000,signal_4,30.0,signal_2,25.0
48961934000,signal_4,20.0,signal_2,30.0
54001936000,signal_4,5.0,signal_2,40.0
65082665000,signal_4,5.0,signal_2,0.0
72293677000,signal_4,5.0,signal_2,0.0
```

When complete adherence to the original logs is not requested, the user may opt to convert the trace to a fixed sample-step, which represents a more efficient solution (see Chapter 8).

7.4.2 Fixed Sample-Step: Minimal

By selecting the `fixed-min` option, the user can choose to use the shortest interval in the original trace as a fixed-sample step. The shortest interval between records in the raw trace in Listing 7.8 was 1260000000 microseconds (21 minutes). Although it tends to increase the number of records in the trace, using `fixed-min` is a convenient solution to keep a high level of detail on the processed trace, avoiding the risk of canceling particularly fast dynamics. In this case, the number of records in the traces was increased from 10 to 58.

Listing 7.11: Excerpt of the trace after pre-processing using as a fixed sample-step the shortest interval of the raw trace

```

0,signal_4,100.0,signal_2,15.0
1260000000,signal_4,100.0,signal_2,15.874850303392531
2520000000,signal_4,100.0,signal_2,16.749700606785062
3780000000,signal_4,100.0,signal_2,17.624550910177593
5040000000,signal_4,100.0,signal_2,18.499401213570124

               ...

60480000000,signal_4,5.0,signal_2,16.615025960837052
61740000000,signal_4,5.0,signal_2,12.066588759638467
63000000000,signal_4,5.0,signal_2,7.518151558439882
64260000000,signal_4,5.0,signal_2,2.969714357241294
65520000000,signal_4,5.0,signal_2,0.0
66780000000,signal_4,5.0,signal_2,0.0
68040000000,signal_4,5.0,signal_2,0.0
69300000000,signal_4,5.0,signal_2,0.0
70560000000,signal_4,5.0,signal_2,0.0
```

```
71820000000,signal_4,5.0,signal_2,0.0
73080000000,signal_4,5.0,signal_2,0.0
```

For readability's sake the full extent of the trace is not reported, but, in Listing 7.11, the difference between the two interpolation strategies described in Section 7.4 is evident when looking at the values of the two different signals:

- `signal_4` records were generated with Constant Interpolation. There are long periods of constant values with abrupt changes from one value to the next.
- `signal_2` records were generated with Linear Interpolation, so the variations are more gradual and the trace rarely presents back to back constant values, unless the signal is actually constant, for example in the last part of the trace.

Fixed Sample-Step Manual

The trace shown in Listing 7.12 was interpolated with a manually adjusted fixed sample-step of one hour. This option allows for flexibility in the level of detail and in the number of records obtained after preprocessing. If interested only in the slow dynamics of a process, for example, a longer sample step might be useful in reducing the length of a particularly resource-intensive trace to a more manageable size.

Listing 7.12: Trace after preprocessing using as a fixed sample-step 1 hour

```

0,signal_4,100.0,signal_2,15.0
3600000000,signal_4,100.0,signal_2,17.49957229540723
7200000000,signal_4,100.0,signal_2,19.99914459081446
10800000000,signal_4,80.0,signal_2,23.99863111111111
14400000000,signal_4,80.0,signal_2,27.99863111111111
18000000000,signal_4,80.0,signal_2,31.99863111111111
21600000000,signal_4,80.0,signal_2,35.99863111111111
25200000000,signal_4,80.0,signal_2,39.99863111111111
28800000000,signal_4,60.0,signal_2,-17.12330158730159
32400000000,signal_4,40.0,signal_2,34.996886364160545
36000000000,signal_4,30.0,signal_2,30.746490745861365
39600000000,signal_4,30.0,signal_2,26.491524165669013
43200000000,signal_4,20.0,signal_2,26.44325061728395
```

```
46800000000,signal_4,20.0,signal_2,28.665472839506172
50400000000,signal_4,5.0,signal_2,32.853304423291895
54000000000,signal_4,5.0,signal_2,39.99615873168304
57600000000,signal_4,5.0,signal_2,27.011453849290962
61200000000,signal_4,5.0,signal_2,14.015918988723577
64800000000,signal_4,5.0,signal_2,1.0203841281561878
68400000000,signal_4,5.0,signal_2,0.0
72000000000,signal_4,5.0,signal_2,0.0
75600000000,signal_4,5.0,signal_2,0.0
```

7.5 Trace-Checking Procedure

For every trace-property couple, ThEodorE's code generator produces an executable file that, once run, verifies the assumptions made in the property declaration against the values contained in the trace.

Listing 7.13 shows the output file for the property-trace couple represented in Listing 7.9.

Listing 7.13: ThEodorE output file

```python
from z3 import *
import time
def property_01():
  start_time=time.time()
  z3solver=Solver()

  s=Int('s')

  timestamps=Array('timestamps', RealSort(), IntSort())
  signal_4=Array('signal_4', RealSort(), IntSort())
  signal_2=Array('signal_2', RealSort(), IntSort())

  z3solver.add(timestamps[ 0]==0)
  z3solver.add(signal_4[0]==100.0)
  z3solver.add(signal_2[0]==15.0)
  z3solver.add(timestamps[ 1]==3600000000)
  z3solver.add(signal_4[1]==100.0)
  z3solver.add(signal_2[1]==17.49957229540723)

  ...

  z3solver.add(timestamps[ 21]==75600000000)
  z3solver.add(signal_4[21]==5.0)
  z3solver.add(signal_2[21]==0.0)

  interval_s=And(s>0, s<21)
  conditions_s=And(signal_4[s]<1000, signal_2[s]>=-(15.27))
  z3solver.add(Not(ForAll([s], Implies(interval_s, conditions_s))))
  status=z3solver.check()
  print(status)

  print("--- %s seconds ---" % (time.time() - start_time))
  if status == sat:
    print("REQUIREMENT VIOLATED")
    return 0
  if status == unsat:
    print("REQUIREMENT SATISFIED")
    return 1
  else:
    print("UNDECIDED")
    return 2


if __name__ == "__main__":
  property_01()
```

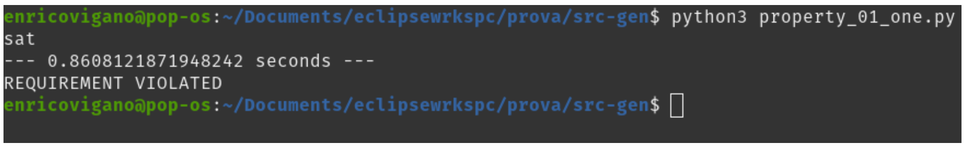
The proper verification script is contained in a function, which is defined with the same name as the property to verify, and is composed as follows:

- The different variables involved in the expression of the property are initialized in the first part of the script. `Timestamp` and `Num` variables are initialized as real numbers, while `Index` variables are initialized as natural numbers since they represent the indices of an array.

- The signals and the `timestamps` array are declared after the variables. Then for every index of the array, the code generator writes an assertion that declares the value of that specific record. Conflicts between these assertions and the ones declared by the property are what usually prompt Z3 to declare it unsatisfied.

- The property itself is declared in the last section in the Z3py logic. It is divided into different code snippets to increase legibility in the case of complex properties.

To run the script is sufficient to have a machine with Python and the Z3py library installed.



```
enricovigano@pop-os:~/Documents/eclipsewrkspc/prova/src-gen$ python3 property_01_one.py
sat
--- 0.8608121871948242 seconds ---
REQUIREMENT VIOLATED
enricovigano@pop-os:~/Documents/eclipsewrkspc/prova/src-gen$ 
```

Figure 7.3: Running the script with Python from a Linux terminal

In Figure 7.3 is depicted an example of successful execution of the script. The verdict yielded by ThEodorE can be `REQUIREMENT SATISFIED` , `REQUIREMENT VIOLATED` or `UNKNOWN` .

The execution time is also reported.

The script can be run by itself, but it can also be easily executed by other Python files since is wrapped in a function. This feature is very helpful to automatically verify large numbers of properties, as we did in the testing and evaluation phases of this thesis in Chapter 8.

7.6 Additional Options of Provided by ThEodorE

Below we describe some of ThEodorE' options.

Goal

The ".hls" file contains multiple properties and refer to multiple traces, but it always starts with a brief header section in which some options relative to the trace-checking process are selected by using a different keyword. (see Listing 7.14).

Listing 7.14: The header section of an ".hls" file

```
Goal: generate
Sample_Step: fixed-min
```

The header section contains the options for the generation of the output file and the preprocessing of the traces: `Goal` has two options:

1. if Save is selected, when saving, the only file that will be modified will be the ".hsl".
2. if Generate is selected, on saving the traces will be preprocessed and the output files generated.

Sample Step

The second option is Sample_step: that has three options:

1. Writing fixed-min instructs ThEodorE to use the smallest sample-step in the original trace as a fixed sample-step when preprocessing the trace, interpolating to replace missing values if necessary.
2. If fixed-manual is selected instead, on saving, the traces will be preprocessed using a fixed sample-step, specified trace by trace.
3. Finally, variable will prompt ThEodorE to use the variable sample-step of the original trace, without any interpolation in the preprocessing.

Listing 7.15: Property declarations in a ".hls" file

```

property_01::=
{
  Signal signal_4 Interpolation Constant;
  Signal signal_2 Interpolation Linear;
  Index s;
  Requirement::='signal 4 must always stay under the 1000
      threshold and signal 2 must be greater or equal than
      -15.27';
  Specification::=ForAll Index s In (0,FinalIndex): (signal_4
      (@index s)<1000 And signal_2(@index s)>=(-15.27));
}

property_02::=
{
  Signal signal_7 Interpolation Linear;
  Num c;
  Timestamp t;
  Requirement::= 'In the first 10 hours of operation, signal
      7 must stay around a value c between -200 and 200,
  with a maximum error of 10';
  Specification::=Exists Value c In ((-200),200):
  ForAll Timestamp t In (0, 10 [h]): (signal_7(@timestamp t)<
      c +10 And signal_7(@timestamp t)> c-10);
}

...
```

An example of a portion of the ".hls" file is shown in Listing 7.15. The property declaration is made by four different parts:

1. The name chosen for the property, followed by `::=` and by the rest of the declaration between curly brackets;
2. The variable declarations, using different keywords that may be used multiple times and a different name for every one of the variables:
   - `Num` for initializing simple numeric variables;
   - `Timestamp` for timestamp variables;
   - `Index` for index variables;
   - `Signal` for selecting a particular signal from the trace: the name of the signal in the declaration and in the trace must coincide;
3. The optional keyword `Requirement::=` followed by a string that describes the requirement, to make the subsequent code easier to interpret;
4. The keyword `Specification::=` followed by the property expressed in HLS;

The properties expressed this way may be assigned to one or more traces and also left unassigned without causing any error.

Listing 7.16: Trace assignment in a ".hls" file

```

Trace one 'trace1.tsv' SampleStep= 1 [h]
{
  Properties={property_01, property_03, property_06}
}

Trace two 'trace2.tsv' SampleStep= 3 [h]
{
  Properties={property_02, property_04, property_05}
}
```

The last section, represented in Listing 7.16, is dedicated to the coupling of the specifications with the traces they need to be verified on.

Every ".tsv" file containing an unprocessed trace produced is given a name and subjected to an arbitrary number of properties. On a syntactic level, the keyword `Trace` is followed by the name assigned to the trace by the user and then by a string containing the path to the file.

For every trace-property couple assigned this way, ThEodorE will produce an output script that will verify, once run, if the requirement expressed in the property is verified by the trace's behaviour.

The script's name will be the combination of the name of the trace and the requirement, which are both unique identifiers.

Measuring units can also be used to better quantify `Time` variables in HLS specifications; the notations are explained in Table 7.1.

Table 7.1: Units of measure for the sample-step and their respective symbols

Symbol	Unit
$[h]$	hours
$[m]$	minutes
$[s]$	seconds
$[ms]$	milliseconds
$[micros]$	microseconds
$[nanos]$	nanosenconds

Then, within curly brackets, the user can assign the set of properties to verify employing this notation `Properties::=` followed again by curly brackets containing the names of the desired properties separated by commas. LuxSpace

Chapter 8

Evaluation

In this Chapter, we report on the evaluation of our contributions. First, we evaluate the expressiveness of HSL, and compare it with state-of-the-art specification languages (Section 8.1). Second, we evaluate the applicability of the ThEodorE trace checker, and compare it to state-of-the-art tools (Section 8.2).

Specifically, we aim to answer the following research questions:

RQ1 *To which extent can HSL express requirements from industrial CPS applications and how does it compare with state-of-the-art specification languages in terms of expressiveness?*

RQ2 *Can ThEodorE verify CPS requirements on real-world execution traces within practical time and how does it compare with state-of-the-art tools?*

8.1 Expressiveness of HSL (RQ1)

To answer RQ1, we collected a set of industrial CPS requirements expressed in plain English text, and verified whether they could be expressed in HSL and in other state-of-the-art specification languages.

Dataset. We considered 212 industrial requirements from our satellite case study, coming from three different sources:

$\mathcal{S}1$: 61 requirements were randomly selected from 745 requirements contained in the requirement specification document of the satellite on-board software (OBSW). Due to the prohibitive effort (more than 20 hours spanned across several working days) involved, both on our part

Table 8.1: Number of requirements expressible in each of the languages for each set of requirements.

	$\mathcal{S}1$	$\mathcal{S}2$	$\mathcal{S}3$	Total
HSL	61/61	101/101	50/50	212/212 (100%)
SB-TemPsy-DSL	34/61	92/101*	19/50	145/212 (68%)
STL	38/61	51/101*	13/50	102/212 (48%)

and that of the domain experts who helped us formalize these requirements, we could only process a subset. Such requirements mostly refer to the software dynamics of the satellite, as in "*When the satellite switches to "Idle Mode", the OBSW shall checkout the GPS, wait* 50 ms*, and then checkout the sun sensors*".

$\mathcal{S}2$: 101 requirements were provided by the authors of SB-TemPsy-DSL [27]. They mostly refer to the physical dynamics of the satellite, as in "*the beta angle [5] shall show an oscillatory behavior with a maximum period of* 2500 s".

$\mathcal{S}3$: 50 requirements were extracted from the design and architectural documents of the satellite. These documents describe the relations and interactions among the different components of the satellite. They contain cyber-physical requirements that relate the software and the physical dynamics of the satellite, as in "*if the satellite mode switches from "Idle Mode" to "Safe Spin Mode" and the satellite is not in eclipse, the magnetic field recorded by the magnetometer shall contain a spike with a maximum amplitude of* 0.02 T".

Methodology. We tried to express the requirements from our dataset using HSL and two state-of-the-art specification languages, namely SB-TemPsy-DSL [27] and STL [57]. We selected these languages because they are both supported by trace checking tools. We assessed the extent to which requirements were expressible in each language.

Results. Table 8.1 reports[1] the number of requirements that we were able to express in each of the languages, for each set of requirements ($\mathcal{S}1$, $\mathcal{S}2$, and $\mathcal{S}3$). HSL was able to express 100% (212/212) of the requirements, while

[1]The values in Table 8.1 marked with an asterisk are slightly different from those reported in [27]. In the latter, quantification on real-valued variables (not supported in STL and SB-TemPsy-DSL) was handled by artificially selecting a value for the quantified variables within their quantification range. In this work, we marked such requirements as not specifiable.

SB-TemPsy-DSL and STL were able to express 68% (145/212) and 48% (102/212) of the requirements, respectively. These results confirm that HSL is highly expressive and much more so than alternatives. We remark that all the HSL constructs were useful to express at least some of the considered CPS requirements, though in very different proportions.

The answer to RQ1 is that HSL could express *all* the requirements of our case study, many more than SB-TemPsy-DSL and STL

8.2 Applicability of ThEodorE (RQ2)

To answer RQ2, we (i) assessed to which extent ThEodorE can be applied to check the execution traces of our case study; (ii) compared, in terms of applicability, ThEodorE with two other trace-checking tools: SB-TemPsy-Check [27] and Breach [35]. SB-TemPsy-Check is the trace checker for SB-TemPsy-DSL; Breach is a trace checker for STL. We chose Breach among other similar tools listed in a recent survey [20] (i.e., AMT [61, 61] and S-TaLiRo [16]), because AMT 2.0, in contrast to Breach, is not publicly available, and because Breach is faster than S-TaLiRo [35]. Furthermore, we excluded from our comparison tools tailored for online trace checking (e.g., SOCRaTEs [58] and RTAMT [62]).

Dataset. LuxSpace provided 20 traces, obtained by simulating the behavior of the satellite in different scenarios; the simulation time ranged from four to six hours. Their size (in number of entries) ranges from 41844 to 1202241 entries (avg = 389771, sd = 393718); the corresponding file size ranges from 1.7 MB to 58.9 MB (avg 17.6 MB, sd 19.4 MB). The traces have a considerably large (yet variable) number of records and size.

For each trace in our dataset, LuxSpace indicated which requirements to check. Indeed, since only a subset of the satellite signals is recorded in each simulation scenario, not all the requirements have to be checked on each trace. In total, we considered 747 trace-requirement combinations: 320 obtained from requirements in $\mathcal{S}1$, 178 obtained from requirements in $\mathcal{S}2$, and 249 obtained from traces in $\mathcal{S}3$. We remark that, out of these 747 combinations, 337 involve a requirement that can be expressed neither in SB-TemPsy-DSL nor in STL.

LuxSpace used a variable sample-rate for generating the trace records; hence not all the signal values were recorded at each sample index. Since our approach assumes that all the signals are assigned a value at each sample index, we pre-processed the traces. First, for each trace-requirement combination, we filtered out from the trace all the records that contained only signals that were not used in the HSL specification of the requirement. This step prevents the trace checker from handling an unnecessarily large set of records. Then, we transformed the traces using both pre-processing strategies $\mathcal{A}1$ and $\mathcal{A}2$ presented in section 5.2; in both cases, the interpolation function to use for each signal was indicated by the engineers of LuxSpace.

By applying the $\mathcal{A}1$ and $\mathcal{A}2$ strategies on the original 747 trace-requirement combinations, the final dataset contains 1494 trace-requirement combinations (with half of them obtained using one of the two strategies). The size of the traces obtained using $\mathcal{A}1$ ranges from 2 to 17321 entries (avg = 2071, sd = 3840); the corresponding file size ranges from 15 B to 5.9 MB (avg 0.1 MB, sd 0.4 MB). The size of the traces obtained using $\mathcal{A}2$ ranges from 2 to 2360674 entries (avg = 52406, sd = 185875); the file size ranges from 15 B to 90.0 MB (avg 2.3 MB, sd 8.4 MB).

Methodology. We ran ThEodorE over the 1494 trace-requirements combinations in our dataset. When translating the HSL properties in the target logic, we used function ι^V for the trace-requirement combinations generated using strategy $\mathcal{A}1$ (since the pre-processed traces have a variable sample rate), and function ι^F for those generated using strategy $\mathcal{A}2$ (since the pre-processed traces have a fixed sample rate).

We conducted our evaluation on a high-performance computing platform, using nodes equipped with Dell C6320 units (2 Xeon E5-2680v4@2.4 GHz, 128 GB).[2] Each run (checking a distinct combination of a trace and a property) was repeated 10 times, to account for variations in the performance of the HPC platform and of the SMT solver. In total, we executed $1494 \times 10 = 14940$ runs of ThEodorE. We allocated 4 GB of memory for each run and considered a timeout of one hour. We recorded whether the trace-checking procedure ended within the timeout, the trace checking result, and the time required to yield a verdict.

As for the comparison with SB-TemPsy-Check and Breach, we only considered the requirements from $\mathcal{S}2$ since it has the highest number of requirements expressible in SB-TemPsy-DSL and STL, and it was recently used for comparing SB-TemPsy-DSL with STL [27]. More specifically, we considered the 162 trace-requirement combinations (with requirements from the set $\mathcal{S}2$) expressible in SB-TemPsy-DSL, and the 103 trace-requirement combinations

[2]We executed our experiments on the HPC facilities of the University of Luxembourg [70].

Table 8.2: Output of ThEodorE (percentage and execution time) when using the pre-processing strategies $\mathcal{A}1$ and $\mathcal{A}2$.

	Output	%	*avg*	*min*	*max*	*sd*
$\mathcal{A}1$	*satisfied*	53.9	80.2	0.01	2693.0	334.7
	violated	12.1	14.2	0.01	513.9	57.9
	unknown	1.6	6.5	5.8	7.4	0.6
	timeout	0.5	-	-	-	-
	max_ depth_ exceeded	13.0	-	-	-	
	out_ of_ memory	18.9	-	-	-	
$\mathcal{A}2$	*satisfied*	53.8	102.5	0.01	3432.9	331.7
	violated	20.7	96.5	0.01	3143.5	379.8
	unknown	2.2	8.7	5.4	12.3	2.1
	timeout	23.3	-	-	-	-

expressible in STL. We ran the tools following the same methodology described above. Since each run was repeated ten times, in total we considered 1620 runs of SB-TemPsy-Check and 1030 runs of Breach.

Results - Applicability of ThEodorE. Table 8.2 shows the different types of output returned by ThEodorE for checking the 7470 trace-requirement combinations generated using the variable sample rate interpolation (row $\mathcal{A}1$) and the fixed sample rate interpolation (row $\mathcal{A}2$). Column "%" indicates the percentage of cases in which each type of verdict was returned. For each of the cases in which ThEodorE finished within the timeout (i.e., it yielded a *satisfied*, *violated*, or *unknown* verdict), Table 8.2 also provides the average (*avg*), minimum (*min*), maximum (*max*) and standard deviation (*sd*) of the ThEodorE execution time (s).

The results in row $\mathcal{A}1$ show that ThEodorE finished within the timeout in 67.6% of the cases. In 66.0% of the cases, ThEodorE produced a definitive verdict (i.e., *satisfied* or *violated*); in 0.5% of the cases, ThEodorE timed out. ThEodorE returned a "*max_ depth_ exceeded - maximum recursion depth exceeded during compilation*" error in 13.0% of the cases, and an "*out_ of_ memory*" error in 18.9% of the cases; both errors are generated by the Z3 solver. The root cause of these errors is the translation of function ι^V, used in the case of variable sample rate traces: the size of the arithmetic expression resulting from the translation is linear in the length of the trace. As expected, ThEodorE inherits the limitations of SMT solvers and its applicability is expected to improve along with the quick pace of progress in that field.

The results in row $\mathcal{A}2$ show that ThEodorE finished within the timeout in 76.7% of the cases. In 74.5% of the cases, ThEodorE produced a definitive verdict; in 23.3% of the cases, ThEodorE timed out. When using strategy $\mathcal{A}2$, the number of times ThEodorE reached the timeout was higher than when using $\mathcal{A}1$. Indeed, many trace-requirement runs that generated *max_ depth_ exceeded* and *out_ of_ memory* errors in the case of $\mathcal{A}1$, timed out when using $\mathcal{A}2$. As discussed for the case of $\mathcal{A}1$, the applicability of ThEodorE when using $\mathcal{A}2$ is determined by the scalability of the underlying SMT solver.

To evaluate whether ThEodorE is applicable in cases in which neither SB-TemPsy-Check nor Breach is applicable, we considered the subset of 3370 runs associated with the 337 trace-requirement combinations that involve a requirement that can be expressed neither in SB-TemPsy-DSL nor in STL. For those combinations, ThEodorE was able to produce a verdict in 67.9% of the cases.

To evaluate the impact of the trace accuracy (as determined by the application of the pre-processing strategies $\mathcal{A}1$ and $\mathcal{A}2$) on the correctness of the trace-checking procedure, we considered the 449 runs in which ThEodorE returned a definitive verdict both when using $\mathcal{A}1$ and when using $\mathcal{A}2$, and we compared the verdicts. In 95.1% of the cases (427 over 449), the verdicts coincided. For the 22 cases in which the verdicts were different, we manually inspected the generated traces and confirmed that differences in verdicts were caused by the pre-processing strategies.

Overall, these results show that ThEodorE, when configured with the pre-processing strategy based on a fixed sample rate ($\mathcal{A}2$), produced a definitive verdict for a considerable number of trace-requirement combinations (74.5%), thus confirming ThEodorE's applicability in practical scenarios. Relying on the $\mathcal{A}2$ strategy led to a significantly wider applicability of ThEodorE than with the $\mathcal{A}1$ strategy (74.5% vs 66.0%), while resulting in negligible differences in trace accuracy. Therefore, for comparing ThEodorE with other tools, we resorted to using the $\mathcal{A}2$ pre-processing strategy.

Results - Comparison with other tools. Table 8.3 reports the percentage of cases in which ThEodorE, SB-TemPsy-Check, and Breach provided a verdict within the timeout and the minimum, maximum, average and standard deviation of the time required to yield the verdict.

The results show that, when the requirements are expressible in SB-TemPsy-

Table 8.3: Comparison of ThEodorE, SB-TemPsy-Check, and Breach in terms of the execution time.

Tool	%	*avg*	*min*	*max*	*sd*
ThEodorE	72.2	69.6	0.01	2506.2	317.6
SB-TemPsy	94.1	30.1	0.09	3440.0	310.1
ThEodorE	95.1	81.4	0.01	2506.2	345.7
Breach	100	0.03	0.02	0.1	0.007

DSL and STL, SB-TemPsy-Check and Breach are faster than ThEodorE. However, given the usage scenario considered in our work (offline trace checking), the difference in execution times reported in Table 8.3 does not have significant practical consequences since the average trace-checking time (less than two minutes) is significantly lower than the time required to collect the traces (several hours). Note that all tools were consistent in terms of verdicts: when ThEodorE returned a definitive verdict, it matched the verdict returned by SB-TemPsy-Check and Breach (when they did not time out).

The answer to RQ2 is that ThEodorE could compute a definitive verdict, within one hour, for 74.5% of the trace-requirement combinations of our industrial case study, and produced a verdict for 67.9% of the 337 trace-requirement combinations that could not be checked by the other tools.

8.3 Discussion and Threats to Validity

Based on results, we recommend the following workflow. Developers should initially use ThEodorE since its language (HSL) is the most expressive, and it is generally difficult to know in advance which requirement types engineers will need to specify. If the property to be verified does not contain the `t2i` HSL operator, which causes the generation of large arithmetic expressions, engineers should use ThEodorE with the pre-processing strategy based on a variable sample rate ($\mathcal{A}1$). If the property contains the `t2i` operator, engineers should use the pre-processing strategy based on a fixed sample rate ($\mathcal{A}2$). If ThEodorE was not able to produce a definitive verdict, and the requirement is expressible in SB-TemPsy-DSL or STL, engineers should use SB-TemPsy-Check or Breach.

Threats to validity. The requirements and traces we used in our evaluation come from a single case study in the satellite domain. Although this could influence the generalization of our results, our industrial case study is representative of what can be found in other cyber-physical domains, where the system requirements are complex properties related to the software system, its environment and their interactions, and traces are obtained by simulating (or executing) the behavior of the CPS in many different scenarios.

Chapter 9

Conclusions

Software verification and validation requires specification-driven trace-checking techniques that strike a balance between the expressiveness of the specification language and the efficiency of its trace-checking procedures. In this paper, we specifically address this problem in the CPS domain.

- We proposed the Hybrid Logic of Signals (HLS), a specification language tailored to the specifics of CPS requirements. HLS allows engineers to specify complex CPS requirements related to its cyber and the physical components, as well as their interactions.
- We developed ThEodorE, an efficient SMT-based trace-checking procedure for HLS.

We evaluated our solutions through a large-scale, complex industrial case study involving an on-board satellite system. Results show that our approach achieves a better trade-off between expressiveness and performance than existing solutions.

- HLS was able to express all system requirements in contrast to existing languages. As a result, ThEodorE supports a much wider set of property types than other trace checkers.
- In most cases, ThEodorE was able to check those properties within practical time limits. Furthermore, the applicability of ThEodorE is expected to improve in the future along with the underlying SMT technology.

Last, based on results, we suggest a way to effectively combine various trace-checking tools.

As part of future work, we plan to develop trace diagnostics methods for HLS, inspired by existing work [36, 40], to explain the violations found by ThEodorE.

Another desirable pursuit would be offering support to the engineers employing ThEodorE by determining what factors contribute to the undecidability of some specifications, to better delimit the applicability of its approach.

We also plan, through an ongoing work of verification of industrial properties, to extend the grammar of HLS with more construct amenable to practical use in the field, such as the ones described in Section 5.5.

Bibliography

[1] SMT-COMP 2020. https://smt-comp.github.io/2020/.

[2] SMT-LIB The Satisfiability Modulo Theories Library. http://smtlib.cs.uiowa.edu/about.shtml.

[3] Testing the Untestable: Model Testing of Complex Software-Intensive Systems | TUNE Project | H2020 | CORDIS | European Commission.

[4] ACM SIGPLAN - Programming Languages Software Award. http://www.sigplan.org/Awards/Software/, 07 2020.

[5] Beta angle. https://en.wikipedia.org/wiki/Beta_angle, 2020.

[6] ETAPS 2018 Test of Time Award. https://etaps.org/about/test-of-time-award/test-of-time-award-2018, 07 2020.

[7] Luxspace. https://luxspace.lu/, 2020.

[8] Python. https://www.python.org/, 2020.

[9] Satellite development phases. https://www.esa.int/Science_Exploration/Space_Science/Building_and_testing_spacecraft, 2020.

[10] Xtend. https://www.eclipse.org/xtend/, 2020.

[11] Xtext. https://www.eclipse.org/Xtext/, 2020.

[12] Xtext documentation. https://www.eclipse.org/Xtext/documentation/index.html, 2020.

[13] Z3. https://github.com/Z3Prover/z3, 2020.

[14] Z3py. https://github.com/Z3Prover/z3/wiki/Using-Z3Py-on-Windows, 2020.

[15] Rajeev Alur. *Principles of cyber-physical systems*. MIT Press, 2015.

[16] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. S-taliro: A tool for temporal logic falsification for hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 254–257. Springer, 2011.

[17] Alexey Bakhirkin, Thomas Ferrère, Thomas A Henzinger, and Dejan Ničković. The first-order logic of signals: keynote. In *International Conference on Embedded Software*, page 1. IEEE, 2018.

[18] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[19] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard - version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017.

[20] Ezio Bartocci, Jyotirmoy Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Ničković, and Sriram Sankaranarayanan. Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In *Lectures on Runtime Verification*, pages 135–175. Springer, 2018.

[21] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to Runtime Verification. In *Lectures on Runtime Verification. Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 1–33. Springer, February 2018.

[22] David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zălinescu. Monitoring metric first-order temporal properties. *Journal of the ACM (JACM)*, 62(2):1–45, 2015.

[23] Marcello M Bersani, Matteo Rossi, and Pierluigi San Pietro. A logical characterization of timed (non-) regular languages. In *International Symposium on Mathematical Foundations of Computer Science*, pages 75–86. Springer, 2014.

[24] Marcello Maria Bersani, Domenico Bianculli, Carlo Ghezzi, Srđan Krstić, and Pierluigi San Pietro. SMT-based checking of SOLOIST over sparse traces. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 8411, pages 276–290. Springer, 2014.

[25] Domenico Bianculli, Carlo Ghezzi, Srđan Krstić, and Pierluigi San Pietro. Offline trace checking of quantitative properties of service-based applications. In *International Conference on Service Oriented Computing and Application (SOCA)*, pages 9–16. IEEE, November 2014.

[26] Chaima Boufaied, Maris Jukss, Domenico Bianculli, Lionel Claude Briand, and Yago Isasi Parache. Signal-based properties: Taxonomy and logic-based characterization. *arXiv preprint arXiv:1910.08330*, 2019.

[27] Chaima Boufaied, Claudio Menghi, Domenico Bianculli, Lionel Briand, and Yago Isasi-Parache. Trace-checking signal-based temporal properties: A model-driven approach. In *International Conference on Automated Software Engineering (ASE 2020)*. IEEE, 2020.

[28] Davide Bresolin. HyLTL: a temporal logic for model checking hybrid systems. *arXiv preprint arXiv:1308.5336*, 2013.

[29] Lionel Briand, Shiva Nejati, Mehrdad Sabetzadeh, and Domenico Bianculli. Testing the untestable: model testing of complex software-intensive systems. In *International Conference on Software Engineering Companion*, ICSE '16, New York, NY, USA, 2016. ACM.

[30] Adrien Champion, Arie Gurfinkel, Temesghen Kahsai, and Cesare Tinelli. Cocospec: A mode-aware contract language for reactive systems. In *International Conference on Software Engineering and Formal Methods*, pages 347–366. Springer, 2016.

[31] Alessandro Cimatti, Marco Roveri, and Stefano Tonetta. HRELTL: A temporal logic for hybrid systems. *Information and Computation*, 245:54 – 71, 2015.

[32] 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 *International Symposium on Temporal Representation and Reasoning (TIME)*, pages 166–174. IEEE, 2005.

[33] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[34] Alexandre Donzé. Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems. In *Computer Aided Verification*, pages 167–170. Springer, 2010.

[35] Alexandre Donzé, Thomas Ferrère, and Oded Maler. Efficient robust monitoring for STL. In *Computer Aided Verification Conference (CAV)*. Springer, 2013.

[36] Wei Dou, Domenico Bianculli, and Lionel Briand. Model-driven trace diagnostics for pattern-based temporal specifications. In *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 278–288. ACM, 2018.

[37] E Allen Emerson and Joseph Y Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM (JACM)*, 33(1):151–178, 1986.

[38] Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. A stream-based specification language for network monitoring. In *Runtime Verification*. Springer, 2016.

[39] Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Hazem Torfah. Stream-

lab: Stream-based monitoring of cyber-physical systems. In *Computer Aided Verification*, pages 421–431. Springer, 2019.

[40] Thomas Ferrère, Oded Maler, and Dejan Ničković. Trace diagnostics using temporal implicants. In *ATVA*, Cham, 2015. Springer.

[41] Thomas Ferrère, Oded Maler, and Dejan Ničković. Mixed-time signal temporal logic. In *Formal Modeling and Analysis of Timed Systems*, pages 59–75. Springer, 2019.

[42] Aaron W Fifarek, Lucas G Wagner, Jonathan A Hoffman, Benjamin D Rodes, M Anthony Aiello, and Jennifer A Davis. SpeAR v2. 0: Formalized past LTL specification and analysis of requirements. In *NASA Formal Methods Symposium*, pages 420–426. Springer, 2017.

[43] Eclipse Foundation. Eclipse IDE 2020-06 | The Eclipse Foundation.

[44] Khouloud Gaaloul, Claudio Menghi, Shiva Nejati, Lionel C. Briand, and David Wolfe. Mining assumptions for software components using machine learning. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pages 159–171, New York, NY, USA, 2020. ACM.

[45] Lars M. Garshol. Bnf and ebnf: What are they and how do they work? `https://www.garshol.priv.no/download/text/bnf.html`, 08 2008.

[46] Dimitra Giannakopoulou, Thomas Pressburger, Anastasia Mavridou, and Johann Schumann. Generation of formal requirements from structured natural language. In *Requirements Engineering: Foundation for Software Quality (REFSQ 2020)*, pages 19–35. Springer, 2020.

[47] Felipe Gorostiaga and César Sánchez. Striver: Stream runtime verification for real-time event-streams. In *Runtime Verification*. Springer, 2018.

[48] Richard Gronback. Eclipse Modeling Project | The Eclipse Foundation.

[49] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. *IEEE transactions on software engineering*, 18(9):785–793, 1992.

[50] Constance Heitmeyer. Requirements specifications for hybrid systems. In *International Hybrid Systems Workshop*, pages 304–314. Springer, 1995.

[51] Constance Heitmeyer, James Kirby, and Bruce Labaw. The SCR method for formally specifying, verifying, and validating requirements: Tool support. In *International Conference on Software Engineering (ICSE)*. ACM, 1997.

[52] Thomas A Henzinger, Zohar Manna, and Amir Pnueli. Towards refining temporal specifications into hybrid systems. In *Hybrid systems*, pages 60–76. Springer, 1992.

[53] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299, 1990.

[54] Jaekwon Lee, Seung Yeob Shin, Shiva Nejati, Lionel C. Briand, and Yago Isasi Parache. Schedulability Analysis of Real-Time Systems with Uncertain Worst-Case Execution Times. *arXiv:2007.10490 [cs]*, July 2020. arXiv: 2007.10490.

[55] University of Luxembourg. Home. `https://wwwen.uni.lu`.

[56] University of Luxembourg. SnT. `https://wwwen.uni.lu/snt`.

[57] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166. Springer, 2004.

[58] Claudio Menghi, Shiva Nejati, Khouloud Gaaloul, and Lionel C. Briand. Generating automated and online test oracles for simulink models with continuous and uncertain behaviors. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE. ACM, 2019.

[59] Claudio Menghi, Christos Tsigkanos, Thorsten Berger, and Patrizio Pelliccione. PsALM: specification of dependable robotic missions. In *International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 99–102. IEEE, 2019.

[60] Claudio Menghi, Christos Tsigkanos, Patrizio Pelliccione, Carlo Ghezzi, and Thorsten Berger. Specification patterns for robotic missions. *IEEE Transactions on Software Engineering*, 2019.

[61] Dejan Ničković, Olivier Lebeltel, Oded Maler, Thomas Ferrère, and Dogan Ulus. AMT 2.0: qualitative and quantitative trace analysis with extended signal temporal logic. *International Journal on Software Tools for Technology Transfer*, pages 1–18, 2020.

[62] Dejan Ničković and Tomoya Yamaguchi. RTAMT: Online robustness monitors from STL. In *Proc. International Symposium on Automated Technology for Verification and Analysis (ATVA 2020)*. Springer, October 2020.

[63] André Platzer. Differential dynamic logic for hybrid systems. *Journal of Automated Reasoning*, 41(2):143–189, 2008.

[64] André Platzer. *Logical foundations of cyber-physical systems.* Springer, 2018.

[65] Andrè Platzer and Jan-David Quesel. Keymaera: A hybrid theorem prover for hybrid systems (system description). *Automated Reasoning*, pages 171–178, 2008.

[66] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, 1977.

[67] César Sánchez, Gerardo Schneider, Wolfgang Ahrendt, Ezio Bartocci, Domenico Bianculli, Christian Colombo, Yliès Falcone, Adrian Francalanza, Srđan Krstić, Joao M Lourenço, et al. A survey of challenges for runtime verification from advanced application domains (beyond software). *Formal Methods in System Design*, 54(3):279–335, 2019.

[68] Joshua Schneider, David Basin, Frederik Brix, Srđan Krstić, and Dmitriy Traytel. Adaptive online first-order monitoring. In Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza, editors, *Automated Technology for Verification and Analysis*. Springer, 2019.

[69] Seung Yeob Shin, Shiva Nejati, Mehrdad Sabetzadeh, Lionel C. Briand, and Frank Zimmer. Test case prioritization for acceptance testing of cyber physical systems: a multi-objective search-based approach. In *SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 49–60, New York, NY, USA, 2018. ACM.

[70] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos. Management of an academic hpc cluster: The ul experience. In *Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014)*, pages 959–967, Bologna, Italy, July 2014. IEEE.

[71] Chunhui Wang, Fabrizio Pastore, Arda Goknil, Lionel Briand, and Zohaib Iqbal. Automatic generation of system test cases from use case specifications. In *International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 385–396. ACM, 2015.

[72] Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz, and Giles Reger. The SMT competition 2015-2018. *J. Satisf. Boolean Model. Comput.*, 11(1):221–259, 2019.