NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

TOMÁS JORGE JARDIM FERNANDES RODRIGUES DA SILVA

Bachelor in Computer Science

# MONITORING ROAD TRAFFIC RULES WITH SPATIO-TEMPORAL PROPERTIES

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
Feb, 2022

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

# MONITORING ROAD TRAFFIC RULES WITH SPATIO-TEMPORAL PROPERTIES

TOMÁS JORGE JARDIM FERNANDES RODRIGUES DA SILVA

Bachelor in Computer Science

**Adviser:** Carla Ferreira
*Associate Professor, NOVA University Lisbon*

**Co-adviser:** André Matos Pedro
*Research Scientist, VORTEX CoLab*

**Examination Committee**

**Chair:** Armanda Rodrigues
*Associate Professor, NOVA University Lisbon*

**Rapporteur:** Alcino Cunha
*Assistant Professor, University of Minho*

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
Feb, 2022

**Monitoring Road Traffic Rules with Spatio-Temporal Properties**



This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.8.3) [24].

*Para a estrela dos Barreiros, a menina que estará sempre à janela.*

# Acknowledgements

This thesis was a very complex and challenging project that allowed me to grow professionally and personally. The development and accomplishment of it was only possible due to a third party. Therefore, I would like to make some acknowledgements.

Firstly, a big praise for NOVA School of Science & Technology, for receiving me back in 2015 and providing me the tools and opportunities to flourish throughout these years. To NOVA Lincs and VORTEX CoLab I am grateful you received me in with this project and taken me in.

To my supervisors, Professor Carla Ferreira and Research scientist André Matos Pedro a big thank you for giving me this opportunity, monitoring and guiding me during this whole process and most important believing in me. To Professor João Lourenço and Professor João Costa Seco that were also part of this whole process and helped me a lot. To all the Professors and colleagues that accompanied me on this journey.

To my family and friends who were always there for me providing guidance and the conditions for me to succeed. A special thanks to my parents and grandparents who have given me everything so I could become successful as a professional and specially as a human being. To my girlfriend that has been my safe harbor and who always believed in me. A big thank you all, from the bottom of my heart.

*"Tell me and I forget. Teach me and I remember. Involve me and I learn. " (Benjamin Franklin)*

# Abstract

As technology evolves our world becomes filled with ubiquitous systems. Yet, at some point, technology has to be tamed because safety-critical systems, if not meticulously designed and validated, may represent a threat to us and to our environment.

This work aims at the verification and validation of safety-critical systems, such as autonomous driving systems, using runtime monitoring methods. We studied the monitoring of spatio-temporal properties which cope with the evolution of a space model during the flow of time, and an automatic monitor generation mechanism.

The situation presented as a case study in this dissertation considers an autonomous driving system on an urban environment, where our purpose is to monitor safety-margins between vehicles, the road, and the surrounding environment. A set of informally written requirements for the system were formalized in a spatio-temporal logic. This allows for these expressions to be transformed into monitors, programs, that are able to observe the behaviours considering space and time. Finally, a tool was implemented which applies the monitoring method developed throughout this dissertation given a sequence of events captured by an autonomous driving simulation software.

In conclusion, we hope to be taking a step forward to making driving a more automated, pleasant, and safer experience through the verification and validation of autonomous driving systems.

**Keywords:** monitoring, verification, spatial-temporal properties, safety-critical systems, autonomous driving systems.

# Resumo

À medida que a tecnologia evolui o nosso mundo fica cada vez mais repleto de sistemas ubíquos. No entanto, a certo ponto é preciso domar esta tecnologia porque se sistemas críticos de segurança não forem meticulosamente desenhados e validados poderão representar perigo para nós ou para o nosso ambiente.

Este trabalho consiste na verificação e validação de sistemas críticos, por exemplo, sistemas de condução autónoma, usando métodos de monitorização em tempo de execução. Estuda-se a monitorização de propriedades espácio-temporais que lidam com a evolução do espaço ao longo do tempo e a geração de monitores de forma automática.

O cenário apresentado como caso de estudo nesta dissertação considera um sistema de condução autónoma num ambiente urbano onde o objetivo é monitorizar as margens de segurança entre os veículos, a estrada e o ambiente que os rodeia. Foram escritos vários requisitos em linguagem natural e depois formalizados numa logica espácio-temporal. Isto permite que estas expressões possam dar lugar a monitores, programas, que são capazes de observar os comportamentos de espaço e tempo. Por fim foi desenhada uma ferramenta que aplica o método desenvolvido nesta tese e integrada a ferramenta com o software de simulação de condução autónoma.

Para concluir, esperamos estar a dar um passo em frente para tornar a condução de veículos uma experiência mais automatizada, prazerosa e segura através da verificação e validação de sistemas de condução autónoma.

**Palavras-chave:** monitorização, verificação, propriedades espaço-temporais, sistemas críticos de segurança, sistemas de condução autónoma

# Contents

# List of Figures

# List of Tables

# List of Listings

# Acronyms

| | |
|---|---|
| $FOL_{\mathbb{R}}$ | First-order logic 13, 26, 28, 29, 38, 39, 40, 41, 42, 44, 46, 58 |
| $LTL \times MS^{\leq}$ | Linear Temporal Logic combined with MS xii, 4, 5, 21, 22, 24, 29, 30, 31, 32, 33, 34, 35, 38, 39, 40, 41, 42, 44, 45, 47, 58, 60, 66 |
| **AD** | Autonomous Driving 1, 26 |
| **ADS** | Autonomous Driving System 1, 2, 5, 6, 7, 8, 24, 25, 26, 30, 37, 48, 51, 56, 57, 58, 59, 60 |
| **CPSs** | Cyber-Physical Systems 5, 8, 23 |
| **LTL** | Linear Temporal Logic 2, 12, 14, 15, 16, 24, 42 |
| **RV** | Runtime Verification 1, 2, 5, 11, 12 |
| **SMT** | Satisfiability Modulo Theories 2, 3, 4, 5, 11, 24, 40, 43, 44, 47, 58 |
| **STL** | Signal Temporal Logic 23, 24 |
| **UML** | Unified Modeling Language 9 |

# 1 Introduction

The problem we face in the scope of this thesis is how to correctly implement a monitor that checks road traffic rules as spatio-temporal properties of an autonomous driving system. For this, we need to know what are the safety requirements within the safety-critical systems, how to define a logical (or specification) language which combines space and time, and how these requirements are formalized. After this, we will have to verify if these properties are satisfied or not, integrating them in our monitors along with input information from the system in order to get a final verdict. With this, we can assure that either the system adheres to the safety requirements or that it does not.

## 1.1 Context and Motivation

Safety-critical systems have been evolving enormously since the last few years, including the Autonomous Driving System (ADS). Autonomous Driving (AD) is the key part on which we are going to focus in this thesis. ADS and other embedded systems are the part of day to day life of several people, we can say that such critical systems are ubiquitous. Therefore, the correctness and validation of such systems is crucial to prevent malfunctions as these lead to catastrophic events.

Challenges on the verification and validation methodologies for these systems are being introduced by complex AI methods. One main concerns in the design of such systems is that their product does not fail its purpose [17]. If we think further, no one designs a system to fail, but sometimes it is just beyond our reach or even understanding. The bottom line is, failures happen! So what can we do about it? What can we do to prevent it or ease its consequences? Although it is not one hundred percent reliable since a monitor may lead to an incorrect verdict, which in turn leads to an incorrect action or lack of it [26], verification is our best shot to keep a system correct and secure during its lifetime. Software designers use many known verification techniques in different scenarios to keep their computation running smoothly. Runtime Verification (RV) is a complementary verification method commonly used in safety-critical systems [15] due to its capability of being performed during runtime, offering the possibility to act whenever

a fault is observed. It uses monitors to check if the target system is compliant. Linear Temporal Logic (LTL) is perhaps the most common formal logic used in RV [15] and one of most used formal languages to describe reactive systems. Despite the fact that LTL only considers temporal behaviours, this work combines temporal and spatial behaviours. Combining both behaviours allows us to describe properties on ADS that require the specification of the evolution of spatial constraints (e.g., regions) over a flow of time. This evolution introduces the concept of time and projection of objects in different instances of the flow of time. The requirements we mean to monitor need to express region constrains and relations between objects in their space model for different instances of that discrete linear time. Road conventions can be formalized as spatio-temporal requirements, as they normally represent safety behaviours over space and time. Safety requirements normally dictate a state that must always hold, or that in the case of a particular event must hold in order to reduce the risk of malfunction of the system.

With this formalization we are able to produce monitors that validate the properties under a simulated environment. These monitors produce their own verdict indicating if the property is maintained under the assumption of a sequence of events that represent the observations on the simulated environment during its runtime. If the requirement is satisfied the produced verdict is positive, otherwise it produces a negative verdict, informing the system that a error has occurred since the property was violated during runtime.

## 1.2 Problem description

Given a scenario of a vehicle driving under an urban intersection as seen in Figure 1.1, the vehicle must comply with the road safety-margins and rules defined by the Vienna Convention [29] and Portuguese road traffic rules [30].

We need to correctly implement a monitor that checks spatial-temporal properties of our system that we define as being our traffic rules. These rules represent safety-margins between vehicles, the road and the surrounding environment as well as behaviours the vehicles should adhere in that situation. The statement is how we could automatically implement a monitor (source-code) that checks the spatio-temporal properties described using a logic fragment with enough expressive power to represent these properties on a ADS. These properties shall be formally defined using a logic fragment that combines LTL with metric spaces and then subjected to a Satisfiability Modulo Theories (SMT) in order to define the satisfiability of such formula. At the end what we want is to check whether the formula that defines a certain property is satisfactory or not throughout the execution of the simulated environment of our system.

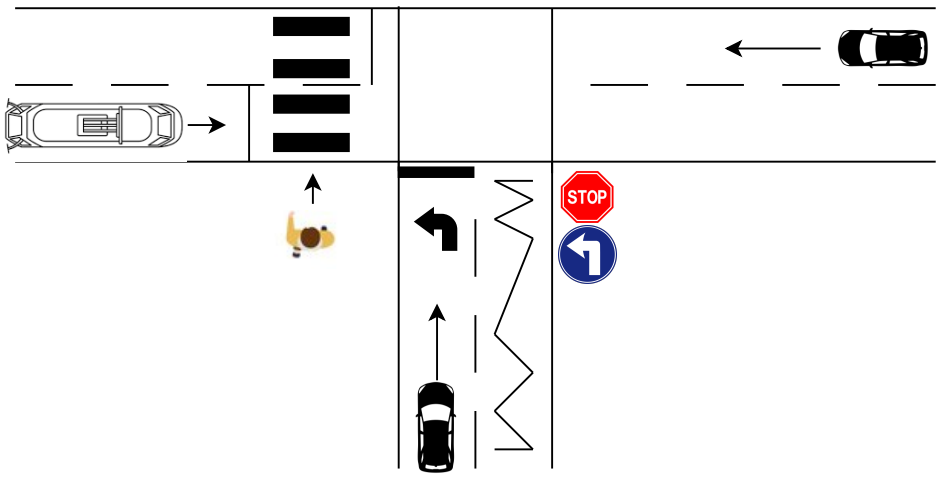



Figure 1.1: Scenario layout

### 1.2.1 Solution

In first place, we will formalize four known traffic rules (from the Vienna convention [29] and the Portuguese road code [30]) using a logic fragment that can reason about topological and metric spaces, combined with temporal logic. We will exploit these rules that represent road-safety requirements on a intersection as the one in Figure 1.1. Then, these logical expressions are converted into the input language of an SMT solver. This conversion allows the use of the SMT solver to check the validity of each property that has been formalized under the assumption of our specific traffic scenario and a trace representing the spatial evolution of the dynamic objects in our scenario along a flow of time. Finally, our tool automatically generates monitors that check the properties we want while feeding these monitors new information from a closed-loop testing scenario with an autonomous driving simulator.

## 1.3 Running example

Our use case is composed by several actors (e.g., vehicles, pedestrians, tram) inside an urban road intersection. The scenario in which we test our approach is a 'T' shaped junction between two roads where vehicles, pedestrians, and trams circulate, as seen in Figure 1.1. Extra road signs and marks of our scenario include a stop sign, pedestrian cross, and a box junction. Moreover, it is known that the lane represented vertically is a one-way-only lane, and the vehicles on it must only turn left in the junction.

In our scenario, the two roads are represented by the space in between the thin black horizontal and vertical lines. The pedestrian cross is represented as the black horizontal lines crossing the horizontal road. The box junction is the square in the intersection of the two roads.

Consider that we want to verify with our tool a simple road safety requirement such

as the one in which the car must stop when it reaches a stop sign, and then carry on when the path is clear in the scenario described above. We may see this requirement as a spatio-temporal property as we can apply the following (informal) translation: whenever the car reaches a stop sign, its next projection, i.e., the position occupied by the car in the next moment of the timeline, has to be the same as the position in the current moment, which implies that the car has indeed stopped at the stop sign. And then, if the car has already stopped at the stop sign (looking at its past projections) it may carry on.

It may be obvious that every human driver understands, respects and complies to this *"stop at the stop sign"* rule, but our first problem is how can we transform these rules so they can be monitored, which we address with the formalization of the requirement in a formal language. After the formalization of the traffic rule, the monitor must check the validity of the formula combined with the scenario's static elements information and the incremental trace information which comes from the simulator. The trace is a sequence of events captured by the simulator that indicates a set of attributes of the dynamic objects in our scenario for example the position of the car in each event. A relevant scenario element information in this case is the position and region occupied by our stop sign, that we interpret as a rectangle crossing the vertical road as seen in Figure 1.1. This information together with the position of the car in every instance of our trace is what allows us to validate our property since we want to check if the car stops once it arrives at the region where there is a stop sign. We will be able to know this relying on the distance between the car and the stop sign and verifying if the car stops within a predefined margin.

In order to check the validity of the property, we use an algorithm that translates the formula in $LTL \times MS^{\leq}$ combined with the trace and scenario metadata into a model interpretable by the SMT solver. This algorithm was meticulously designed with respect to efficiency.

Finally, the system must comply with the rule, thus the monitor produces a verdict indicating if the property is being maintained or not while receiving new observations from the simulator. For example, if the car reaches the stop sign but does not stop, our verdict will be false. If it reaches the stop sign and stops we may conclude the rule was satisfied and therefore the verdict will be true.

## 1.4 Contributions

With the problem in hand we can enumerate our contributions in a sequential order so as to reach our final goal which is building a framework that generates runtime monitors to check safety rules in our system and integrating it in a Autonomous Driving software.

1. Analyze the appropriate formalism for representing traces in the form of metadata.

2. Formalization of traffic road conventions in $LTL \times MS^{\leq}$.

3. Define a algorithm that translates a formula from $LTL \times MS^{\leq}$ into an SMT model accepted by Z3.

4. Generate a monitor that receives the SMT solver's model together with the trace information and validates the model.

5. A tool that generates runtime monitors from a specification language (spatio-temporal logic) to check traffic road conventions according to document [29].

## 1.5 Document Structure

In Chapter 2, we start by defining what is a Safety-Critical System and some examples with a focus on ADS and what are the challenges such systems hold out. The requirements and specifications of safety-critical systems help to understand the importance of the Verification and Validation (V&V) of such systems and the interest in adopting both spatial and temporal constrains. We also survey the different verification techniques that exist with focus on the RV and explain what is a monitor and how they can be used to check relevant ADS properties. We briefly describe SMT solvers and the solver we will use in our approach. We introduce the formal specification languages studied for the possible formalization of our properties. We start with a brief introduction to first-order logic over real variables. Then we introduce the logic notion over time an immediately after the logic notion over space. Since we are looking to formalize spatio-temporal properties we introduce the combination of these two worlds and we talk about the syntax and semantics of the fragment we use to define our properties. We finalize the chapter with a discussion of the related work over monitoring of safety-critical and Cyber-Physical Systems (CPSs) and over the combination of temporal and spatial logic and their complexity.

In Chapter 3 we introduce the traffic rules we proposed to monitor and demonstrate the encoding for our empirical scenario on which the rules mean to apply. We will monitor our properties under the analysis of a trace and we will introduce how this trace is dismantled for the encoding of the dynamic objects. Finally we show how we formalized our traffic rules making usage of our specification language.

In Chapter 4 we present the detailed approach we follow for the monitoring of our properties. Following the approach in we present our algorithm which transforms a formal logic expression into a model interpreted by our solver and how it will be used to provide us the verdict on our monitor.

In Chapter 5 we present how we evaluate and intend to evaluate our tool. We present our two evaluation methods used for evaluating our tool and define the metrics and success criteria for those evaluations. We analyse the type of traces we use for testing our tool, where we include empirical and simulated traces. At last we present the result obtained by evaluating our traces using both methods we described and do a brief analysis of those results.

Finally, in Chapter 6 we present our conclusion and what we were able to accomplish. We analyse possible future work for optimizing our tool as well as implementing it in real time ADS.

# 2

# Background and related work

In this chapter we survey the background in three main areas, safety critical system definition, common features, requirements and a closer look at the Autonomous Driving System (ADS), system verification techniques adequate for safety-critical systems, and finally we give a thorough understanding of formal languages adequate for describing our properties towards spatial-temporal behaviours. We recap that we intend to construct a tool that can generate monitors for ADS verification and validation and that ADS are in their essence safety-critical systems.

## 2.1 Safety Critical Systems

In this section we will discuss about Safety-critical systems since it is the type of systems we are looking to monitor, the examples and the tests are based on an ADS which itself is a safety-critical system.

Safety-critical systems are systems in which a failure may compromise the purpose of the system itself or even threaten any form of life or lead to property or financial damage. These systems are becoming enormously common and ubiquitous in our daily lives due to the constant improvement in technology.

Mission-Critical systems, a more general notion of safety-critical systems, are those systems which have critical properties that may compromise the main mission of a system. "Such systems generally have failure rate requirements ranging from $10^{-5}$ to $10^{-9}$ failures per hour" [15]. By this quote we may conclude that all mission-critical systems have to have very few failures and maintain the system operational due to the possible failures outcomes, making the verification and validation of such systems an important aspect to have in mind when building these.

Nowadays many systems around the world are controlled by computers and if some of them fails completely in their mission, with no setbacks, may lead to disasters as loss of lives, environment damage, financial loss, or information leaks. Traditional systems that we might immediately think about being safety-critical are our medical care system,

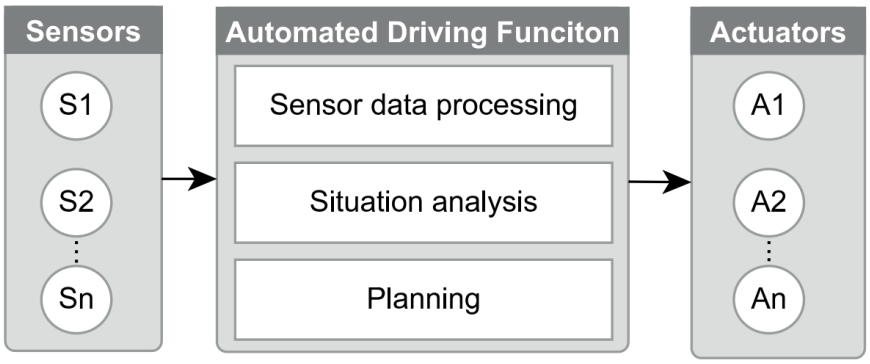



Figure 2.1: Generalized ADS block diagram [26]

aircraft, and nuclear power. But we can consider many other systems, mostly Cyber-Physical Systems (CPSs), that are also embedded systems such as smart homes, smart cities, or ADS. Even non-embedded systems as a bank, may compromise your information and your finances therefore may be categorised as safety-critical systems [17].

ADS is a field of study that is part of CPSs which can mostly be seen as safety-critical due to the high impact a hazard can have [32]. Correctness and validation of an ADS is crucial, as any error or malfunction of the system may lead to loss of life, property damage, or financial loss, this may happen, for example, in the unfortunate case of a car crash due to the systems malfunction [26]. An ADS is a computer-based system embedded in vehicles for a more comfortable and automated driving experience. It depends on various sensors that read the car's environment to provide input for the computations to occur. These computations may then act directly on the vehicle with actuators (e.g., steering, braking, signaling danger) [26]. An ADS may be categorised in many levels defining the autonomy level of these actuators. Being that the last level, level 5, defines a fully autonomous driving system. In Figure 2.1 we can see a generalization of an ADS.

### 2.1.1 Challenges

As technology improves, computers are getting more omnipresent in our society. The result is that serious consequences of failures arise for all these systems being of a physical nature or through removal of service or damage to information.

To maintain a safe environment around these systems, techniques that provide high levels of assurance of non-interference will be required. This makes the monitoring and verification of these systems a very important aspect to improve from a software perspective. As well as improvement in areas such as specification, architecture and process [17]. However in this thesis we will focus on improving verification of these systems. In order to have confidence in the correction and safety of the variety of safety-critical systems flourishing around the world, rapid and efficient verification technique

approaches are fundamental to ensure the good implementation and execution of such systems [17]. To put it all together it is important to guarantee that these type of software systems work in a correct, secure, reliable matter because life may depend on it [20].

### 2.1.2 Requirements and Specification

The requirements and specification of a critical system are important to determinate which correctness properties are inevitable to keep our systems gears spinning with the counter effect of the endangerment of the system itself or for its failures putting in danger human life.

Having a broad awareness of the system and its specification is crucial to predict where the failures may arise, which is essential for constructing a good verification layer for that system. Hazard and risk analysis is a common initial approach in order to find the systems requirements. Traditional design processes such as the V-model may also be used to create a functional requirements document [15]. This model starts by defining the users needs decomposing them to the core, and ends with a user-validated system using integration and verification. From these requirements documents a system specification can be created and used as the basis for the system design [15].

Focusing on the correct system specifications may be crucial to maintain a safe operative system. Therefore it is a crucial part of system design to find the correct set of system requirements because it can be equally dangerous building a system with an incorrect or incomplete set of requirements as it is to monitoring a system with incorrect specifications that may lead to a technically positive report but practically useless results [15].

There are a variety of formal and informal languages for specifying requirements documentation. Informal specification languages are more user friendly but sometimes lead to incorrect requirements. An example of a semi-informal language is Unified Modeling Language (UML). Formal specifications are unambiguous and verifiable for their correctness and enable other formal methods, including software verification. The conjunction of formal specification and lightweight analysis to verify correctness properties is known to be effective[13]. Again, one of the main objectives of this thesis is to design a tool based on a lightweight formal method for system verification and therefore a suitable and complete requirements documentation is needed.

### 2.1.3 Hazard Analysis

Hazard analysis on the other point is the method of finding these *hazard* states of a system during its development lifecycle. "A *hazard* is a system state or set of conditions together with other environmental conditions of the system that can lead inevitably to an accident" [21]. The identification of a systems safety requirements and safety cases may be found using a hazard analysis which is the primary step of safety engineering. These safety requirements found throughout a hazard analysis are the requirements to be monitored on the verification layer to prevent a system failure. Therefore, we can conclude that

hazard analysis is important to determine which requirements are worth monitoring in a system. The lack of a safety requirement property to monitor may be as dangerous as monitoring useless properties in a system with incorrect specifications. Many different hazard analysis techniques are used to find different stages of safety requirements in a system [21].

## 2.2 System Verification

System verification is a software architecture layer to have in account during software design. It is composed of many different verification techniques that define and maintain the correctness of a program. It basically ensures that the system maintains its design specification. The first step in defining this verification layer is to determine which properties to check in our system and how we are going to check them.

As explained above, we need, somehow, to define what we want to check and how. Instrumentation is used to retrieve information from the system, it can be done at runtime or before running the program. That information retrieved is then used to produce events. Events can then be of three types: *signals*, *changes of state*, or *description of the current state*. Events can be functions or blocks of code that interact with the state of a program, this interaction is how we can separate the events in those three types. When instrumenting a sequential program, the trace is a sequence of events. For our tool we will use events containing the description of the current state. A trace only contains events important to check the property being tested ignoring all the others. Therefore, a trace is an abstraction of the useful parts of the execution of a program for the property in cause.

### 2.2.1 Properties

Properties are mathematical objects, in our case, represented as logic formulas, that are used to check if a trace or sequence of events complies with a specification. A specification is a description of something that must happen during the execution of the program. Properties can be of several types such as:

- *Hyper-Properties*: Must specify exactly the allowed sets of execution points.
- *Safety Properties*: Must hold true at all points of the execution dictating that something bad can never happen.
- *Liveness Properties*: Should hold true at some point of the execution. Represent a "good thing" that should happen infinitely often.

We have to have in account that *liveness* properties aren't always traceable in partial executions since the "good thing" might still happen in the future [2]. As we are working on monitoring a safety-critical system we will be focusing on safety properties as they are more commonly used in for representing safety requirements in these types of systems.

Besides being less common, liveness properties may also be used when monitoring this type of systems.

### 2.2.2 Static vs. Dynamic Verification

There are three main verification techniques:

- *Theorem proving* that allows proving the correctness of programs;
- *Model checking* which is an automatic verification technique based on algorithms that check if given logic formulas hold in a specification [34]; and
- *Testing* that covers methods for showing the correctness of a program and finding bugs.

A static verification technique verifies the correctness of the program's design and implementation without executing the program, checking the syntax and semantics of the code.

Satisfiability Modulo Theories (SMT) refers to the problem of defining the satisfiability of a first-order formula within some logic theory, i.e., defining if a formula is satisfiable and therefore can be used in model-checking as a property. The more expressive is the logic language more complicated is it to SMT solvers to check the satisfiability of its formulas [4]. SMT is used instead of SAT when there are further restrictions in the semantics of a language, i.e., when they are more expressive. It generalizes SAT by interpreting certain symbols to models of logical background theories, e.g., equality, arrays, lists, quantifiers and more. SMT-LIB is the standardization and benchmark collection of SMT.

Z3 is the solver used on the tool application created for this dissertation as it can be called procedural by many API and uses new quantifier and theory combination algorithms [27]. For a better understanding of Z3's system architecture check [27].

As model checking was developed in the context of temporal logic it is very useful and powerful when applied to temporal models, specially when these represent finite structures, even if these display infinite behaviours. The fact of dealing with finite structures allows the decidability of model checking as the non-finiteness of behaviours allows us to model interesting situations [34]. A dynamic verification technique verifies the correction of a program while it is running. It has to keep track of the different states of a system while ascertaining it. An example of a dynamic verification technique is runtime verification that has its root in model checking. Testing is also considered a dynamic verification technique as it requires the program to be running for it to check the correctness of the methods.

### 2.2.3 Runtime Verification

Runtime Verification (RV) is considered to have its origins in model checking. What distinguishes RV from model checking is the nature of it being performed during runtime,

offering the possibility to act whenever a fault is observed before it becomes a failure. Therefore, if something applies to model-checking it also applies to RV. There is online and offline RV, online deals with a runtime, live verification, as the offline deals with verification from recorded logs. RV is a verification technique especially used in systems that depend on environment behaviour and hardware like the known embedded systems. Model checking and theorem proving are used in more simplified environment models [15].

A software failure is when the observed behaviour is not the required behaviour of the system for that situation. While a fault is a when the current behaviour is different from the expected one [20].

**Definition 2.2.1** *[20] Runtime verification is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property.*

In RV, a correctness property is translated into a monitor which is used to check if the target system complies with its requirements. Linear Temporal Logic (LTL) is perhaps the most common formal logic used in RV [15]. There is one mismatch between these two however, LTL expressions iterate over an infinite time flow where RV deals with finite and infinite traces. There have been many attempts of solutions for this problem like defining finite trace semantics for LTL so it can be able to represent inconclusive values that happen when looking at finite prefixes of infinite traces [15]. RV deals with finite sequences of events also known as traces that are produced using instrumentation. The fact that it verifies finite traces brings out verdict inconclusive problems when dealing with liveness properties since RV cannot act on properties that take infinite amount of time to check. These properties may be checked using model checking for example that may compliment our RV to obtain a more safe and complete verification [15].

### 2.2.4 Runtime Monitoring

As in RV, monitors may be used to check the current execution of the system (online monitoring) or work on logs (offline monitoring). Monitors can be an expensive part of the system and have to be taken in count. They have to be made to influence the system as less as possible.

The job of a monitor is to check a correctness property throughout a trace to deliver a verdict on that correctness property. A verdict, in its simplest form delivers either true or false but in certain systems one may consider extra outputs to the verdict to make it more precise. However, there are only two type of correct conditions a runtime monitor can experience, true positive or true negative [26]. A true positive is a positive detection of a fault in an incorrectly operating system, i.e., when a runtime monitor correctly identifies that a faulty system is at fault. A true negative is a negative detection of a fault in a

correctly operating system, i.e., when a runtime monitor correctly identifies that non-faulty system is not at fault [26].

We then have two types of error conditions, false positive and false negative. A false positive is a positive detection of a fault in a correct system, i.e., when a runtime monitor falsely concludes that a non-faulty system is at fault. False negative is a negative detection of a fault in an incorrect system, i.e., when a runtime monitor wrongly concludes that a faulty system is not at fault [26].

We can measure the correctness of our verification comparing the values of true positives and true negatives with false positives and false negatives. Needless to say that in safety-critical systems like ours it is of top priority to keep the rate of false positives and false negatives as low as possible [26].

A monitor has two maxims it should adhere, impartiality and anticipation [20].

- *Impartiality* requires that a finite trace is not evaluated too soon, i.e., is not classified to true or, respectively false, if there still exists a continuation of the trace leading to another verdict.

- *Anticipation* requires that the evaluation is done as soon as possible respecting impartiality, i.e., once every continuation of a finite trace leads to the same verdict, then the finite trace evaluates to this very same verdict.

This means for a monitor to be impartial then it can only decide for false/true if and only if that decision cannot change during the rest of the execution of a program. So, for a monitor to be more correct it should also have, at least, one more truth value, inconclusive, or unknown, for situations where you can not predict the future of a correctness property. Impartiality and anticipation, guarantee that the monitor is neither premature nor overcautious in its evaluations [20].

## 2.3 Formal Specification Languages

A formal specification language is a logic language that contains a well-defined syntax and semantics that can express properties of systems such as temporal order of states, timing of events, and counting time. We will select one of these languages to encode our environment and describe our requirements [6].

### 2.3.1 First-order logic over real numbers $FOL_{\mathbb{R}}$

First-order logic uses quantified variables over terms and allows the use of expressions that contain variables. These expressions represent predicates that are evaluated to either True or False. First-order logic ($FOL_{\mathbb{R}}$) over real numbers consists on first-order-logic over the structure $\langle \mathbb{R}, <, +, x, 1, 0 \rangle$ where terms are represented using real numbers and predicates as inequalities or equations of expressions over real numbers. This means that all constants belong to $\mathbb{R}$ and the variables represent real numbers as well. Following

Tarski's theorem, [36] the first-order theory of real-closed fields is complete and decidable and accepts quantifiers [31]. A real-closed field is seen as mathematical field that has the same first-order properties as the field of real numbers therefore being able to represent all real numbers with the structure $\langle \mathbb{R}, <, +, x, 1, 0 \rangle$. This allows us to build first-order expressions with quantifiers using real numbers for example a first-order expression such as

$$(\exists x, y, z)(x * y = 0.54) \wedge (z + y < 1.50).$$

### 2.3.2 Temporal Logic

Temporal logic represents a logic tool to reason about the temporal order of states, events or symbols in many different areas. Here, we are interested on computer science applications where temporal logic is widely adopted in the specification and validation of programs [34]. In model checking and runtime verification, this tool is highly used to formulate the properties we want to verify. We can have many different types of temporal logic, propositional versus first-order, global versus compositional, branching versus linear time, point versus intervals, discrete versus continuous, and past versus future [8].

However, the one that makes more sense using for our application is a propositional, global, point-based, discrete linear time, future tense logic. What each of these axes means is as follow [8]:

- *Propositional* - The non temporal part of the formulas are build up with truth functional connectives as $\vee$ , $\wedge$ and $\neg$.
- *Global* - The temporal operators refer to a single universe.
- *Point-based* - This means every temporal moment can be pinpoint to a specific timestamp.
- *Discrete* - This means the present moment represents the programs current state and the next moment represents the programs actual successor state, and so on.
- *Linear time* - This means the representation of time is linear, i.e., it is a unique sequential timeline. One moment in time has only one possible next moment.
- *Future tense* - This means the temporal part of the formulas only represents future tense operators allowing us to reason about the future.

#### 2.3.2.1 LTL

LTL is a linear-time temporal logic [1], and a suitable logic for reasoning about reactive systems. Ideally applicable in model checking, and also in runtime verification. The flow of time in LTL is an ordered set $(S, <)$ with time-points $s$ belonging to the set $S$ and a precedent relation $<$ that indicates the strict order in which each time-point occurs [8]. An LTL formula $\varphi$ can be constructed by known Boolean operators and the temporal

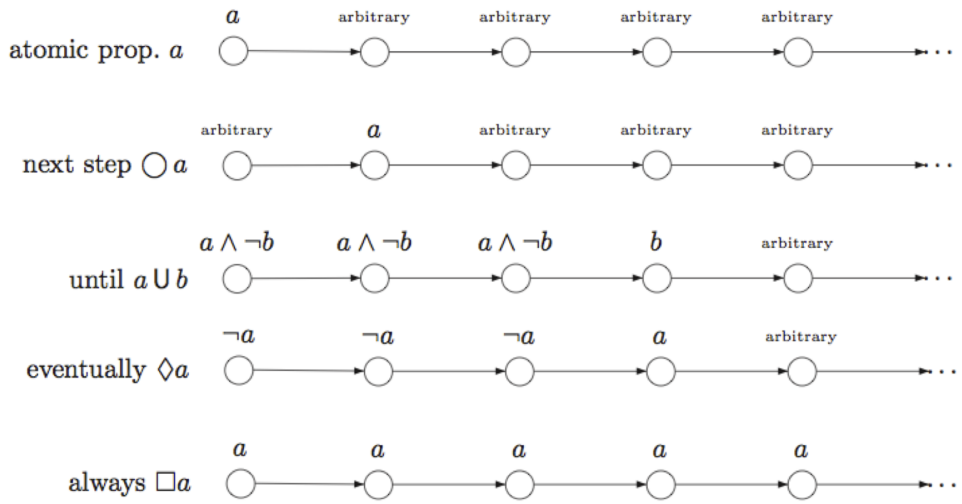



Figure 2.2: Temporal operators in LTL [28]

operator $\mathcal{U}$ that express a sequence of propositional variables. Each atomic proposition is a formula [12]. $\mathcal{U}$ stands for until and is inductively defined such as $\varphi_1 \mathcal{U} \varphi_2$ where $\varphi_1$ and $\varphi_2$ are also formulas. Informally, it reads as "$\varphi_1$ holds true until $\varphi_2$ holds true". From the operator $\mathcal{U}$, we can define other temporal connectives such as $\Diamond\phi$ (sometimes in the future), $\Box\phi$ (always in the future), and $\bigcirc\phi$(next moment) [1]. These are useful shortands for writing succinct formulas in LTL as shown in the Figure 2.2. Note that $a$ and $b$ are atomic propositional variables. It is important to refer that the interpretation of $\mathcal{U}$ do not include the present time-point. One has to formulate the *always* and *eventually* operators such as $\Box\phi \wedge \phi$ and $\Diamond\phi \vee \phi$ to do so.

In order to evaluate a LTL formula we need to establish at which time-points our propositions hold. To do so, we define a structure for our model by

$$\mathfrak{M} = (\mathsf{F}, {p_0}^{\mathfrak{M}}, {p_1}^{\mathfrak{M}}, {p_2}^{\mathfrak{M}}, \ldots),$$

where ${p_i}^{\mathfrak{M}} \subseteq S$ for all i and $\mathsf{F}$ is the ordered time flow [1]. We define the truth relation $(\mathfrak{M}, s) \models \varphi$ or simply $s \models \varphi$ that is read as "the formula $\varphi$ holds at a time-point $s$ in model $\mathfrak{M}$" and is defined as follows:

- $s \models p_i \quad$ *iff* $\quad s \in {p_i}^{\mathfrak{M}}$;
- $s \models \varphi \mathcal{U} \mu \quad$ *iff* there is $v > s$ such that $v \models \mu$ and $u \models \varphi$ for all $u \in S$ and $s < u < v$.

The truth relation for other operators can be defined by

- $s \models \bigcirc\varphi \quad$ *iff* $\quad s+1 \models \varphi$ where s + 1 is the immediate successor to moment $s$;
- $s \models \Diamond\varphi \quad$ *iff* there is $v > s$ such that $v \models \varphi$;
- $s \models \Box\varphi \quad$ *iff* for all $u \in$ S and $u > s$, $u \models \varphi$.

A formula $\varphi$ is *satisfiable* if there is a model $\mathfrak{M}$ over $(\mathbb{N}, <)$ and a time point $n \in \mathbb{N}$ such that $(\mathfrak{M}, n) \models \varphi$. We say that $\varphi$ in *finitely satisfiable* if there is a finite strict linear order $\mathfrak{F}$ and a model $\mathfrak{M}$ such that $(\mathfrak{M}, n) \models \varphi$ for some $n$ in $\mathfrak{F}$ [1].

We can also add past operators such as *since* to our LTL and according to the Kamp theorem[14], the propositional temporal language with both *until* and *since* is as expressive as the monadic first-order language over $(\mathbb{N}, <)$.

LTL always operator is useful to describe safety properties, as they dictate something that must happen at all points of an execution. The eventually operator in the other hand is usefull for describing liveness properties as they dictate that something good should happen at some point of an execution.

### 2.3.3 Spatial Languages

There are three main types of modal logic for space, the ones that are build upon:

- *Topological spaces* that reasons over topological relations between objects, i.e., direct relations between the interior **I** of objects from a universe **U**. So it is a pair $(\mathbf{U}, \mathbf{I})$ that can represent all spatial relations in a universe [1]. Other operators, such as the closure operator **C**, may be defined using **U** and **I**.

- *Metric spaces* that are able to reason about the distance between objects [1]. A metric space contains a nonempty set of points $\Delta$ and the relation $d$ between them $d : \Delta \times \Delta$ which is the distance between two points of the set. This allows metric spaces to induce topological spaces. A metric space $(\Delta, d)$ satisfy three axioms, as follows:

  1. $d(x, y) = 0$ *iff* $x = y$ *(identity of indiscernibles)*
  2. $d(x, y) = d(y, x)$ *(symmetry)*
  3. $d(x, z) < d(x, y) + d(y, z)$ *(triangle inequality)*

- *Distance spaces* are a more expressive form of metric spaces since it only satisfies the first axiom presented in the metric spaces being represented by the same pair $(\Delta, d)$.

Based on the problem described in section 1.2, we assume we need a spatial specification language that deals with regions and distances.

#### 2.3.3.1 Topological model

A well known topological space logic (topo-logic) is the modal logic $S4_u$ that can be seen as the more expressive from other languages such as RCC-8 and BRCC-8. Spatial terms on $S4_u$ are of the form:

$$\rho ::= p_i \mid \overline{\rho} \mid \rho_1 \sqcap \rho_2 \mid \rho_1 \sqcup \rho_2 \mid \mathbf{I}\rho \mid \mathbf{C}\rho,$$

where $p_i$ are spatial variables; $\overline{\rho}, \sqcap, \sqcup$ are the standard Boolean operators know as complement, intersection and union; **I** and **C** are the interior and closure topological operators.

To express how spatial terms relate to each other we make use of formulas that are either true or false. The common formulas to all spatial models are:

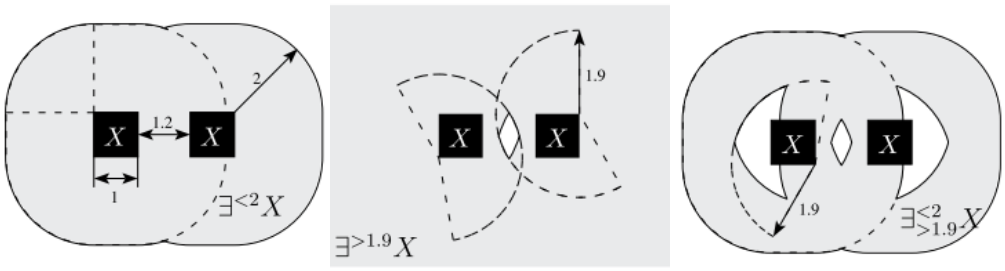
- $\rho_1 \sqsubseteq \rho_2$ which indicates that $\rho_1$ is a subset to $\rho_2$;
- $\neg\varphi$ which is the negation of formula $\varphi$;
- $\varphi_1 \wedge \varphi_2$ which is the conjunction of $\varphi_1$ and $\varphi_2$; and
- $\varphi_1 \vee \varphi_2$ which is the disjunction of $\varphi_1$ and $\varphi_2$.

We still include the constant terms $\top$ and $\bot$ that represent the whole space and the empty set respectively.

When we talk about regions, we mean a standard closed nonempty set of points that can be represented using $S4_u$ terms as $X = \mathbf{CI}X$. RCC-8 is the best-known language that reasons with regions instead of points. It has an '8' on its name because of the eight predicates in its syntax that can be combined with Boolean operators [38]:

- $DC(X_1, X_2)$ *iff* $\neg\exists x, x \in X_1 \cap X_2$ (disconnected),
- $EC(X_1, X_2)$ *iff* $(\exists x, x \in X_1 \cap X_2) \wedge (\neg\exists x, x \in \mathbf{I}X_1 \cap \mathbf{I}X_2)$ (externally connected),
- $PO(X_1, X_2)$ *iff* $\exists x, x \in \mathbf{I}X_1 \cap \mathbf{I}X_2 \wedge \exists x, x \in \mathbf{I}X_1 \cap \neg X_2 \wedge \exists x, x \in \neg X_1 \cap X_2$ (partially overlap),
- $EQ(X_1, X_2)$ *iff* $\forall x, (x \in X_1 \leftrightarrow x \in X_2)$ (equal),
- $TPP(X_1, X_2)$ *iff* $\forall x, x \in \neg X_1 \cup X_2 \wedge \exists x, x \in X_1 \cap \mathbf{C}\neg X_2 \wedge \exists x, x \in \neg X_1 \cap X_2$ (tangential proper part),
- $NTPP(X_1, X_2)$ *iff* $\forall x, x \in \neg X_1 \cup \mathbf{I}X_2 \wedge \exists x, x \in \neg X_1 \cap X_2$ (nontangential proper part),
- $TPP_i(X_1, X_2)$ *iff* $TPP(X_2, X_1)$,
- $NTPP_i(X_1, X_2)$ *iff* $NTPP(X_2, X_1)$.

However, it is not a very expressive language because it only operates in simple regions, not being able to perform unions ($\cup$) or intersections ($\cap$) between regions, and is strict to its binary predicates [38]. As referred to above, the RCC-8 is embedded in S4u. Therefore, it can be translated into the terms and formulas of S4u. There are extensions of RCC-8 that include more expressive power namely the BRCC-8 that allows boolean combinations of region variables [38].




(a) $\exists^{<2}X$ (b) $\exists^{>1.9}X$ (c) $\exists^{<2}_{>1.9}X$

Figure 2.3: Region metric distance example in region consisting of two black boxes [1]

#### 2.3.3.2 Distance model

As defined above, spatial logic over metric and distance spaces include the notion of distances between objects. We can still use topo-logics like $S4_u$ or $RCC-8$ in the topological space induced by the distance space $\mathbf{D} = (\Delta, d)$ . By this we can introduce new distance reasoning to the ones used in topological model using bounded quantifiers that represent distance operators. Given the model

$$\mathbf{M} = (\mathbf{D}, {p_0}^{\mathfrak{M}}, {p_1}^{\mathfrak{M}}, {p_2}^{\mathfrak{M}}, \dots)$$

we can introduce these operators as

1. $(\exists^{<a}\rho)^{\mathbf{M}} = \{x \in \Delta \mid \exists y(d(x,y) < a \wedge y \in \rho^{\mathbf{M}}\}$ (somewhere at distance $< a$),
2. $(\exists^{>a}\rho)^{\mathbf{M}} = \{x \in \Delta \mid \exists y(d(x,y) > a \wedge y \in \rho^{\mathbf{M}}\}$ (somewhere at distance $> a$),
3. $(\forall^{<b}_{>a}\rho)^{\mathbf{M}} = \{x \in \Delta \mid \forall y(a < d(x,y) < b \rightarrow y \in \rho^{\mathbf{M}}\}$ (everywhere within distance d for $a < d < b$), etc.

Note that that each item has an illustration in Figure 2.3 (from left to right).

MS is a full modal logic of distance spaces introduced in (Kutz et al., 2003) [19] that include the previously mentioned operators. The spatial terms of this logic are as follows [18]:

$$\rho ::= p_i \mid \{l_i\} \mid \overline{\rho} \mid \rho_1 \cap \rho_2 \mid \exists^{=a}\rho \mid \exists^{<a}\rho \mid \exists^{>a}\rho \mid \exists^{<b}_{>a}\rho,$$

where $\{l_i\}$ represents the set of single points that represent location constants. However, this full modal logic of distance spaces is not decidable containing the "donut" operator $\exists^{<b}_{>a}$ [1]. Therefore variations of $MS$ without the "donut" operator are more likely to be used for example the $MS^{\leq,<}$ introduced by (Wolter and Zakharyaschev, 2003) [37].

Distance spaces is what allows us to define the distance operators that for example for $\exists^{\leq a}\rho$ return the a-neighbourhood of $\rho$. This is only possible as our distance space contains the distance between every two points on a universe.

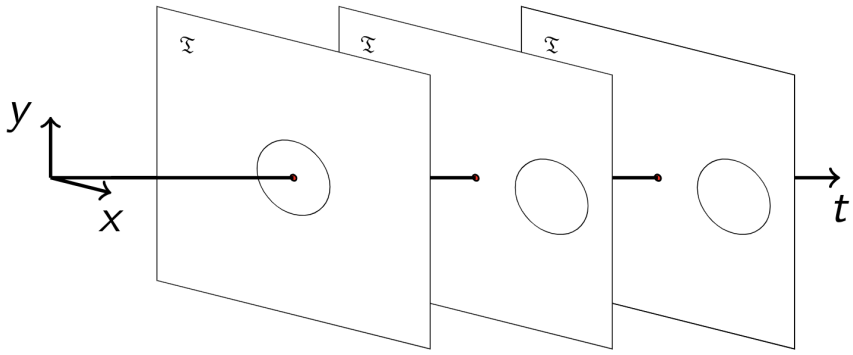



Figure 2.4: Example of a spatial-temporal demonstration of a ball moving then stopping

### 2.3.4 Combination of Temporal and Spatial Logics

To define some restrictions, a combination of both spatial and temporal formalisms is the most adequate if not necessary. Mainly when referring to safety requirements in an embedded system. This because several safety-critical systems contain safety requirements that have to be maintained throughout a linear time. As seen in Figure 2.4, the combination of both spatial and temporal logic is needed to prove that an object started moving and stopped. Without one of the logic the formalization of this scenario would not be possible.

The complexity proofs from D.Gabelaia, et. al. [10] show there decidable fragments of spatio-temporal logic over topological spaces however most are of high computational complexity.

For this thesis, we will try to combine two well-known spatial and temporal logics. However, to avoid leading to an undecidable formalism, we will define our spatio-temporal logics as a cartesian product of the intended timeline and the spatial model defined for our system [10]. Besides that we restrict our intended model to models where the same topological space is kept along the whole timeline [1].

The expressiveness of this combination depends on three parameters:

1. The expressive power of the spatial component;

2. The expressive power of the temporal component;

3. The allowed interaction between the two components.

The minimum requirement for a spatial-temporal combination is the possibility to express changes in time of the truth-values of purely spatial propositions (**PC**) [10, 1].

However with this minimum we cannot express the *evolution* of the spatial objects in time, for that we introduce two fundamental principles [1]:

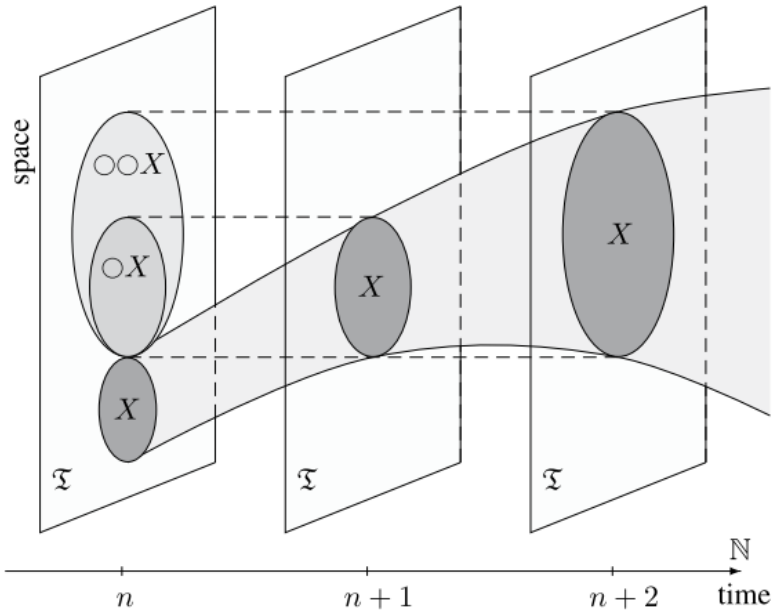



Figure 2.5: Temporal operators on regions [10]

- Local spatial object change principle (**LOC**) - The language should be able to express changes or evolutions of spatial objects over some fixed finite periods of time.
- Asymptotic spatial object change principle (**AOC**) - The language should be able to express changes or evolution of spatial objects over the whole duration of time.

As shown in Figure 2.5 we can see the denote of spatial evolution of region X with $\bigcirc X$ that represents the space occupied by X at moment n + 1. To represent statements where an object changes or moves over time **PC** is not enough, we need to have principles like (**AOC**) and (**LOC**) that refer to evolution [10].

A maximalist approach would be having a combination with no restrictions, agreeing to all principles. The most overreaching representation of this language would be $LTL \times S4u$. Its terms $\rho$ and formulas $\varphi$ are represented as follows:

$$\rho ::= p \mid \overline{\rho} \mid \rho_1 \cap \rho_2 \mid \rho_1 \cup \rho_2 \mid \mathbf{I}\rho \mid \mathbf{C}\rho \mid \rho_1 \mathfrak{U} \rho_2$$
$$\varphi ::= \rho_1 \sqsubseteq \rho_2 \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathfrak{U} \varphi_2$$

The appliance of this language is shown in [1]. An $LTL \times S4u$ formula $\varphi$ is satisfiable if there existes a topological temporal model $\mathfrak{M}$ such that $(\mathfrak{M}, n) \models \varphi$ for some time point $n \in \mathbb{N}$.

By restricting the terms like we can do with simple spatial logics we can obtain fragments of the $LTL \times S4u$ language. However, the satisfiablity of $LTL \times RCC8$ is not known but with BRCC8 a formula can be satisfied if in finite flows [1].

The combination of temporal logics with metric spaces has not got much study on it. However in [1] it is shown how the computational behaviour of what Linear Temporal Logic combined with MS ($LTL\times MS^{\leq}$) would look like and that it satisfies the (PC), (LOC) and (AOC) principles. Gabbay et al. [9] prove that the satisfiability problem for $LTL\times MS^{\leq}$ formulas in metric temporal models is $\sum_1^1$-complete. It is concluded that the satisfiability problem for $LTL\times MS^{\leq}$ terms is decidable but not in elementary time.

### 2.3.5 LTLxMS

The terms and formulas of $LTL\times MS^{\leq}$ are inductively defined as follows:

$$\rho ::= p \mid \overline{\rho} \mid \rho_1 \sqcap \rho_2 \mid \rho_1 \sqcup \rho_2 \mid \exists^{\leq a}\rho \mid \rho_1 \mathfrak{U} \rho_2 \qquad \text{(terms)}$$
$$\varphi ::= \rho_1 \sqsubseteq \rho_2 \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathcal{U} \varphi_2 \mid \varphi_1 \mathcal{S} \varphi_2 \qquad \text{(formulas)}$$

A metric temporal model [1] is a pair of the form $\mathfrak{M} = (\mathfrak{D}, \mathfrak{N})$, where $\mathfrak{D} = (\Delta, d)$ is a metric space. $\mathfrak{N}$, a valuation, is a map associating each spacial variable $p$ and time instant $n$ to a set $\mathfrak{N}(p,n) \subseteq \Delta$. The valuation can be inductively extended to arbitrary $LTL\times MS^{\leq}$ terms such as:

$$\begin{aligned}
\mathfrak{N}(\overline{\rho}, n) &= \Delta - \mathfrak{N}(\rho, n),\\
\mathfrak{N}(\rho_1 \sqcap \rho_2, n) &= \mathfrak{N}(\rho_1, n) \cap \mathfrak{N}(\rho_2, n),\\
\mathfrak{N}(\exists^{\leq a}\rho, n) &= \{x \in \Delta \mid \exists y (d(x,y) \leq a \wedge y \in \mathfrak{N}(\rho, n))\},\\
\mathfrak{N}(\rho_1 \mathfrak{U} \rho_2, n) &= \bigcup_{m>n} (\mathfrak{N}(\rho_2, m) \cap \bigcap_{k\in(n,m)} \mathfrak{N}(\rho_1, k)).
\end{aligned}$$

The examples in Figure 2.6 demonstrate the usage of some terms defined in our fragment where $\mathbb{X}$ represents $\Delta$.

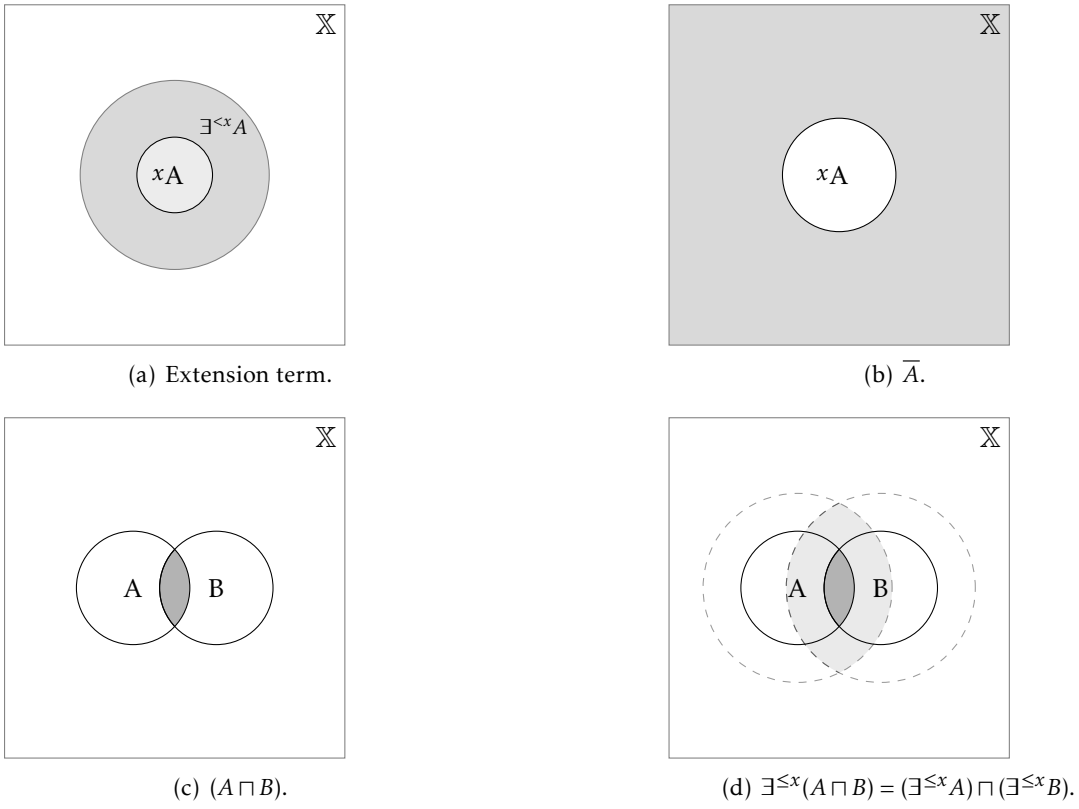
The shorthand $\bigcirc_\rho$, $\Diamond_\rho$, and $\Box_\rho$ are defined using $\mathfrak{U}$ as:

$$\Diamond_\rho = \top \mathfrak{U} \rho, \quad \Box_\rho = \neg \Diamond_\rho \neg \rho \quad \text{, and} \quad \bigcirc_\rho = \bot \mathfrak{U} \rho.$$

Or can be described as:

$$\begin{aligned}
\mathfrak{N}(\bigcirc_\rho, n) &= \mathfrak{N}(\rho, n+1),\\
\mathfrak{N}(\Diamond_\rho, n) &= \bigcup_{m>n} \mathfrak{N}(\rho, m), \text{ and}\\
\mathfrak{N}(\Box_\rho, n) &= \bigcap_{m>n} \mathfrak{N}(\rho, m).
\end{aligned}$$

The meaning of $\bigcirc_\rho$ is, at the moment $n$, it denotes the space occupied by $\rho$ at the next moment $n+1$. For $\Diamond_\rho$ it denotes that at a moment $n$, term $\Diamond_\rho$ is interpreted as the union of all spatial extensions of $\rho$ at moments $m > n$. For $\Box_\rho$ it denotes that at a moment $n$, term $\Box_\rho$ is interpreted as the intersection of all spatial extensions of $\rho$ at moments $m > n$. As seen in Figure 2.5 with $\mathfrak{U}$ we can operate over future projections of a specific region.




(a) Extension term. (b) $\overline{A}$.

(c) $(A \sqcap B)$. (d) $\exists^{\leq x}(A \sqcap B) = (\exists^{\leq x}A) \sqcap (\exists^{\leq x}B)$.

Figure 2.6: Examples of spatial term operators

The truth value of $LTL \times MS^{\leq}$ formulas in a model $\mathfrak{M}$ is inductively defined as follows [10]:

$$
\begin{aligned}
(\mathfrak{M}, n) &\models \rho_1 \sqsubseteq \rho_2 && \textit{iff} && \mathfrak{N}(\rho_1, n) \subseteq \mathfrak{N}(\rho_2, n), \\
(\mathfrak{M}, n) &\models \neg\varphi && \textit{iff} && (\mathfrak{M}, n)\varphi, \\
(\mathfrak{M}, n) &\models \varphi_1 \wedge \varphi_2 && \textit{iff} && (\mathfrak{M}, n) \models \varphi_1 \quad \textit{and} \quad (\mathfrak{M}, n) \models \varphi_2, \\
(\mathfrak{M}, n) &\models \varphi_1 \vee \varphi_2 && \textit{iff} && (\mathfrak{M}, n) \models \varphi_1 \quad \textit{or} \quad (\mathfrak{M}, n) \models \varphi_2, \\
(\mathfrak{M}, n) &\models \varphi_1 \,\mathcal{U}\, \varphi_2 && \textit{iff} && \text{there is a } m > n \text{ such that} (\mathfrak{M}, m) \models \varphi_2 \text{ and} \\
& && && (\mathfrak{M}, k) \models \varphi_1 \text{ for all } k \in (n, m), \\
(\mathfrak{M}, n) &\models \varphi_1 \,\mathcal{S}\, \varphi_2 && \textit{iff} && \text{there is a } m < n \text{ such that} (\mathfrak{M}, m) \models \varphi_2 \text{ and} \\
& && && (\mathfrak{M}, k) \models \varphi_1 \text{ for all } k \in (n, m).
\end{aligned}
$$

An $LTLxMS^{\leq}$ formula $\varphi$ is said satisfiable if there exists a model $\mathfrak{M}$ such that $(\mathfrak{M}, n) \models \varphi$ for some time point $n \in \mathbb{N}$.

For the temporal connectors we have $\Diamond$ for "eventually in the future", $\Box$ for "always in the future" and $\bigcirc$ for "at the next moment". These temporal connectors can be defined using $\mathcal{U}$ such as

$$\Diamond\varphi = \top\,\mathcal{U}\,\varphi, \quad \Box\varphi = \neg\Diamond\neg\varphi \quad \text{, and} \quad \bigcirc\varphi = \bot\,\mathcal{U}\,\varphi.$$

$\top$ and $\bot$ when referencing formulas represent the Boolean constants of True and False respectively. The dual operators are defined in a similar way via the operator "Since" ($\mathcal{S}$) so we have $\diamondminus\varphi = \top \mathcal{S} \varphi$ for "once in the past", $\boxminus\varphi = \neg\diamondminus\neg\varphi$ for "always in the past" and $\ominus\varphi = \bot \mathcal{S} \varphi$ for "the previous moment".

Note that the traditional universal modalities $\forall$ and $\exists$ are expressible in our language. $\forall\rho$ can be seen as an abbreviation for $(\top \sqsubseteq \rho)$ and $\exists\rho$ for $\neg(\rho \sqsubseteq \bot)$, where $\top$ and $\bot$ are constant terms denoting the whole space and empty set. We will also make use of the atomic formulas $\rho_1 = \rho_2$ and $\rho_1 \neq \rho_2$ standing for $(\rho_1 \sqsubseteq \rho_2) \wedge (\rho_2 \sqsubseteq \rho_1)$ and $\neg(\rho_1 = \rho_2)$.

With these shorthand's we can define four spatial patterns over region terms:

$$
\begin{aligned}
DC(\rho_1, \rho_2) &= (\rho_1 \sqcap \rho_2 = \bot) && \text{(disconnected)},\\
EQ(\rho_1, \rho_2) &= (\rho_1 \sqsubseteq \rho_2) \wedge (\rho_2 \sqsubseteq \rho_1) && \text{(equally connected)},\\
O(\rho_1, \rho_2) &= \neg(\rho_1 \sqcap \rho_2 = \bot) \wedge \\
&\quad \neg(\rho_1 \sqsubseteq \rho_2) \wedge \neg(\rho_2 \sqsubseteq \rho_1) && \text{(overlapped)},\\
I(\rho_1, \rho_2) &= (\rho_1 \sqsubseteq \rho_2) \wedge \neg(\rho_2 \sqsubseteq \rho_1) && \text{(included)}.
\end{aligned}
$$

Note that $I(\rho_1, \rho_2)$, $\rho_1$ is included in $\rho_2$ but $\rho_2$ is not included in $\rho_1$.

## 2.4 Related Work

In [32] there is a review of various paper that approach the assessment of safety-critical systems, in particular advanced driver-assistance system (ADAS). Kapinski *et al.* [16] describe several practical methods for using simulation traces to perform verification and other quality checking activities on their embedded control systems, from testing and falsification, to verification with the most scenario-based techniques that check only concrete scenarios.

As our specification language offers great expressiveness we are able to make formal verification on a huge set of system behaviours, however in [16] Kapinski proves there is a lack of scalability for complex systems.

In [32] they separate the verification methods in three categories but most papers using this technique use the formalization of traffic rules as the key requirement, such as us. Aréchiga [3] uses a formal language to define a set of contracts that enable multiple formal techniques such as automatic generation of runtime monitors, similar to what we do.

Like us Shalev-Shwartz *et al.* [35] describe safety requirements using axioms and lemmas and produce a formal proofing dictating whether a accident may be caused or not. However, their approach targets only the trajectory of an autonomous vehicle.

Tengfei Li et al. [22] presents a spatio-temporal specification language combining Signal Temporal Logic (STL) with spatial logic $S4_u$ to characterize dynamic behaviours of CPSs. They also conclude that there are few research on spatio-temporal modeling of behaviours of motion based spatially distributed systems mostly because the level of

expressiveness needed is an issue. They define their temporal logic on dense time associated with a set of real-valued signals therefore bounding their time domain. For example in STL it is not suitable to use the next operator as its only suitable for discrete time. This shows that for certain systems a discrete time is needed to describe certain behaviours, therefore the best specification language for each system may vary.

Bennett et al. [5] built a modal logic named PSTL combinig the Cartesian product of the temporal logic PTL and the modal logis $S4_u$ to specify the discrete time and general topological space. Gabelaia et al. [10] combines PTL and some fragments of modal logic $S4_u$ and testes their complexity.

Most take profit of the expressiveness of modal logic $S4_u$ and not many studies were found combining temporal logic with modal logic over metric spaces.

## Summary

We can assume automated driving systems as safety-critical systems or mission-critical systems based on the safety risks the system may lead. Therefore we focus on the requirements and specification of such systems in order to learn how to analyse and verify our ADS. We will be handling events that describe the current state of the system. The main type of property we will deal with are safety and liveness properties. Runtime verification will be our main verification technique, since we are dealing with a dynamic system that will be in constant change our verification must be done at runtime. We will use Z3 as the SMT solver to check the satisfiability of our formulas since we are prone to use the combination of spatial and temporal logic to define our properties which may lead to undecidable formulas. As the temporal logic of choice we have LTL as it describes time as a linear discrete flow, the ideal for representing a sequence of events. As to spatial logics we have to consider talking about regions, since the environment of the ADS can be mapped to regions, and distances, since the properties we want to check talk about safety-margins. In conclusion a combination of LTL with a spatial logic that can represent regions and distances is ideal to formulate the properties we want to check in our monitors, namely $LTL \times MS^{\leq}$.

# 3

# Traffic Rules, Scenario and Trace Formalization

We intent to monitor proprieties that are safety requirements on a ADS run on a simulator. Therefore, there is a need to formalize these safety requirements that, in turn represent safety traffic rules to be maintained in a urban road scenario. Besides formalizing the safety rules we will monitor, we also need to encode all objects from that scenario in order to be able to include expected behaviours between different objects in our formalization. Objects are entities such as pedestrians, cyclists, vehicles, trajectories, or horizontal/vertical traffic signs(e.g., crosswalk, stop sign).

## 3.1 Rules Overview

Within our ´T' junction scenario, there are several applicable road rules taken from the Vienna Convention of road traffic [29] and the Portuguese road code [30]. We will exploit some of these rules that represent the safety requirements for our use case.

Our safety requirements are:

- A vehicle should leave a safety-margin from the vehicle in front of it, based on article 13 of the Vienna convention;
- A vehicle should leave a safety-margin towards the side-walks;
- A vehicle should stop at a stop sign, based on the Portuguese road signals;
- A vehicle should not stop on top of:
  - a box junction, based on the Portuguese road marks M17b and article 18 of the Vienna convention;
  - pedestrian crossing, based on article 23 al.3 of the Vienna convention;
  - tram rails, based on article 23 al.3 of the Vienna convention.

Besides these conventional road safety rules that we establish as requirements for our ADS monitoring we add one more that is more of an Autonomous Driving (AD) road safety rule [23] that is: *A vehicle should maintain its predefined trajectory*.

## 3.2 Scenario Encoding

In Figure 3.1(a) we can see an example of the traffic scenario used throughout the use case. It consists of a T-shaped junction where the vehicle, noted as **C**, that approaches the intersection in a one-way road comes across a stop sign in order to enter a bi-directional road. In that bi-directional road there is a tram going one direction in its rails noted as **T** in the figure, and a car going the other direction noted as **C1**. In the junction of the two roads there is a box junction that according to the Portuguese road marks is a zone where it is prohibited to stop due to the probability of provoking a traffic congestion. Besides that there is pedestrian zebra cross in the bi-directional road. In Figure 3.1(a) it is also possible to see three different dotted lines noted as **T1**, **T2** and **T3** that represent the possible trajectories the vehicles can take in this specific use case.

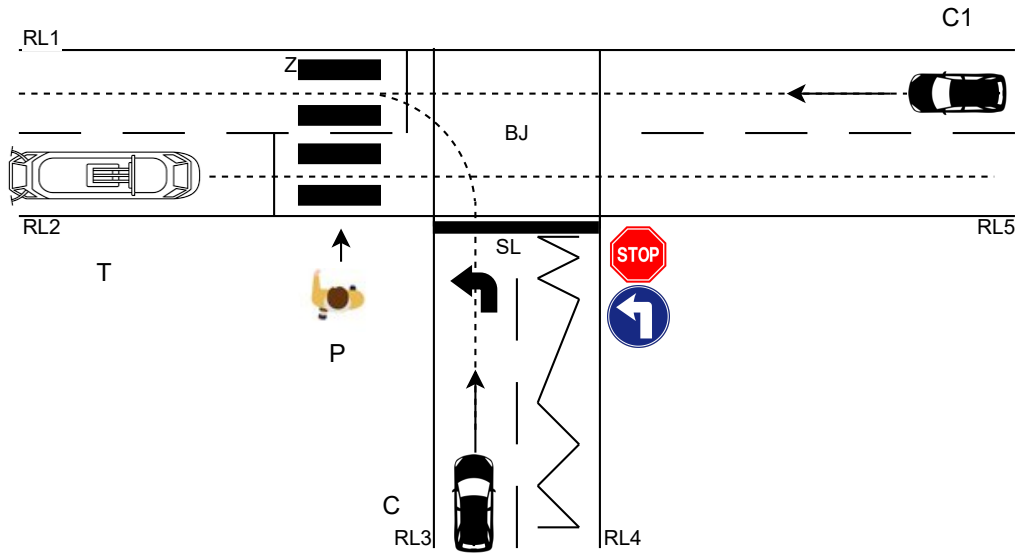
We formalize our scenario based on $FOL_{\mathbb{R}}$ that consists on the set of all well-formed sentences of first-order logic that involve quantifiers and logical combinations of polynomial expressions over real variables. The first-order language $FOL_{\mathbb{R}}$ forms the set $\mathfrak{L}$, and $\mathcal{P}$ means the set of real variables in $FOL_{\mathbb{R}}$.

We use different encoding for each element of the scenario, this encoding is based on a region restriction that we represent as inequalities. For the pedestrians and cars we use a circle inequality with radius $r$ while the tram, box junction and zebra cross are defined by bounding boxes. The trajectories and road limits are encoded as polynomials in order to form the lines we see in the figure. The encoded mapping of Figure 3.1(a) is presented in Figure 3.1(b).

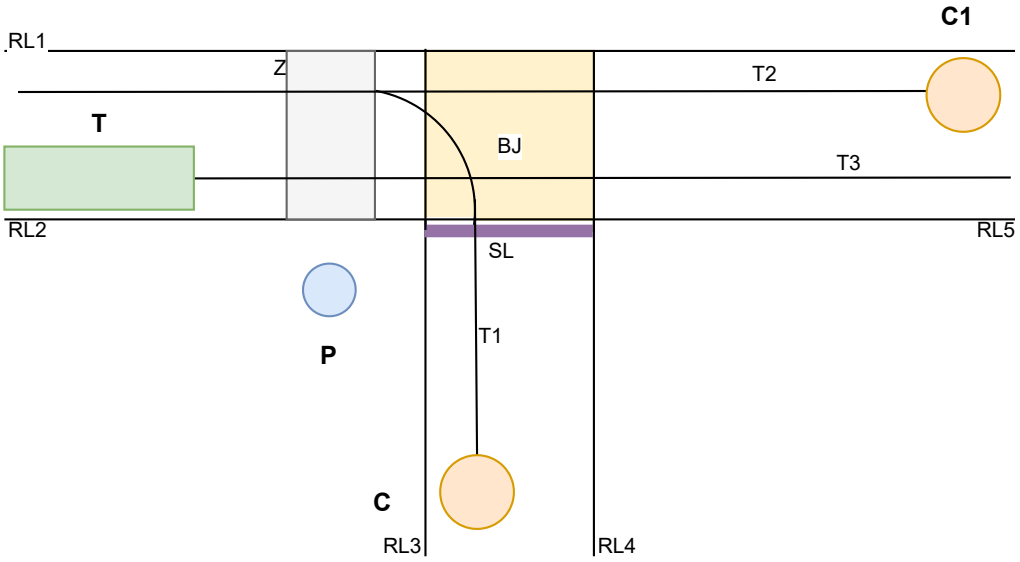
All information received from the system representing objects of our scenario has to be encoded as described above in order to form and monitor our property. However, as some objects are dynamic, as the cars, tram and pedestrian a continuous trace is produced and sent in the form of metadata from the simulator as to keep track of the evolution of these dynamic objects. Therefore it is necessary to continuously update the property formula with the newest trace information.

The scenario metadata will firstly be processed to define the constant restrictions from the scenario. For our use case it is the static elements that represent the road limits, trajectories, zebra cross, box junction, and stop sign limit. In the equations described in Figure 3.1 we can see an example of the final result of that metadata already encoded into $FOL_{\mathbb{R}}$. All constant values on the equations are empirical, set to form the mapping of the scenario seen in Figure 3.1. We represent our scenario in a two-dimensional space in which every point is represented as a pair of real numbers.

Each of the expressions on Figure 3.2 create the regions and lines seen in Figure 3.1(b) for the respective names. The mapping of the environment of this scenario can be such




(a) Layout with trajectories




(b) Encoded mapping

Figure 3.1: Scenario

as to form a layout as seen in Figure 3.1(b) containing all static objects described and an example of the dynamic objects where $\mathsf{RLn}_{0<n<5}$ are the road limit; C and C1 as cars; T as Tram; P as a pedestrian; Z as the pedestrian crossing; BJ as the box junction; SL as the stop limit; T1 as the trajectory for car **C**; T2 as the trajectory for car **C2**; and T3 as the trajectory for tram **T**.

In equation 3.1 we describe the trajectory that car **C** must take. It is composed of three different line segments that together form the line of the trajectory. In equations 3.2 and 3.3 we describe the trajectories from car **C2** and the tram respectively. In 3.4 we

| Region | Formula | |
|---|---|---|
| **T1** | $(((x+2)^2+(x-1)^2=2^2) \wedge x<0 \wedge -2<x \wedge 1<y) \vee$ $(y=3 \wedge x<-2 \wedge -16<x) \vee$ $(x=0 \wedge y<1 \wedge -7.5<y)$ | (3.1) |
| **T2** | $y=3 \wedge x<15 \wedge -15<x$ | (3.2) |
| **T3** | $y=0 \wedge x<15 \wedge -15<x$ | (3.3) |
| **RL** | $(y=5.5 \wedge x>-15 \wedge x<15) \vee (y=-1.5 \wedge x>-15 \wedge x<-3) \vee$ $(x=-3 \wedge y>-8 \wedge y<-1.5) \vee (x=3 \wedge y>-8 \wedge y<1.5) \vee$ $(y=-1.5 \wedge x>3 \wedge x<15)$ | (3.4) |
| **BJ** | $x>-3 \wedge x<3 \wedge y>-1.5 \wedge y<5.5$ | (3.5) |
| **Z** | $x>-5 \wedge x<-3 \wedge y>-1.5 \wedge y<5.5$ | (3.6) |
| **SL** | $x>-3 \wedge x<3 \wedge y=-1.5$ | (3.7) |

Figure 3.2: Scenario encoding as spatial variables $T1, T2, T3, RL, BJ, Z, SL$.

describe the equations that represent the five lines that form the road limits of the scenario. In equation 3.5, 3.6 and 3.7 we describe the bounding boxes that represent the box junction area, the zebra cross area and the stop sign limit area respectively.

However, in practice these equations must be transformed from $FOL_{\mathbb{R}}$ into a language a SMT solver can interpret describing the scenario of the Figure 3.1(a). For example to describe the constrains from the box junction seen in equation 3.5 in Z3 we express it as shown in equation 3.8.

$$\begin{aligned} &x = Real(\text{"}x\text{"}) \\ &y = Real(\text{"}y\text{"}) \\ &And(x>-3, x<3, y>-1.5, y<5.5) \end{aligned} \tag{3.8}$$

## 3.3 Encoding of the trace

For the dynamic objects in our scenario there is a need of a continuous trace to keep track of their position at every time step. At all instants, the trace is sent from the simulator in the form of a tree data structure. Besides the objects a trace also contains an event ID and a timestamp for every event. The encoding of dynamic elements very similar with the scenario encoding as both are encoding objects represented as circular or bounding box regions. In practice, the scenario and trace are transformed in such a way that a satisfiability solver can interpret them.

We may see three types of traces, finite, infinite and bi-infinite traces. A finite trace forms the set $A^n = \{\sigma : 0..n \mapsto A\}$, where $(\sigma_0, \sigma_1, \ldots, \sigma_n)$ defines a sequence of symbols.

A infinite trace forms the set $A^{\mathbb{N}_0} = \{\sigma : \mathbb{N}_0 \mapsto A\}$, where $(\sigma_0, \sigma_1, \sigma_2, \ldots)$ defines a sequence of symbols.

A bi-infinite trace forms the set $A^{\mathbb{Z}} = \{\sigma : \mathbb{Z} \mapsto A\}$, where $(\ldots, \sigma_{-2}, \sigma_{-1}\sigma_0, \sigma_1, \sigma_2, \ldots)$ defines two infinite traces $(\sigma_{-1}, \sigma_{-2}, \ldots)$ and $(\sigma_0, \sigma_1, \sigma_2, \ldots)$.

Without loss o generality, we can consider our objects in a 2D Euclidean space. Other geometric shapes besides the circle and bounding box could be considered but are not required for our use case.

Since the attributes as position and size for dynamic objects may change from event to event the encoding of these objects is done only when we need to refer to the object in a time instance. The $constr : id \mapsto \mathfrak{L}$ function generates the constraints of circles and bounding boxes with two free variables $x, y$. $id \in \{circle, bbox\}$ such as

$$constr(t) := \begin{cases} (x - cpx)^2 + (y - cpy)^2 < r^2, & \textit{if } \text{t=circle} \\ (cpx - width/2) \leq x \wedge x \leq (cpx + width/2) \wedge \\ (cpy - lenght/2) \leq y \wedge y \leq (cpy + length/2) & \textit{if } \text{t=bounding box} \end{cases},$$

where $(cpx, cpy)$ is the center point of the region obtained in the trace where the values for $r$, $width$ and $length$ can be found as well.

To encode bi-infinite traces, we define the function $encode : A^{\mathbb{Z}} \mapsto (\mathfrak{P} \times \mathfrak{L})$, where the set $\mathcal{P}$ contains real variables in $FOL_{\mathbb{R}}$. $\mathfrak{P}$ denotes the set of spatial variables in $LTL \times MS^{\leq}$. To conclude the encoding of the trace, the functions $n : A^{\mathbb{Z}} \mapsto A^{\mathbb{Z}}$ and $p : A^{\mathbb{Z}} \mapsto A^{\mathbb{Z}}$ form the next and the previous element of a trace.

To assure a fluid access to the attributes of each object in each event we used a dictionary data structure to sort the events by id, for each id we allocated a event object of the class Event and its respective list of elements from class Element. Each element can be represented as a circle or bounding box, when its a circle we add an object circle from class Circle to element, if its a bounding box we add the object from class BoundingBox. Class position creates a position object indicating the position and rotation of an object. These classes are as seen in listing 3.1.

Listing 3.1: Event object

```
class Event:
    def __init__(self : Self , eventID : Int , timestamp : Data ,
    elements : List[Element])

class Element:
    def __init__(self : Self , id : Int , type : String ,
    region : BoundingBox or Circle , position : Position ):

class Circle:
    def __init__(self : Self , type : String , radius : Float ):

class BoundingBox:
    def __init__(self : Self ,type : String , length : Float ,
```

```
    width : Float , height : Float ):

class Position:
    def __init__(self : Self, x : Float, y : Float, z : Float,
    yaw : Float, pitch : Float, roll : Float):
```

When receiving new trace information being from a stream or a log file it may be pre-processed and organized for latter access for the encoding of the dynamic objects when evaluating the property.

## 3.4 Encoding of the traffic rules

According to the Vienna convention [29], road traffic rules describe the way pedestrians and vehicles should behave on a street environment. Without loss of generality, we identify three rules of interest to describe in the $LTL \times MS^{\leq}$ specification language. We consider these rules as general ADS safety requirements to check in a given scenario. These rules describe the properties we will monitor which can only be defined following a specific grammars syntax and semantics, seen in table 3.1, build upon $LTL \times MS^{\leq}$. Any property that respects this grammar can be fed to our monitor, whereas we believe that any road safety rule can be described without resorting to more than one sequence of nested temporal operators, i.e., a temporal operator inside another temporal operator, as it will have a negative impact in the performance of the tool.

We now follow how our rules are formally described using $LTL \times MS^{\leq}$:

**Rule 1 (vehicle safety-margin)** *To simplify the presentation, this rule is divided in two parts: (a) a vehicle should maintain a safety-margin relative to the walkways (based on article 13 of the Vienna convention [29]), and (b) a vehicle should maintain a safety-margin from the vehicle in front of it.*

*The (a) part of this rule is written, using the defined grammar, as*

```
(property
        (and (always (overlap (prop "T") (expand (prop "C") 1.)))
        (not (eventually (overlap (prop "RL") (expand (prop "C" ) 1.)))))
       )
```

*and described in* $LTL \times MS^{\leq}$ *using a compact form by*

$$\Box(O(\mathbf{T}, (\exists^{\leq 1}\mathbf{C}))) \wedge \neg\Diamond(O(\mathbf{RL}, (\exists^{\leq 1}\mathbf{C}))),$$

*where* **T** *corresponds to the trajectory,* **C** *to the car and* **RL** *to the road limits. Informally, it reads as the vehicle* **C** *should maintain a safety-margin of at least one meter between the car and the road limit, and follow its predefined trajectory within an error of one meter. We begin now the exemplification of the transformation from the compact form to the extended form step by step, as follows:*

| | | |
|---|---|---|
| Property: | property(formula) | |
| Formulas: | subseteq(term, term) | implies(formula, formula) |
| | and(formula, formula) | alwayspast(formula) |
| | or(formula, formula) | once(formula) |
| | not(formula) | eventually(formula) |
| | include(term, term) | always(formula) |
| | overlap(term, term) | previous(formula) |
| | equal(term, term) | next(formula) |
| | disconect(term, term) | until(formula, formula) |
| | | since(formula, formula) |
| Terms: | prop(string) | nextt(term) |
| | union(term, term) | intersection(term, term) |
| | negation(term) | expand(term, float) |

Table 3.1: Grammar

- *step 1 (apply O overlap)*

$$\Box(\neg(\mathbf{T} \sqcap (\exists^{\leq 1}\mathbf{C}) = \bot) \wedge \neg(\mathbf{T} \sqsubseteq (\exists^{\leq 1}\mathbf{C})) \wedge \neg((\exists^{\leq 1}\mathbf{C}) \sqsubseteq \mathbf{T})$$
$$\wedge \neg\Diamond((\neg(\mathbf{RL} \sqcap (\exists^{\leq 1}\mathbf{C}) = \bot) \wedge \neg(\mathbf{RL} \sqsubseteq (\exists^{\leq 1}\mathbf{C})) \wedge \neg((\exists^{\leq 1}\mathbf{C}) \sqsubseteq \mathbf{RL})))$$

- *step 2 (apply $\Diamond$ eventually)*

$$\Box(\neg(\mathbf{T} \sqcap (\exists^{\leq 1}\mathbf{C}) = \bot) \wedge \neg(\mathbf{T} \sqsubseteq (\exists^{\leq 1}\mathbf{C})) \wedge \neg((\exists^{\leq 1}\mathbf{C}) \sqsubseteq \mathbf{T})$$
$$\wedge \neg(\top \mathcal{U} (\neg(\mathbf{RL} \sqcap (\exists^{\leq 1}\mathbf{C}) = \bot) \wedge \neg(\mathbf{RL} \sqsubseteq (\exists^{\leq 1}\mathbf{C})) \wedge \neg((\exists^{\leq 1}\mathbf{C}) \sqsubseteq \mathbf{RL}))))$$

- *step 3 (apply $\Box$ always)*

$$\neg(\top \mathcal{U} \neg(\neg(\mathbf{T} \sqcap (\exists^{\leq 1}\mathbf{C}) = \bot) \wedge \neg(\mathbf{T} \sqsubseteq (\exists^{\leq 1}\mathbf{C})) \wedge \neg((\exists^{\leq 1}\mathbf{C}) \sqsubseteq \mathbf{T})$$
$$\wedge \neg(\top \mathcal{U} (\neg(\mathbf{RL} \sqcap (\exists^{\leq 1}\mathbf{C}) = \bot) \wedge \neg(\mathbf{RL} \sqsubseteq (\exists^{\leq 1}\mathbf{C})) \wedge \neg((\exists^{\leq 1}\mathbf{C}) \sqsubseteq \mathbf{RL}))))$$

*Let us now consider the (b) part of this rules that introduces the safety-margin between two cars. (a) and (b) parts can be described in $LTL \times MS^{\leq}$ by the expression*

$$\Box(O(\mathbf{T}, (\exists^{\leq 1}\mathbf{C})) \wedge \neg\Diamond(O(\mathbf{RL}, (\exists^{\leq 1}\mathbf{C})))) \wedge \neg\Diamond(O((\exists^{\leq 2}\mathbf{C2}), (\exists^{\leq 2}\mathbf{C}))),$$

*where* **C2** *corresponds to the car.*

*The first half of the conjunction corresponds to the (a) part of this rule that is already extended above. While the second half of the conjunction is transformed in*

$$\neg(\top \mathcal{U} (\neg((\exists^{\leq 2}\mathbf{C2}) \sqcap (\exists^{\leq 2}\mathbf{C1}) = \bot) \wedge \neg((\exists^{\leq 2}\mathbf{C2}) \sqsubseteq (\exists^{\leq 2}\mathbf{C1})) \wedge$$
$$\neg((\exists^{\leq 2}\mathbf{C1}) \sqsubseteq (\exists^{\leq 2}\mathbf{C2})))).$$

**Rule 2 (stop-on-forbidden areas)** *A vehicle should not stop on top of (a) a box junction, based on the Portuguese road marks M17b and article 18 of the Vienna convention; (b) pedestrian crossing, based on article 23 al.3 of the Vienna convention; (c) tram rails, based on article 23 al.3 of the Vienna convention.*

*The (a) part of this rule can be written using our grammar as*

$$
\begin{aligned}
&(property \\
&\quad (always\ (implies\ (or\ (include\ (prop\ "C")\ (prop\ "BJ"))\\
&\quad (overlap\ (prop\ "C")\ (prop\ "BJ")))\\
&\quad (not\ (equal\ (prop\ "C")\ (nextt\ (prop\ "C"))))))\\
&\quad )
\end{aligned}
$$

*and described in* $LTL \times MS^{\leq}$ *in a compact form by*

$$\square((I(\mathbf{C},\mathbf{BJ}) \vee O(\mathbf{C},\mathbf{BJ})) \rightarrow \neg EQ(\mathbf{C}, \bigcirc_{\rho} \mathbf{C}))$$

*where* **BJ** *corresponds to the box junction and* **C** *to the car. The (a) part of tis rule describes that the car must never stop on top of a box junction. This means that at all times the car is overlapped or included in the box junction region and that moment is represented as n that means in moment* $n+1$*, i.e. the next moment in time, the car must not be at the exact same position. Otherwise, if verified that in two sequential moments the car is in the same position it means the car has stopped.*

*We begin now the exemplification of the transformation from the compact form to the extended form step by step, as follows:*

- *step 1 (introduce implication)*

$$\square(\neg(I(\mathbf{C},\mathbf{BJ}) \vee O(\mathbf{C},\mathbf{BJ})) \vee \neg EQ(\mathbf{C}, \bigcirc_{\rho} \mathbf{C}))$$

- *step 2 (apply temporal connectives* $\square$ *"always")*

$$\neg(\top\, \mathcal{U}\, \neg(\neg(I(\mathbf{C},\mathbf{BJ}) \vee O(\mathbf{C},\mathbf{BJ})) \vee \neg EQ(\mathbf{C}, \bigcirc_{\rho} \mathbf{C}))$$

- *step 3 (apply* $\bigcirc_{\rho}$ *"next projection")*

$$\neg(\top\, \mathcal{U}\, \neg(\neg(I(\mathbf{C},\mathbf{BJ}) \vee O(\mathbf{C},\mathbf{BJ})) \vee \neg EQ(\mathbf{C}, (\bot\, \mathfrak{U}\, \mathbf{C}))))$$

- *step 4 (introduce O "overlap", EQ "equal", and I "included")*

$$
\begin{aligned}
&\neg(\neg(\top\, \mathcal{U}\, \neg(\mathbf{C} \sqsubseteq \mathbf{BJ}) \wedge \neg(\mathbf{BJ} \sqsubseteq \mathbf{C}) \vee\\
&(\neg(\mathbf{C} \sqcap \mathbf{BJ} = \bot) \wedge \neg(\mathbf{C} \sqsubseteq \mathbf{BJ}) \wedge \neg(\mathbf{BJ} \sqsubseteq \mathbf{C})) \vee\\
&\neg((\mathbf{C} \sqsubseteq (\bot\, \mathfrak{U}\, \mathbf{C})) \wedge ((\bot\, \mathfrak{U}\, \mathbf{C}) \sqsubseteq \mathbf{C}))))
\end{aligned}
$$

*Parts (b) and (c) follow the same encoding as described since we can only replace BJ with a region of crosswalk and tramway.*

**Rule 3 (stop-sign)** *A vehicle should stop at a stop sign, based on the Portuguese road signals. This rule is written, using the defined grammar, as*

$$
\begin{aligned}
&(property\\
&\quad(always\ (implies\ (and\ (overlap\ (prop\ "S")\ (expand\ (prop\ "C")\ 1.))\\
&\quad(not\ (once\ (and\ (equal\ (prop\ "C")\ (nextt\ (prop\ "C")))\\
&\quad(overlap\ (prop\ "S")\ (expand\ (prop\ "C")\ 1.))))))\\
&\quad(or\ (equal\ (prop\ "C")\ (nextt\ (prop\ "C")))\\
&\quad(eventually\ (and\ (equal\ (prop\ "C")\ (nextt\ (prop\ "C")))\\
&\quad(overlap\ (prop\ "S")\ (expand\ (prop\ "C")\ 1.)))))))\\
&\quad)
\end{aligned}
$$

*and described in* $LTL \times MS^{\leq}$ *using a compact form by*

$$
\begin{aligned}
&\Box((O(\mathbf{S},(\exists^{\leq 1}\mathbf{C}))\wedge\neg\diamondsuit(EQ(\mathbf{C},\bigcirc_{\rho}\mathbf{C})\wedge O(\mathbf{S},(\exists^{\leq 1}\mathbf{C}))))\rightarrow\\
&(EQ(\mathbf{C},\bigcirc_{\rho}\mathbf{C})\vee\diamondsuit(EQ(\mathbf{C},\bigcirc_{\rho}\mathbf{C})\wedge O(\mathbf{S},(\exists^{\leq 1}\mathbf{C})))))
\end{aligned}
$$

*where* **S** *corresponds to the stop sign and* **C** *to the car. This compact form can be informally described as: always, if the extension by 1 of the region of the car is overlapped with the stop region and never has the car stopped when overlapped with that stop limit, then eventually the car will stop while overlapped with that stop limit, i.e., the current region of the car will be the same as its next projection while being overlapped with the region of the stop limit. We begin the exemplification of the transformation from the compact form to the extended form step by step, as follows:*

- *step 1 (introduce implication)*

$$
\begin{aligned}
&\Box(\neg(O(\mathbf{S},(\exists^{\leq 1}\mathbf{C}))\wedge\neg\diamondsuit(EQ(\mathbf{C},\bigcirc_{\rho}\mathbf{C})\wedge O(\mathbf{S},(\exists^{\leq 1}\mathbf{C}))))\vee\\
&(\diamondsuit(EQ(\mathbf{C},\bigcirc_{\rho}\mathbf{C})\wedge O(\mathbf{S},(\exists^{\leq 1}\mathbf{C})))))
\end{aligned}
$$

- *step 2 (apply* $\Box$ *"always",* $\diamondsuit$ *"eventually", and* $\diamondsuit$ *"once")*

$$
\begin{aligned}
&\neg\top\,\mathcal{U}\,\neg(\neg(O(\mathbf{S},(\exists^{\leq 1}\mathbf{C}))\wedge\neg\top\,\mathcal{S}(EQ(\mathbf{C},\bigcirc_{\rho}\mathbf{C})\wedge O(\mathbf{S},(\exists^{\leq 1}\mathbf{C}))))\vee\\
&(\top\,\mathcal{S}(EQ(\mathbf{C},\bigcirc_{\rho}\mathbf{C})\wedge O(\mathbf{S},(\exists^{\leq 1}\mathbf{C})))))
\end{aligned}
$$

- *step 3 (apply* $\bigcirc_{\rho}$ *"next projection")*

$$
\begin{aligned}
&\neg\top\,\mathcal{U}\,\neg(\neg(O(\mathbf{S},(\exists^{\leq 1}\mathbf{C}))\wedge\neg\top\,\mathcal{S}(EQ(\mathbf{C},(\bot\mathfrak{U}\mathbf{C}))\wedge O(\mathbf{S},(\exists^{\leq 1}\mathbf{C}))))\vee\\
&(\top\,\mathcal{S}(EQ(\mathbf{C},(\bot\mathfrak{U}\mathbf{C}))\wedge O(\mathbf{S},(\exists^{\leq 1}\mathbf{C})))))
\end{aligned}
$$

- *step 4 (introduce O "overlap" and EQ "equal")*

$$\neg\top\,\mathcal{U}\,\neg(\neg(((\neg(\mathbf{S}\sqcap(\exists^{\leq 1}\mathbf{C})=\bot))\wedge(\neg(\mathbf{S}\sqsubseteq(\exists^{\leq 1}\mathbf{C})))\wedge(\neg(\exists^{\leq 1}\mathbf{C})\sqsubseteq\mathbf{S}))\wedge$$
$$\neg\top\,\mathcal{S}(((\mathbf{C}\sqsubseteq(\bot\,\mathfrak{U}\mathbf{C}))\wedge((\bot\,\mathfrak{U}\mathbf{C})\sqsubseteq\mathbf{C}))\wedge$$
$$((\neg(\mathbf{S}\sqcap(\exists^{\leq 1}\mathbf{C})=\bot))\wedge(\neg(\mathbf{S}\sqsubseteq(\exists^{\leq 1}\mathbf{C})))\wedge(\neg(\exists^{\leq 1}\mathbf{C})\sqsubseteq\mathbf{S}))))\vee$$
$$(\top\,\mathcal{S}(((\mathbf{C}\sqsubseteq(\bot\,\mathfrak{U}\mathbf{C}))\wedge((\bot\,\mathfrak{U}\mathbf{C})\sqsubseteq\mathbf{C}))\wedge$$
$$((\neg(\mathbf{S}\sqcap(\exists^{\leq 1}\mathbf{C})=\bot))\wedge(\neg(\mathbf{S}\sqsubseteq(\exists^{\leq 1}\mathbf{C})))\wedge(\neg(\exists^{\leq 1}\mathbf{C})\sqsubseteq\mathbf{S})))))$$

## Summary

As we intent to monitor safety proprieties we needed to formalize our traffic as well the objects that represent the spatial propositions on those rules. We showed how we encoded our scenario into logical expressions that represent each static objects' region. We explained the difference between static and dynamic objects and how we structure a trace to provide the needed information to encode our dynamic objects. We then shown how each of our traffic rules are formalized using our specification language $LTL\times MS^{\leq}$. This formalization together with the scenario is passed as input for our tool to generate our monitor.

# 4

# Monitor Generation and Decision Method

In this chapter we present the detailed approach we follow for the monitoring of safety properties. The verification of a $LTL \times MS^{\leq}$ formula consists in the construction of a monitoring model and a decision procedure. Given a trace, the decision procedure answers whether a trace satisfies the monitoring model. Following the approach we present the algorithm that transforms formal expressions into models to be passed through a solver. We describe how it works, its expressiveness as well as its restrictions and how it will be used to provide us the verdict on the monitor. We then present the alignment method of the model with the trace in order to produce the final verdict.

## 4.1 Detailed Approach

For our use case, we assume a mapping of an urban environment in a metadata form that is already provided for us described as our scenario seen in section 3.2. Since we are working on a simulator, it will be a mapping of the simulator's Field-of-View, as for example a road-side unit camera. This will grant us the expressions that stage the static objects on our scenario which are added to a dictionary data structure with a string as key that will indicate which objects expression we refer to.

Besides the mapping of the scenario itself, we will be provided with a trace that contains the mapping of the dynamic objects, such as the cars, from that same simulator's Field-of-View.

Each event from the trace represented by a frame captured by a road-side unit camera will contain the snapshot's timeframe and event incremental ID, as well as the set of dynamic objects captured in that frame. Each dynamic object from the trace is identified by their unique ID, coordinates, type of object, i.e., if it's a pedestrian, a car, a tram, etc, the type of region that represents the object, i.e., if it is represented as a bounding box or a circle, and those regions constrains. Therefore, the trace will be a sequential set of events that are snapshots taken one after another containing information about the dynamic objects of the simulation organized in a JSON data file as seen in Figure 4.1.

```
[{
"eventID" : 1,
"timestamp" : "10:00:00",
"elements" : [
    {
        "ID" : 1,
        "type" : "Car",
        "position" : {"x" : 0, "y" : -4.5},
        "region" : {"type" : "circle", "radius" : 2}
    }
]
},
{
    "eventID" : 2,
    "timestamp" : "10:00:15",
    "elements" : [
        {
            "ID" : 1,
            "type" : "Car",
            "position" : {"x" : 0, "y" : -2},
            "region" : {"type" : "circle", "radius" : 2}
        }
    ]
    },
{
    "eventID" : 3,
    "timestamp" : "10:00:30",
    "elements" : [
        {
            "ID" : 1,
            "type" : "Car",
            "position" : {"x" : 0, "y" : -2},
            "region" : {"type" : "circle", "radius" : 2}
        }
    ]
    }
]
```

Figure 4.1: Example of a trace with the object from the diagram in 4.2

The trace basically completes our scenario information with the introduction of the dynamic elements of the simulation. The information from the trace together with the one from our scenario provides us with the data needed to give us the notion of how each object is performing in the simulation. Consequently providing us the tools to describe the behaviours between objects we expects to see throughout the simulation.

One trace includes at least two sequential frames captured by the camera because, to check safety properties, we need to have a notion of evolution in the trace, therefore we need more than one frame to be able to compute that evolution.

Even though we use simulated traces, it is predictable to conclude these traces can be extracted and then computed equally within a real-world camera unit.

A example mapping of the information adhered from the simulation can be visually expressed as seen in Figure 4.2 where the sub-figures are shown in a sequential time flow where if sub-figure 4.2(a) is a capture of moment $n$ then sub-figure 4.2(b) is captured at

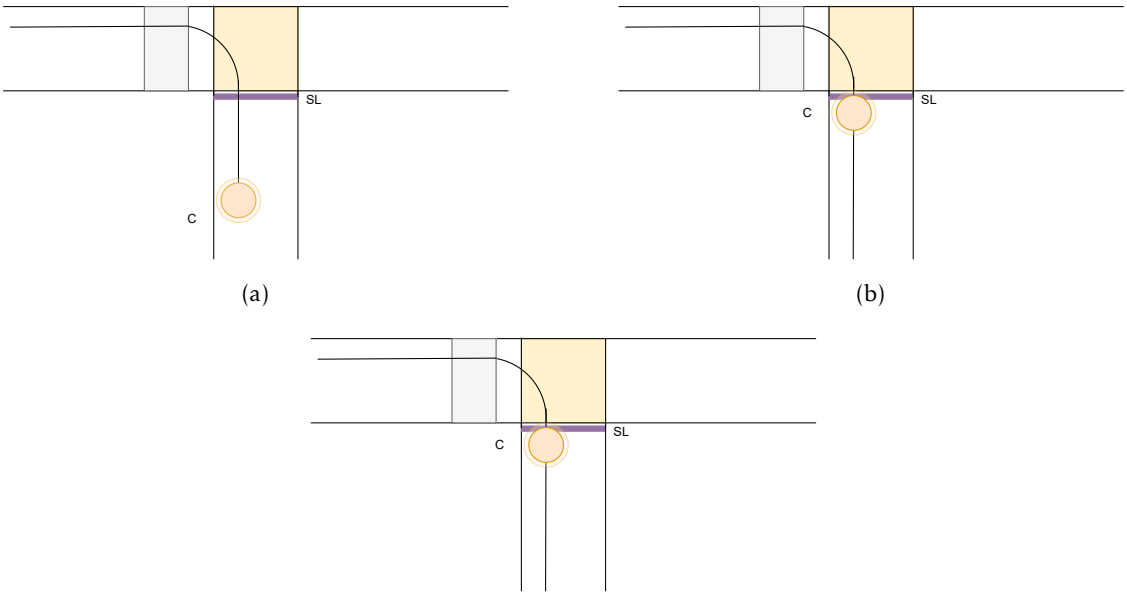



Figure 4.2: Example diagram

moment *n+1* and sub-figure 4.2(c) at moment *n+2*.

We can establish a direct relation with the trace information in Figure 4.1 and the position of the dynamic object, i.e., the car, in Figure 4.2 where *eventID 1* corresponds to sub-figure 4.2(a), *eventID 2* corresponds to sub-figure 4.2(b), and *eventID 3* corresponds to sub-figure 4.2(c).

Before we even start any computational work, we need to read the environment and follow a hazard analysis to define a group of safety rules applicable to an ADS. Road safety rules that we defined in section 3.4. These road safety rules will define the requirements to meat in an ADS, therefore the requirements we want to monitor. The use case example we use to demonstrate this approach is the one we informally introduced in the first chapter in 1.3, namely the monitoring of the 'stop at the stop sign' rule.

The next step is translating the safety requirement we want to monitor into a formal language. This language will include both spatial and temporal constraints in order to be able to express spatial behaviours over a flow of time. We will use spatial models, with spatial propositions that are non-empty sets of points representing objects, to define the relation between the objects in our environment. To define the objects' relational evolution we use the temporal constraints that aboard time in a discrete, linear way [1]. This approach of using the spatio-temporal logic fragment we introduced in section 2.3.5 offers the expressive power to formalize all the rules we described. Any property compliant to the grammar introduced in section 3.4 with a maximum of one nested temporal operator can be monitored by our tool using this approach.

For example, we have the car represented as a circle and the road stop limit as a

bounding box as seen in Figure 4.2. The expressive power of our formal language allows us to evaluate if at some point of time the car is overlapped with the stop sign limit or even dictate that this behaviour must be constant throughout a trace. This notion of overlapped can be expressed using the terms and formulas provided by $LTL \times MS^{\leq}$ in the following way:

$$O(\mathbf{C},\mathbf{S}) ::= \neg(\mathbf{C} \sqcap \mathbf{S} = \bot) \wedge \neg(\mathbf{C} \sqsubseteq \mathbf{S}) \wedge \neg(\mathbf{S} \sqsubseteq \mathbf{C})$$

Where $\mathbf{C}$ represents the region of the car, $\mathbf{S}$ represents the region of the stop sign limit and $O$ as the overlapped operator. By inserting the temporal operator "always" before the overlapped expression $\Box(O(\mathbf{C},\mathbf{S}))$ we are inducing that this behaviour must hold in all moments of the flow of time and therefore can be seen as a safety property. In another hand if we insert the temporal operator 'èventually" $\Diamond(O(\mathbf{C},\mathbf{S}))$ we are inducing that this behaviour should happen in a future moment of the flow of time and therefore can be seen as a liveness property.

Together with the $FOL_{\mathbb{R}}$ constrains that define the scenario, the property is processed and transformed into a monitoring model in SMT-LIB, that can be interpreted by Z3 [27], the non-linear satisfiability solver that we rely on. For that, the property is passed through an algorithm that has a function for every term and formula of $LTL \times MS^{\leq}$ that translates the $LTL \times MS^{\leq}$ expression into an incomplete model.

After this we complete this model with the information gathered from the trace, i.e., the position and region of the dynamic objects, that produce the $FOL_{\mathbb{R}}$ formulas representing the dynamic objects at every instant of the flow of time. The tool packs the SMT-LIB model and the $FOL_{\mathbb{R}}$ expressions from the trace and using Z3 as a tool decides the satisfiability of that model testing for the correctness of the property and producing the final verdict.

A simple example of the language the SMT-LIB solver can interpret is shown in Figure 4.3 where we assert if both the car and stop regions from event with ID 2 of Figure 4.1 are equal which in a $LTL \times MS^{\leq}$ expression would be $EQ(\mathbf{C},\mathbf{S})$.

We would then use Z3 as a tool to decide the satisfiability of this model passing it through a non-linear solver. If the model is satisfied it means under the assumption of those values the property is maintained. In this specific case the evaluation will give an output of *unsat* meaning the model in not satisfied since the region of the car is not equal to the region of the stop limit. However, looking over only one instance of the trace may not be enough because as we are evaluating spatial relations on dynamic objects throughout a flow of time we need to check the property for every instance as the position or size of the dynamic object may change. In this example if we wanted to assure that the region of the car is never equal to the region of the stop sign we would have to check this assertion every time the car constrains change throughout the trace, which is done using the temporal operators.

```
x = Real("x")
y = Real("y")

; constrains C
C = (x - 0)*(x - 0) + (y + 2)*(y + 2) <= 4

; constrains S
S = And(x > -3, x < 3, y == -1.5)

;Equality operator
And(ForAll([x,y], Implies(S, C)), ForAll([x,y], Implies(C, S)))
```

Figure 4.3: Ad-hoc encoding of a simple model example of a property to check in Z3

### 4.1.1 Monitor Generation

The verification of a safety property consists in the construction of a monitoring model and a decision procedure. Given a trace, the decision procedure answers whether that trace satisfies the monitoring model. The execution flow of the decision procedure for the monitoring of a property using our tool is as shown in Figure 4.4.

The first step is to identify the traffic rule we want to monitor and describe it as our requirement using the formalism of the logical fragment $LTL \times MS^{\leq}$ as shown in section 3.4. This formalisation is combined with the propositions describing all static objects provided by our scenario, i.e., the formulas that represent the regions of the static objects in $FOL_{\mathbb{R}}$. This expressions is hosted by an algorithm that transforms it into a model, describing the property, interpretative by the SMT-LIB solver.

Then, a trace is fed into the monitor, this trace is processed and organized into a data structure and then encoded to form the $FOL_{\mathbb{R}}$ expressions that represent each dynamic object for every instance of the traces flow of time. This encoding of the dynamic objects from the trace is merged with the previously build model from the property to form a model we will pass through our Z3 solver in order to obtain our desired verdict.

This verdict indicates if our model is satisfied therefore suggesting rather our requirement is maintained throughout that trace.

## 4.2 Model construction

This model consists of formulas written in $FOL_{\mathbb{R}}$. As input our algorithm receives an $LTL \times MS^{\leq}$ property that represents the requirement under analysis. The algorithm has access to the $FOL_{\mathbb{R}}$ expressions that represent the static objects contained in our scenario. The languages of terms and formulas in $LTL \times MS^{\leq}$ are respectively represented by the sets $\mathfrak{T}$ and $\mathfrak{F}$, and the first-order language $FOL_{\mathbb{R}}$ forms the set $\mathfrak{L}$. Our algorithm transforms every term and formula into $FOL_{\mathbb{R}}$ expressions building up our model.

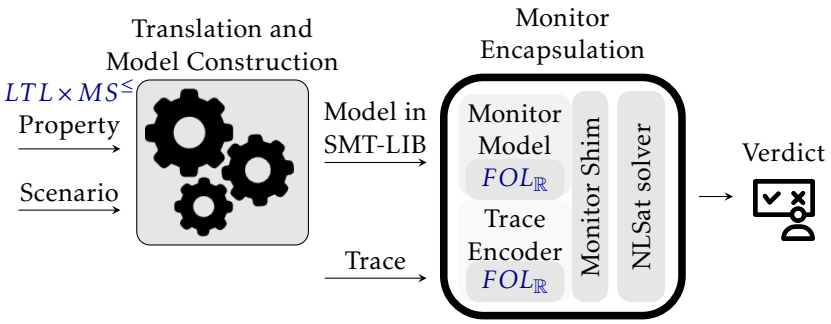



Figure 4.4: Monitor generation and its execution flow.

### 4.2.1 Algorithm description

This algorithm has the purpose to produce a model, representing the property to check, that can be interpreted by the SMT solver. This model is build based on $FOL_{\mathbb{R}}$ over real numbers and in turn the SMT will decide the satisfiability of such model. As the input the algorithm receives an $LTL \times MS^{\leq}$ expression that represents the requirement.

First we will take a look at the functions applied to our terms that describe the objects of our world, these are:

- the spatial variable $\rho$ which identifies an object;
- the complement $\overline{\rho}$ which returns the whole metric space minus the object represented by $\rho$;
- intersection $\rho_1 \sqcap \rho_2$ which return a model referring to the set of points common in both $\rho 1$ and $\rho 2$;
- union $\rho_1 \sqcup \rho_2$ produces a model with both models from $\rho 1$ and $\rho 2$;
- the distance operator $\exists^{\leq a}\rho$ expands the spatial model representing $\rho$ by including in it all the points at a distance less or equal to $a$ of its own closed set of points;
- next projection $false \wr\!\wr \rho$ returns the model referring to $\rho$ of the subsequent time instant; and
- the until $\rho 1 \wr\!\wr \rho 2$ makes use of the valuation map associating each spacial variable to a time instance to apply the binary temporal operator until in the terms and is only possible by the combination of spatial and temporal logic. It offers the capability of navigating through the flow of time searching for spatial variables in different time instances applying the temporal operator to regions.

Every term $\rho$ in $LTL \times MS^{\leq}$ is translated by the function $\mathbf{conv}_{\rho}() : \Upsilon \mapsto \mathfrak{L}$ seen below:

$$\mathbf{conv}_{\rho}(\rho) := \begin{cases} eval(\rho), & if \rho = \rho \\ \neg(\mathbf{conv}_{\rho}(\rho)), & if \rho = \overline{\rho} \\ \mathbf{conv}_{\rho}(\rho_1) \wedge \mathbf{conv}_{\rho}(\rho_2), & if \rho = \rho_1 \sqcap \rho_2 \\ \mathbf{conv}_{\rho}(\rho_1) \vee \mathbf{conv}_{\rho}(\rho_2), & if \rho = \rho_1 \sqcup \rho_2 \\ dist(a, \mathbf{conv}_{\rho}(\rho)), & if \rho = \exists^{\leq a} \rho \\ nextT(\mathbf{conv}_{\rho}(\rho)), & if \rho = \bot \,\mathfrak{U}\, \rho \\ \mathbf{conv}_{\rho}(unfold(\rho_1, \rho_2)), & if \rho = \rho_1 \,\mathfrak{U}\, \rho_2 \end{cases}$$

where $eval : \mathfrak{P} \mapsto \mathfrak{L}$ evaluates a spatial variable into an expression that encodes the spatial constraints available in the trace and scenario as $FOL_{\mathbb{R}}$. Note that, if $\rho$ refers to a dynamic object, this function produces a tag variable in $FOL_{\mathbb{R}}$ that later expands to the valuations provided by the objects inside the trace. In the other hand, if it refers to a static object from the scenario it may be immediately evaluated to the $FOL_{\mathbb{R}}$ expression representing that object.

The $dist : \mathbb{R} \times \Upsilon \mapsto \mathfrak{L}$ function introduces a spatial annotation (the distance $a$ from a spatial term) on the spatial variables (propositions). The following property is applied a priory to every $\rho \in \Upsilon$ where $\rho = \exists^{\leq a} \rho$ in our property:
Let $\rho$ be a term, $A$ the set of spatial variables in $\rho$, and $e$ a rational number. For any $\rho$ and $e$, the distance operator $\exists^{\leq e} \rho$ has an equivalent expression with every spatial variable $a \in A$ such that $\exists^{\leq e} a$. Where if originally $a = \exists^{\leq f} a$ then the expression is translated to $\exists^{\leq e+f} a$.

The $nextt : \Upsilon \mapsto \mathfrak{L}$ function adds a temporal annotation, 1 (the successor) on every spatial variable (proposition) inside term $\rho$. Note that every propositions contains a temporal annotation which is cumulative, let $t$ be the temporal annotation where $t \in \mathbb{Z}$, and every propositions starts with a temporal annotation of $t = 0$.

The $unfold : \Upsilon \times \Upsilon \mapsto \mathfrak{L}$ function generates a bounded instance of the infinite sequence

$$\bigvee_{i=1}^{n} \left[ \bigwedge_{j=1}^{i} \left( \underbrace{\bigcirc_{\rho} \cdots \bigcirc_{\rho}}_{\text{j times}} \rho_1 \right) \wedge \underbrace{\bigcirc_{\rho} \cdots \bigcirc_{\rho}}_{\text{j times}} \rho_2 \right],$$

where $\rho_1, \rho_2 \in \rho$ are the input terms. Note that the temporal annotation is done after the unfolding.

To be able to express how the spatial terms relate with each other we appeal to the atomic formulas described. These formulas can be either true or false. The formulas described include a class of spatial formulas:

- the sub set or equals $\rho 1 \sqsubseteq \rho 2$ that indicate that the set $\rho 1$ is sub set or equal to $\rho 2$;
- the negation $\neg\phi$ that returns the opposite of the result of $\phi$;

- the conjunction $\phi 1 \wedge \phi 2$ return the result of the logical conjunction on the two values of $\phi 1$ and $\phi 2$; and
- the disjunction $\phi 1 \vee \phi 2$ return the result of the logical disjunction on the two values of $\phi 1$ and $\phi 2$.

And a class of temporal formulas:

- next formula $false\,\mathcal{U}\,\phi$ which returns the result of the formula from the next temporal instant, i.e., if we are at moment n it returns the formula from moment n +1;
- until, the typical LTL formula $\phi_1\,\mathcal{U}\,\phi_2$ returns true if there is a model in the linear flow of time where $\phi 2$ holds and for all the previous models $\phi 1$ holds.
- previous formula $false\,\mathcal{S}\,\phi$ which returns the result of the formula from the previous temporal instant, i.e., if we are at moment n it returns the formula from moment n - 1;
- since, the dual operator of until $\phi_1\,\mathcal{S}\,\phi_2$ returns true if there is a model in the linear flow of time where $\phi 2$ holds and for all the further models $\phi 1$ holds.

Every formula $\phi$ in $LTL{\times}MS^{\trianglelefteq}$ is translated by the function $\mathbf{conv}_{\varphi}() : \mathcal{F} \mapsto \mathcal{G}$ as seen below:

$$\mathbf{conv}_{\varphi}(\phi) := \begin{cases} \forall(free(\rho_1,\rho_2)).(\mathbf{conv}_{\rho}(\rho_1) \rightarrow \mathbf{conv}_{\rho}(\rho_2)), & if\,\phi = \rho 1 \sqsubseteq \rho 2 \\ \neg(\mathbf{conv}_{\varphi}(\phi)), & if\,\phi = \neg\rho \\ \mathbf{conv}_{\varphi}(\phi 1) \wedge \mathbf{conv}_{\varphi}(\phi 2), & if\,\phi = \phi_1 \wedge \phi_2 \\ \mathbf{conv}_{\varphi}(\phi 1) \vee \mathbf{conv}_{\varphi}(\phi 2), & if\,\phi = \phi 1 \vee \phi 2 \\ \mathbf{conv}_{\varphi}(\phi 1) \rightarrow \mathbf{conv}_{\varphi}(\phi 2), & if\,\phi = \phi 1 \rightarrow \phi 2 \\ next(\mathbf{conv}_{\varphi}(\phi)), & if\,\phi = false\,\mathcal{U}\,\phi \\ \mathbf{conv}_{\varphi}(unfoldUntil(\phi 1,\phi 2)), & if\,\phi = \phi 1\,\mathcal{U}\,\phi 2 \\ previous(\mathbf{conv}_{\varphi}(\phi)), & if\,\phi = false\,\mathcal{S}\,\phi \\ \mathbf{conv}_{\varphi}(unfoldSince(\phi 1,\phi 2)), & if\,\phi = \phi 1\,\mathcal{S}\,\phi 2 \end{cases}$$

Function $free : \mathcal{T} \times \mathcal{T} \mapsto \mathcal{P}$ maps between different structures of free variables of the terms. For instance, $\rho_1[a]$ and $\rho_2[a]$ maps to $\rho_1[a']$ and $\rho_2[a']$. This allows the free variables in $LTL \times MS^{\trianglelefteq}$ to be translated to expressions in $FOL_{\mathbb{R}}$. Term $\rho[a]$ states that $\rho$ has the free variable $a$.

Function $next : \mathcal{F} \mapsto \mathcal{G}$ adds a temporal annotation of 1, referring to that next instant, in every proposition present in that formula. Function $previous : \mathcal{F} \mapsto \mathcal{G}$ adds a temporal annotation of $-1$, referring to the previous moment, in every proposition present in that formula. These temporal annotations are cumulative, i.e., given a proposition $\rho$ with a

temporal annotation $t \in \mathbb{Z}$, let $b \in \mathbb{Z}$ be its new incoming temporal annotation, then the the annotation associated with $\rho$ becomes $t + b$.

Function $unfold_X : \mathcal{F} \times \mathcal{F} \mapsto \mathfrak{L}$ generates a bounded instance of the infinite sequence

$$\bigvee_{i=1}^{n} \left[ \bigwedge_{j=1}^{i} \left( \underbrace{X \ldots X}_{\text{j times}} \phi_1 \right) \wedge \underbrace{X \ldots X}_{\text{j times}} \phi_2 \right]$$

where $\phi_1, \phi_2 \in \varphi$ are the input formulas. Funtion $unfoldUntil : \mathcal{F} \times \mathcal{F} \mapsto \mathfrak{L}$ is defined by $unfold_X$ when $X = \bigcirc$. Function $unfoldSince : \mathcal{F} \times \mathcal{F} \mapsto \mathfrak{L}$ is defined by $unfold_X$ when $X = \ominus$. This unfolding sequence will become something like when applied $unfoldUntil$:

$$\phi 1 \wedge (false\, \mathcal{U}\, \phi 2) \vee$$
$$\phi 1 \wedge (false\, \mathcal{U}\, \phi 1) \wedge (false\, \mathcal{U} (false\, \mathcal{U}\, \phi 2)) \vee$$
$$\phi 1 \wedge (false\, \mathcal{U}\, \phi 1) \wedge (false\, \mathcal{U} (false\, \mathcal{U}\, \phi 1)) \wedge (false\, \mathcal{U} (false\, \mathcal{U} (false\, \mathcal{U}\, \phi 2))) \vee \ldots$$

Note that the temporal annotation is done after the unfolding.

### 4.2.2 Application

The following are examples of the application of the algorithm defined above that build our monitoring model.

**Example 1 (region contradiction)**

$$\overline{a} \sqsubseteq a$$

*The expression* $\boldsymbol{conv}_\phi(\overline{a} \sqsubseteq a)$ *is translated to formulas interpretable by SMT such as:*

$$\forall (free(a)).(\boldsymbol{conv}_\rho(\overline{a}) \rightarrow \boldsymbol{conv}_\rho(a))$$
$$\forall (free(a)).(\neg(\boldsymbol{conv}_\rho(a)) \rightarrow eval(a)$$
$$\forall (free(a)).(\neg(eval(a)) \rightarrow eval(a))$$
$$\forall (free(a)).(\neg(eval(a)) \rightarrow eval(a))$$
$$\forall (free(a)).(eval(a)) \vee eval(a)$$
$$\forall (free(a)).(eval(a))$$

*Note that when a is the universe,* $\boldsymbol{conv}_\phi(\overline{a} \sqsubseteq a)$ *is satisfied.*

**Example 2 (region expansion)**

$$a \sqsubseteq \exists^{\leq 3} a$$

*The expression* $\boldsymbol{conv}_\phi(a \sqsubseteq \exists^{\leq 3} a)$ *is translated to formulas interpretable by SMT such as:*

$$\forall (free(a)).(\boldsymbol{conv}_\rho(a) \rightarrow \boldsymbol{conv}_\rho(\exists^{\leq 3} a))$$
$$\forall (free(a)).(eval(a) \rightarrow dist(3, \boldsymbol{conv}_\rho(a)))$$
$$\forall (free(a)).(eval(a) \rightarrow dist(3, eval(a)))$$
$$\forall (free(a)).(\neg(eval(a)) \vee dist(3, eval(a)))$$

**Example 3 (region with temporal behaviour)**

$$a \sqsubseteq (\bot \,\mathfrak{U}\, d) \wedge \neg(a \sqsubseteq d)$$

*The expression* $\boldsymbol{conv}_{\phi}(a \sqsubseteq (\bot \,\mathfrak{U}\, d) \wedge \neg(a \sqsubseteq d))$ *is translated to formulas interpretable by SMT such as:*

$\boldsymbol{conv}_{\varphi}(a \sqsubseteq (\bot \,\mathfrak{U}\, d)) \wedge \boldsymbol{conv}_{\varphi}(\neg(a \sqsubseteq d))$

$\forall(free(a,d)).(\boldsymbol{conv}_{\rho}(a) \rightarrow \boldsymbol{conv}_{\rho}(\bot \mathfrak{U} d))) \wedge \neg \boldsymbol{conv}_{\varphi}(a \sqsubseteq d)$

$\forall(free(a,d)).(eval(a) \rightarrow nextT(\boldsymbol{conv}_{\rho}(d))) \wedge \neg\forall(free(a,d)).(\boldsymbol{conv}_{\rho}(a) \rightarrow \boldsymbol{conv}_{\rho}(d))$

$\forall(free(a,d)).(eval(a) \rightarrow nextT(proposition(d))) \wedge \neg\forall(free(a,d)).(eval(a) \rightarrow eval(d))$

$\forall(free(a,d)).(\neg(eval(a)) \vee nextT(eval(d))) \wedge \neg\forall(free(a,d)).(\neg(eval(a)) \vee eval(d))$

## 4.3 Monitor Encapsulation

The monitor encapsulation is where the final verdict is formed, the property model and trace together produce a complete monitoring model that is passed through our non-linear satisfiability solver (Z3) producing a verdict over that model.

As input the monitor receives a model built based on a $LTL \times MS^{\leq}$ property and scenario encoding, and a trace in a JSON data structure. The decision process then merges the given model and encoded trace in order to produce a decidable monitoring model, stating if the property is satisfied or not given that specific trace when fed to our solver.

Since formulas and terms converts into incomplete $FOL_{\mathbb{R}}$ expressions, the formalization of the trace completes the encoding.

The trace encoding consists on the construction of the interpreted function *eval*. This function replaces spatial variables with expressions in $FOL_{\mathbb{R}}$ while *free* function constructs the set of free variables for two terms. Note that the trace is a valuation and assigns constraints to the expressions in $FOL_{\mathbb{R}}$.

$encode : A^{\mathbb{Z}} \mapsto (\mathfrak{P} \times \mathfrak{L})$ has been already defined while $inline : \mathfrak{L} \times (\mathfrak{P} \times \mathfrak{L}) \mapsto \mathfrak{L}$ includes the trace in the monitoring model. The first argument receives the monitoring model, and the second argument receives the mapping of the spatial variables in $LTL \times MS^{\leq}$ to the constraints in $FOL_{\mathbb{R}}$. This mapping is only possible if the user indicates which object in the trace corresponds to which spatial variable from the model. This is done by providing the tool a mapping of each spatial variable to an objects unique identifier in the trace. The process concludes by inlining the trace in the monitoring model. Let *trc* be a trace, and $\phi$ a formula in $LTL \times MS^{\leq}$. The inlining is defined by

$$inline(\mathbf{conv}_{\varphi}(\phi), encode(trc)),$$

where *trc* is a finite trace of length $n$.

When the inline is complete we have our monitoring model ready to be checked. This may present two possible results, *sat*, which indicates the property represented by $\phi$

$$\mathbf{conv}_{\rho}(\rho) := \begin{cases} eval(p), & if\, \rho = p \\ \neg\mathbf{conv}_{\rho}(\rho), & if\, \rho = \overline{\rho} \\ \mathbf{conv}_{\rho}(\rho_1) \wedge \mathbf{conv}_{\rho}(\rho_2), & if\, \rho = \rho_1 \sqcap \rho_2 \\ \mathbf{conv}_{\rho}(\rho_1) \vee \mathbf{conv}_{\rho}(\rho_2), & if\, \rho = \rho_1 \sqcup \rho_2 \\ dist(e, \mathbf{conv}_{\rho}(\rho)), & if\, \rho = \exists^{\leq e}\rho \\ nextT(\mathbf{conv}_{\rho}(\rho)), & if\, \rho = \bot\, \mathfrak{U}\, \rho \end{cases}$$

$$\mathbf{conv}_{\varphi}(\phi) := \begin{cases} \forall(free(\rho_1, \rho_2)).(\mathbf{conv}_{\rho}(\rho_1) \rightarrow \mathbf{conv}_{\rho}(\rho_2)), & if\, \phi = \rho_1 \sqsubseteq \rho_2 \\ \neg(\mathbf{conv}_{\varphi}(\phi)), & if\, \phi = \neg\phi \\ \mathbf{conv}_{\varphi}(\phi_1) \wedge \mathbf{conv}_{\varphi}(\phi_2), & if\, \phi = \phi_1 \wedge \phi_2 \\ \mathbf{conv}_{\varphi}(\phi_1) \vee \mathbf{conv}_{\varphi}(\phi_2), & if\, \phi = \phi_1 \vee \phi_2 \\ next(\mathbf{conv}_{\varphi}(\phi)), & if\, \phi = false\, \mathcal{U}\, \phi \\ previous(\mathbf{conv}_{\varphi}(\phi)), & if\, \phi = false\, \mathcal{S}\, \phi \\ always(\mathbf{conv}_{\varphi}(\phi)), & if\, \phi = \Box\phi \\ eventually(\mathbf{conv}_{\varphi}(\phi)), & if\, \phi = \Diamond\phi \\ once(\mathbf{conv}_{\varphi}(\phi)), & if\, \phi = \diamondminus\phi \\ alwaysPast(\mathbf{conv}_{\varphi}(\phi)), & if\, \phi = \boxminus\phi \end{cases}$$

Figure 4.5: Conversion functions $\mathbf{conv}_{\rho}(\rho)$ and $\mathbf{conv}_{\varphi}(\phi)$ for incremental build.

has been maintained throughout the trace *trc*, or *unsat* meaning the property has been violated somewhere during that trace.

## 4.4 Incremental build

To improve algorithm efficiency and support bi-finite traces we decided to construct a modified version of the previous algorithm without using the unfolding of temporal operators (functions $unfoldUntil$ and $unfoldSince$). We perform this on the assumption that temporal terms are bounded and therefore we do not compute over properties with the $unfold$ function over the terms. The temporal part is then processed incrementally using incremental evaluation over an incremental model with the non-linear satisfiability solver.

We consider any known temporal patterns over the temporal operators $\Box\hat{\phi}$, $\Diamond\hat{\phi}$ and their duals where at most we have one nested temporal operator, i.e., a temporal operator inside another temporal operator. This restriction occurs as we believe it offers enough expressive power to formulate any rational safety requirement and in order to improve efficiency. The translation function over the $LTL \times MS^{\leq}$ terms and formulas using this method is as seen in Figure 4.5. Where similarly to the *eval* function these new functions,$always, eventually, once$ and $alwaysPast$, at first are used as tag variables and later expand sequentially and incrementally alongside the trace.

For this, a state has to be taken into count as to keep track of the incremental validity over each formula inside the temporal operators once we start inlining them with the trace. The s acts as a state such as true t, false f or unknown u. The $e^i_{val}(\Box\phi)$ has the truth value *false* or *unknown*, while $e^i_{val}(\Diamond\phi)$ has the truth value *true* or *unknown*, as when evaluating infinite traces there is still a moment to come. The same happens to their duels when evaluating bi-infinite traces.

One could expect to construct the function $e^i_{val}(\phi, \Sigma, s)$ defined by

$$\begin{cases} solve\Big(conv^i_{\varphi}(\phi), encode_i(\Sigma)\Big) & if\, \phi = \hat{\phi} \\ and\Big[e^i_{val}\Big(\phi_1, \Sigma, s\Big), e^i_{val}\Big(\phi_1, \Sigma, s\Big)\Big] & if\, \phi = \phi_1 \wedge \phi_1 \\ or\Big[e^i_{val}\Big(\phi_1, \Sigma, s\Big), e^i_{val}\Big(\phi_1, \Sigma, s\Big)\Big] & if\, \phi = \phi_1 \vee \phi_1 \\ implies\Big[e^i_{val}\Big(\phi_1, \Sigma, s\Big), e^i_{val}\Big(\phi_2, \Sigma, s\Big)\Big] & if\, \phi = \phi_1 \rightarrow \phi_2 \\ ite\Big[s = u, e^i_{val}\Big(\phi, n(\Sigma), c_{\top}\Big(s, e^i_{val}(\phi_1, n(\Sigma), u)\Big)\Big), f\Big] & if\, \phi = always(\phi_1) \\ ite\Big[s = u, e^i_{val}\Big(\phi, n(\Sigma), c_{\bot}\Big(s, e^i_{val}(\phi_1, n(\Sigma), u)\Big)\Big), t\Big] & if\, \phi = eventually(\phi_1) \\ ite\Big[s = u, e^i_{val}\Big(\phi, p(\Sigma), c_{\bot}\Big(s, e^i_{val}(\phi_1, p(\Sigma), u)\Big)\Big), t\Big] & if\, \phi = once(\phi_1) \\ ite\Big[s = u, e^i_{val}\Big(\phi, p(\Sigma), c_{\top}\Big(s, e^i_{val}(\phi_1, p(\Sigma), u)\Big)\Big), f\Big] & if\, \phi = alwaysPast(\phi_1) \end{cases}$$

where $\phi \in \varphi$ a formula, and $\hat{\phi}$ specifies a formula without temporal operators, $\Sigma$ is a bi-infinite trace with $n, p$ operators, and $s \in \mathfrak{S}$ a symbol. $\mathfrak{S}$ denotes the set $\{t, f, u\}$;
$ite$ : defines the if-then-else function where if $n$ or $p$ fail to return an event and $s = u$ the function returns u;
$and : \mathfrak{S} \times \mathfrak{S} \mapsto \mathfrak{S}$ implements the conjunction over the pairs $(\{t, f\}, \{t, f\})$ and converts the pair $(f, u)$ to f, $(t, u)$ to u and $(u, u)$ to u;
$or : \mathfrak{S} \times \mathfrak{S} \mapsto \mathfrak{S}$ implements the disjunction over the pairs $(\{t, f\}, \{t, f\})$ and converts the pair $(f, u)$ to u, $(t, u)$ to t and $(u, u)$ to u;
$implies : \mathfrak{S} \times \mathfrak{S} \mapsto \mathfrak{S}$ implements the implication over the pairs $(\{t, f\}, \{t, f\})$ and converts the pairs $(f, u)$ to t, $(t, u)$ to u, $(u, f)$ to u, $(u, t)$ to t and $(u, u)$ to u;
$c_{\top} : \mathfrak{S} \times \{t, f\} \mapsto \mathfrak{S}$ converts the pair $(u, f)$ to f and u otherwise, and $c_{\bot} : \mathfrak{S} \times \{t, f\} \mapsto \mathfrak{S}$ converts the pair $(u, t)$ to t and u otherwise. The truth values applied are followed by Stephen Cole Kleene's strong logic of indeterminacy three valued logic [25].

Terms have no free variables and no assumptions have to be given for the incremental evaluation, the reason why it simplifies *implies*, *and* and *or* functions. A spatial variable maintains its form regardless of where it is evaluated.

Function $solve : \mathfrak{L} \times \mathfrak{L} \mapsto \{t, f\}$ solves an expression in $FOL_{\mathbb{R}}$ assuming another expression in $FOL_{\mathbb{R}}$, namely the encoding of bi-finite trace. Note that $\bigcirc$ temporal operator and its dual are already in-lined and do not need to be incrementally evaluated.

## Summary

Once pre-processed and put in data structures the information adhered from the simulator our tool starts the process of generating the monitor. With a property that represents a safety rule in the form of the logic fragment $LTL \times MS^{\leq}$ our tool processes it passing it through an algorithm that together with the information from the simulator produces a model to be checked by the SMT solver. This process while applied for the finite flow of time represented by the trace will produce a final verdict indicating the correctness of the property for that time period. Using the incremental method one can also evaluate a property over a bi-infinite trace, however, this may lead to inconclusive results.

# 5

# Tool Evaluation

In this chapter we monitor the properties described in chapter 3 using the decision process from chapter 4, and check for that monitors performance. The scenario used is the one presented in chapter 3. We monitor the properties under the assumption of finite traces, more precisely simulated traces captured by an ADS on the simulated environment CARLA [11, 7] with the same layout as our empirical scenario. Besides these simulated traces we also tested some empirical traces contained in the same scenario restrictions. For each property multiple evaluations were made with both the empirical and simulated traces in order to test the performance, scalability, and validation of the monitor.

To measure the scalability of monitoring we test each property with different size traces to measure the correlation between performance and the scale of the traces. For our evaluation we also wanted to test the monitoring of both positive and negative outcomes of each property.

## 5.1 Closed loop testing

The monitor is built independent to the ADS having no direct impact with the system itself as seen in Figure 5.1. The system evolves around the simulation of a specific scenario, that feeds it with its observations. According to the observations the system produces actions for its agents on the simulator, that in turn change the development of the simulation producing new observations and the cycle continues. For example, as a car approaches a stop sign the system has to indicate the car, i.e. it's agent, to stop. In this case the observations from the simulator will show that the car is indeed stopping.

The monitor receives the observations from the simulator as well, these are transformed into a trace. However in contrast to the ADS the monitor doesn't take any actions towards the agents, it only observes. It then uses the traces it produces in order to check the correctness of a certain property. Ultimately producing a verdict that indicates whether the property is maintained.

There are two ways the monitor can receive a trace that indicates if we are producing an offline or online runtime monitor. An offline monitor consists in monitoring a property

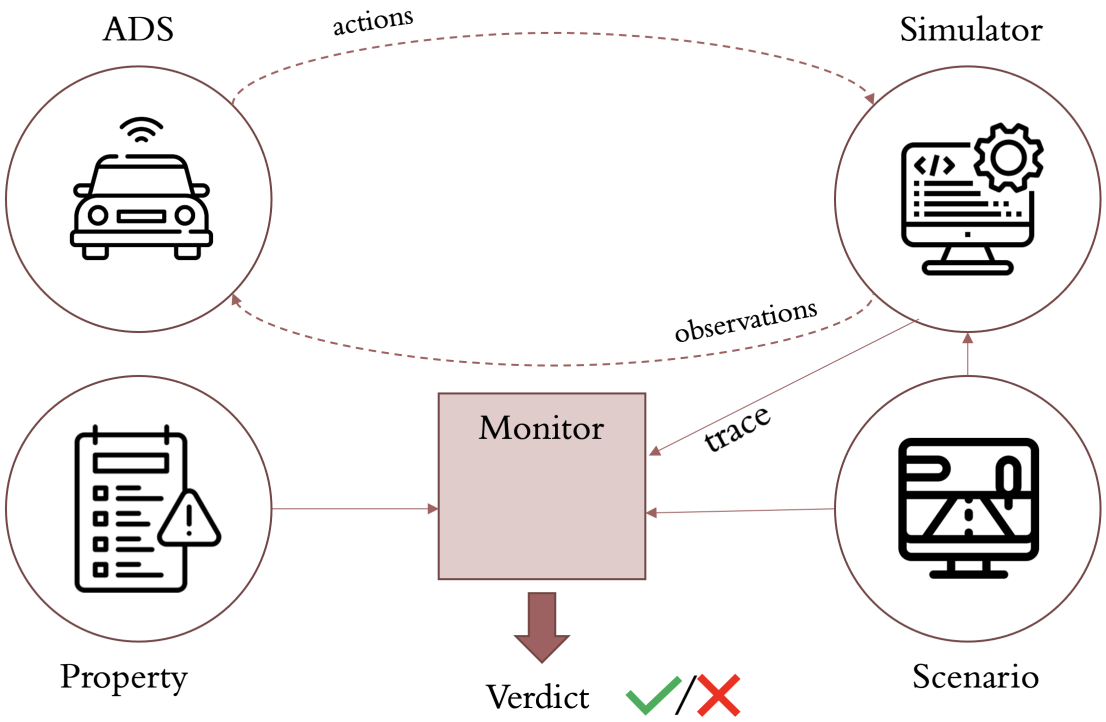



Figure 5.1: Closed Loop

with the whole trace information on a log file. The online monitor incrementally produces its trace as receiving new observations from the simulator.

## 5.2 Evaluation methods

We defined two evaluation methods for a monitor to check safety properties. Since we are monitoring safety properties of a safety-critical system, this means the properties have to be held always, throughout the execution of the system. In order to monitor a property a trace containing incremental information about the dynamic objects of the simulation is essential. The proposed methods differ on how the models representing our properties are created making use of that trace, being that the major difference is in the way each method treat the temporal formulas on our properties and model them.

### 5.2.1 Unroll method

The unroll method consists in building a bounded model of our property for the first $n$ trace events. It unrolls the sequence described in section 4.2.1 for all temporal operators in the property $n$ times. The user has the option to indicate the value he desires for this bound, being that if it is set to 10, for example, the monitor will build a model unrolling all its temporal operators 10 times. This however forces us to monitor a bounded model,

therefore a finite trace or part of it because we will only be checking for the first $n$ positions of the trace.

This method represents the direct appliance of our algorithm, and therefore the method that allows us to implement a more expressive approach. Notwithstanding, its usage may have a direct negative performance results with the scaling of the bounding value since the model increases with each tracked event.

### 5.2.2 Incremental method

The incremental method consists in creating one model for each iteration over the trace and verifying the validity of such model. This method does not apply the unroll sequence for the temporal operators, as the unroll method does. Instead it checks the correctness of the formulas on the temporal operators, such as *always*, *eventually*, and their respective duals, for each event iteration bounding the *unroll* function to the value of one. It uses our non-linear satisfiability solver to check the formula and keep track to an incremental operator correctness state that may have one of three states:

- Unknown;
- True; and
- False.

This method allows us to monitor bi-infinite traces using a variable to define the current incremental state of the monitor. While the state of a temporal operator is still unknown the monitor will expect the next or previous event, depending on the operator, to re-model the formula and update the incremental state. If no new event is given past a timestamp this may lead to an inconclusive evaluation of the property on the bi-infinte trace. However, our tool allows us to produce a valid evaluation for the finite prefix information provided of the infinite trace evaluating the formulas to *True* when the operator is a *always* or *alwaysPast* and *False* when the operator is a *eventually*, *once*, *until* or *since*.

This method does not need to check the whole trace to give a verdict adhering to the two maxims of a monitor, impartiality and anticipation, while the unroll method does not apply the anticipation maxim. Since we are talking about safety properties if at a certain frame the property is proven to fail then there is no need to continue monitoring this property for this trace. The same criteria applies to liveness properties, that should hold true at some point. In this case once we prove that that property holds at a certain point we do not need to check for the rest of the trace.

## 5.3 Metrics and success criteria

Within each metric we define a scale from zero to five that score the degree of accomplishment of each metric where, one is very poor and five is very good (according to the Likert

Scale) [33]. We reach success if all our metrics get a success score of four or five. These metrics are:

- **Performance:** Low temporal and spatial complexity. In an ADS, we have to consider our monitors' computational temporal aspect, i.e., if the monitors can do their computation in a reasonable time regarding the safety-critical aspects of the system. Besides that, storage is also a concern since we are talking about an embedded system producing big amounts of data that can quickly escalate [15].
- **Scalability:** Applying a logarithmic increasing in the number of events on a trace, and our throughput keeps hold.
- **Validation:** Our monitors produce the expected result. We can measure this metric by the monitors' success rate. To obtain the success rate of a monitor, we compare the correct verdicts with the incorrect ones as explained in [26]. With the growth of the level of autonomy of an ADS, the more responsibility the system gains. Therefore, the importance of correct verification of safety requirements increases.
- **Expressiveness:** It is possible to translate all safety requirements into properties of our formal language.

Regarding the measure of the *Performance* we obtain the time the monitors take to obtain a verdict, as well as the memory allocation size for the solver. With a view to have a good performance we need to be able to obtain a verdict faster than the sampling time from our system so that when implementing the monitors online they may keep up with the upcoming information.

*Scalability* measures the relation between the trace size and performance in order to determine the impact of larger traces and more robust ones, i.e. that include more objects. For a high rated scalability we need to maintain a high performance even with the escalation of traces.

To measure *Validation* we need to check if the monitors are producing the expected outcome. For a optimal validation for all tests the obtained verdict must be the expected verdict.

To measure *Expressiveness* we need to check if the solver is able to interpret complex models and if it is possible, using the syntax and semantics proposed, to describe complex properties.

## 5.4 Trace plot analysis

These plots serve for giving a generalized idea of the evolution of the dynamic objects throughout their trace. With the visualization of these plots we cannot assume spatial behaviour between dynamic objects since this plot was constructed using the whole trace and not only one event therefore being a trace footstep. However we can interpret it to

check if the evolution and trajectory of a object is as expected. In these plots we have four different objects:

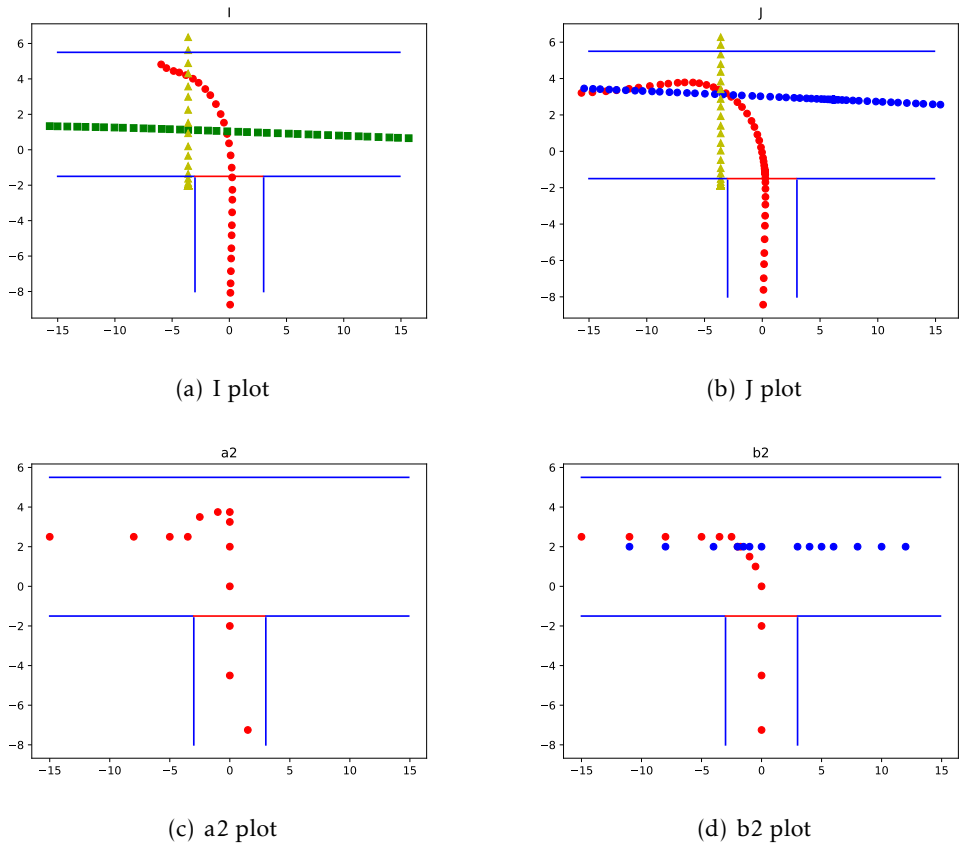
- First car, noted as the red circle;
- Second car, noted as the blue circle;
- Tram, noted as the green square; and
- Pedestrian, noted as the yellow triangle.

In order to provide an easier interpretation of the plot we included the road limits (blue lines) as well as the stop limit (red line).

Lets take a look at plot 5.2(a), showing the footsteps of trace I, one of the traces produced by our simulator. Here it's expected the car, noted as the red circles, to follow its trajectory and maintain a safety margin to the road limits, the car departs from the bottom of the plot and goes up and to the left. Yet we can easily conclude that the car is diverting its trajectory, this may be caused by many situations, it may be deviating from the pedestrian or a simple malfunction of the system. Anyway, the plot shows the last instance of the car is too close to the road limit, probably resulting in a collision with the road limits and therefore violating the first rule. As so to have a definitive conclusion on the correctness of this property for this trace its imperative to monitor the property under the assumption of this trace.

In Figure 5.2(b) that is also a plot from a simulated trace show car one not deviating from its trajectory in contrast to the previously seen trace. The blue circles that represent car two depart from the right and head to the left of the plot and are at some point overlapped to some red circles indicating the possibility of a safety-margin violation between the two cars. However, since this is a footstep representing the whole trace we can not conclude a violation of rule *1.b* only by interpreting the plot since each circle represent a mapping of the object to a time instant. For example, in the first fifty time instances of the trace, car one could have completed its trajectory and gone out of frame, and only at the beginning of instant one hundred car two departed at the right of the plot. In this case there would be no actual collision of the two cars even if the trace footsteps may indicate otherwise. It can be observed in the middle of this plot the red circles start to get closed together and later disperse, this indicates that the car is slowing down and respectively accelerating since each snapshot is equally separated time-wise. For this trace it would be interesting to monitor if the cars actually violate rule *1.b* and if car one stops at the stop sign or on top of the box junction, referring to rules *3* and *2.a* respectively.

The next two plots represent empirical traces. In 5.2(c) we can see car one departs from the bottom of the plot and must maintain in its trajectory and keep a safety margin toward the road limits. Both these rules can be expected to fail just looking at the footsteps of the car. Its first appearance seems to be close to the lateral road limits, yet it depends




(a) I plot

(b) J plot

(c) a2 plot

(d) b2 plot

Figure 5.2: Trace Footstep examples

on the safety-margin value to determine whether that rule is violated, if we check the rule for a large safety-margin it will most certainly fail, however a smaller safety-margin may be enough for the rule to be satisfied. We also observe that towards the middle of the intersection there is a minor detour that may be caused by many possible situations. Nonetheless it may be enough for the car to run off its trajectory therefore violating rule *1.a*.

Figure 5.2(d) shows both cars and a possible violation of rule *1.b* since it looks like the safety-margin between both cars is not maintained. We can also see a deceleration of both cars entering the intersection of the junction. We know this since every event from the trace is equally dispersed on a linear time flow, therefore if two consecutive projections of an object are closer together it means its velocity is lower.

## 5.5 Results

We created a table showing the evaluation results we obtained, for each method used, i.e. unroll and incremental for two types of traces, empirical and simulated. The table contains:

1. the rule we are evaluating, therefore the property they represent;
2. how many temporal and spatial operators the property contain;
3. the trace used for the evaluation;
4. the trace size;
5. the solvers' time and memory to check the model/s;
6. the runtime of the monitor; and,
7. the throughput in frames per second.

For each rule described in chapter 3 we evaluated a set of different traces. The rule we are testing can be seen under the *Rule* column where ($\|\rho\|$,$\|\varphi\|$) indicate how many temporal($\varphi$) and spatial($\rho$) operators the formula that describes the rule contains, these values represent the complexity of the property, being that a property with more spatial and temporal operators is more complex. Noticing that the complexity for solving temporal operators is higher than the complexity for solving spatial operators.

These results represent an offline runtime monitoring for the properties as the traces used are supplied as log files. There are two distinctive types of traces used for these evaluations, real traces captured from the simulator with a sampling of approximately ten frames per second, and empirical traces with four frames per second created to give a positive and a negative monitors' verdict for each property. The names of each trace are presented under the column $\Sigma$. Trace a1, a2, b1, b2, c1, c2, d1, e1, e2, e3 are the empirical traces while the others, F, G, H, I, J and K are the traces captured from the simulator. All of them provided as a .json file under the same structure. In front of the trace name, in ($\|\Sigma\|$) we have the trace size which indicates the number of events on that trace, an important factor to measure the scalability of our computation.

The *Solver* column presents the time(*Time*), in seconds, it takes for the solver to check the models and give a verdict. However this is not the actual time the monitor takes to check the property since we do not include the time the script takes to write these models. In case of the unroll method this result is the time it takes to check one single model, on the other side, the incremental method, this value represents the sum of checking many models. Besides the time it presents the memory(*Mem.* allocation size in MB during the solvers execution. As for the time value, for the unroll method, this value represents the memory of a single model while for the incremental method it represents the average memory of the many models checked.

In *RT*, an abbreviation for run time, we get the time the monitor took to produce its verdict. It is the sum of the time it takes to build up the models and the time it takes to solve them, i.e. the value under *Time*.

Finally, in order to measure the performance for every test we have the *FPS* column indicating the throughput in frames per second. These values correspond to the number

Table 5.1: Evaluation Results

| | | | Unroll | | | | Incremental | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Rule($\|\|\rho\|\|$,$\|\|\varphi\|\|$) | $\Sigma(\|\|\Sigma\|\|)$ | Solver | | RT | FPS | Solver | | RT | FPS |
| | | | Time | Mem | | | Time | Mem | | |
| Empirical | 1.a (2,2) | a1(13) | 0.91 | 4.48 | 1.26 | 10,32 | 0.21 | 3.81 | 0.38 | 34,21 |
| | 1.a (2,2) | a2(13) | 0.07 | 4.08 | 0.43 | 30,23 | 0.15 | 3.80 | 0.27 | 48,15 |
| | 1.b (1,1) | b1(13) | 0.01 | 3.07 | 0.09 | 144,44 | 0.07 | 3.75 | 0.17 | 76,47 |
| | 1.b (1,1) | b2(13) | 0.03 | 3.09 | 0.13 | 100,00 | 0.05 | 3.76 | 0.11 | 127,27 |
| | 2.a (1,3) | c1(13) | 0.13 | 3.56 | 0.27 | 48,15 | 0.11 | 3.81 | 0.21 | 61,90 |
| | 2.a (1,3) | c2(13) | 0.12 | 3.52 | 0.25 | 52,00 | 0.04 | 3.82 | 0.07 | 200,00 |
| | 2.b (1,3) | d1(13) | 0.21 | 3.52 | 0.40 | 32,50 | 0.10 | 3.82 | 0.22 | 63,64 |
| | 3. (3,6) | e1(14) | 0.48 | 7.29 | 2.29 | 6,11 | 0.21 | 3.77 | 0.50 | 28,00 |
| | 3. (3,6) | e2(13) | 0.53 | 7.35 | 1.97 | 6,60 | 0.10 | 3.79 | 0.21 | 61,90 |
| | 3. (3,6) | e3(15) | 1.30 | 7.74 | 3.42 | 4,39 | 0.21 | 3.77 | 0.51 | 29,41 |
| | Average | | 0,38 | 4,77 | 1,05 | 43,47 | 0,12 | 3,79 | 0,26 | 73,10 |
| Simulator | 1.a (2,2) | F(51) | 26.32 | 13.05 | 28.17 | 1,81 | 1.36 | 3.85 | 2.20 | 27,73 |
| | 1.a (2,2) | I(19) | 0.17 | 5.03 | 0.65 | 29,23 | 0.37 | 3.83 | 0.59 | 32,20 |
| | 1.a (2,2) | K(32) | 9.00 | 8.03 | 9.93 | 3,22 | 0.75 | 3.85 | 1.19 | 30,25 |
| | 1.b (1,1) | F(56) | 0.08 | 4.85 | 1.20 | 46,67 | 0.35 | 3.79 | 0.76 | 73,68 |
| | 1.b (1,1) | G(62) | 0.20 | 5.19 | 1.34 | 46,27 | 0.46 | 3.79 | 0.89 | 69,66 |
| | 2.a (1,3) | F(56) | 2.90 | 7.24 | 4.04 | 13,86 | 0.53 | 3.84 | 0.99 | 56,57 |
| | 2.a (1,3) | H(126) | 5.54 | 7.81 | 10.18 | 12,38 | 1.27 | 3.89 | 2.29 | 55,02 |
| | 2.a (1,3) | J(87) | 3.01 | 6.77 | 4.41 | 19,73 | 0.38 | 3.87 | 0.61 | 142,62 |
| | 3. (3,6) | G(62) | 76.23 | 69.82 | 166.63 | 0,37 | 1.03 | 3.84 | 2.21 | 28,05 |
| | 3. (3,6) | H(117) | 220.66 | 60.13 | 794.95 | 0,15 | 2.27 | 3.81 | 4.90 | 23,88 |
| | 3. (3,6) | J(107) | 339.90 | 72.18 | 774.07 | 0,14 | 1.93 | 3.79 | 4.22 | 22,27 |
| | Average | | 62,18 | 23,65 | 163,23 | 15,80 | 0,97 | 3,83 | 1,90 | 51,08 |

of frames per second it took the monitor to get the verdict for the correctness of the property, as so we divide the trace size by the time it took the monitor to produce its verdict.

The results have been obtained using a MacBook Pro with a 2,6GHz Intel Core i7 processor and a 16GB 2667 MHz DDR4 memory, running macOS Big Sur.

### 5.5.1 Analysis

An overall conclusion than we can take looking at our results table 5.1 in the column with the results from the unroll method, in particular the solver time column is that the the more complex the property is, more specifically, the more temporal operators it contains, lower is the performance of the monitor. We can notice also that the memory occupied by these solver are bigger in the more complex properties.

For the larger traces it is also common to have a more expensive monitor when checking for properties that produce a positive verdict.

Looking at table for the unroll methods' performance with the empirical traces we conclude that almost all other tests had a throughput above ten frames per second excluding rule *3* for its temporal complexity. This indicates that with a sampling time of ten frames per second the monitors could keep up with the sampling rhythm.

To sum up, the unroll method, as it is, does not look optimized enough to monitor

more complex properties that include nested temporal constrains. This is easily justified by the way we unroll the temporal operators into a sequence building a singular model for the solver, that in its default settings cannot produce a result in elementary time.

When looking at our results table 5.1 within the incremental method results the focus is the low run time results in comparison with the unroll method. Meaning that the monitor did not take that long to reach its final verdict. However we can still notice a relatively slower response time for rule *3* in comparison to the other rules as its run time is above average. It is justified by the complexity of the rule itself in comparison to all others as it has nested temporal operators. Nonetheless the run time values obtained for this rule is where we notice a bigger divergence to the ones in the unroll method.

We can also see a significant temporal difference when comparing results with a positive verdict to results with negative verdict for the same rule. Which is justified using the incremental method because as we reach a unsatisfactory model we stop evaluating the rule right there as the property has already been violated. Which means that normally it takes less frames to reach a negative verdict than a positive one when monitoring safety properties on two traces with the same size. When monitoring liveness properties it is the opposite, it takes less frames to reach a positive verdict than a negative verdict. This is caused by the anticipation axiom we compliment using this monitoring method for safety and liveness properties.

When interpreting the throughput for the incremental method we can check an overall good monitor performance for all rules, with rule *3* getting the worst throughput values, as expected. Nonetheless with these result we may conclude that even for more complex properties as our rule *3*, this monitoring method would be able to keep up to an online monitor with sampling times up to twenty frames per second which is an acceptable value for the instrumentation of a real ADS.

Comparing these two methods it is obvious that for more complex properties an incremental monitoring approach is more reasonable as seen for rule *3*, yet for simpler rules without nested temporal operators the results do not differ significantly. However the incremental method seems to be at all times the most reliable method.

These results did not surprise us as the incremental method allows us to control each instance of the monitoring better, offering the possibility to abort the monitoring when the property has been violated or maintained with no further investigation needed. On the other hand the unroll method needs to model a whole trace to find its verdict. This is, the incremental method respects both monitoring axioms as the unroll method does not adhere to the anticipation axiom.

For online runtime monitoring the incremental method is the best choice as it may check for the correctness of a certain property in bi-infinite traces. As opposed to the unroll method which is only able to check finite traces. As the performance values from the incremental method suggest, for a system whose instrumentation time is twenty frames per second or lower, the monitor is capable to keep up with the sampling time.

For offline runtime monitoring, both methods can be applied with the pros and cons

described throughout this section.

To evaluate our tool we followed the metrics described in section 5.3 and obtained the following results for each of them:

Evaluating the overall performance of our tool we conclude it produces the expected results in reasonable time with the machine used being able to monitor an ADS. However with weaker machines the result may not be sufficient for a good monitor performance. Specially when applied to safety-critical systems. Therefore an effort must be made to improve these result and get an optimal performance which online monitoring requires. As we may only evaluate our tool for the results we obtained, we give it a score of four.

In terms of scalability for the incremental method the performance values do not vary much when comparing larger and smaller traces as seen in the average throughput difference for empirical and simulated traces. But when using the unroll method the scale of the trace has a major impact on the performance we get. We believe this tool has yet possible optimizations regarding this aspect and therefore give it a score of four in the likert scale [33].

The validation of our tool is known to be optimal for the tests we implemented, as for all traces, using both methods, we obtain the expected verdict. Therefore we can say we have an optimal monitors' success rate.

As for the expressiveness of our tool it is also optimal as our tool was able to formalise and evaluate all the rules we proposed to monitor.

Many optimizations could be still applied to reduce the solvers time for evaluating a model as discussed in section 6.2. The improvement in the data structures used and their iteration may also lead to a meaningfully performance optimization.

## Summary

We evaluated all the properties described in chapter 3 using the algorithm proposed. We performed multiple evaluations to test both methods we defined, to test the scalability of our monitors, the performance of the monitors and the validity of the tool. For that we used different traces, with different sizes, from empirical to simulated, and expected to produce positive and false verdicts. This allowed us to get a large amount of result and statistics useful to measure the performance and correctness of our tool. The results we obtained were rather positive as they indicate our tool works in, what we believe to be an acceptable time. It may yet struggle to monitor more complex properties in elementary time.

# 6

# Conclusion

In this chapter we present our conclusion on what we were able to accomplish and what this tool has to offer. We analyse possible future work for optimizing our tool as well as implementing it in real time ADS.

## 6.1 Accomplishments

We started by following a scenario-based approach for defining the appropriate formalism for representing a trace and scenario and for defining a set of safety rules applicable in our use case for an ADS. So we imagined an ADS in a urban intersection and drove a hazard analysis to check which requirements would make sense to monitor in a ADS in order for it to be safe and compliant to the traffic rules. Once gathered a set of traffic rules we had to formalize them so we could monitor them, becoming our use-cases. Road safety rules normally dictate behaviours we, as a driver, or the car as a more objectified view should follow at all times or on behalf of some kind of specific situation. We had to be able to formally describe these behaviours using a logic language, so we searched the best way to formally represent behaviours that our objects should have during a flow of time and came out with $LTL \times MS^{\leq}$. A specification language that could represent the behaviours we intended reasoning over space and time, offering a vast expressive power capable of defining simple and complex properties. We then concluded that the best way to formally represent our objects would be as referring to them as closed regions offering us the possibility to reason over metric spaces and operate over distances. Following this structure we were able to formally define properties that represent spatial behaviours over infinite and finite traces that may be represented a infinite flow of time.

We then created an algorithm that accepts expressions under a predefined grammars syntax and semantics, build using the format of our specification language $LTL \times MS^{\leq}$ that reasons over space and time, used to represent our properties. This algorithm transform this $LTL \times MS^{\leq}$ expression into a SMT model resorting to $FOL_{\mathbb{R}}$ formulas that, together with the information gathered from a trace also encoded in $FOL_{\mathbb{R}}$, is passed through a Z3 solver indicating the satisfiability of our model. This tool is adapted to create and

evaluate models under the assumption of an incremental trace.

We are able to run monitors that check properties over a trace obtained by log files performing an offline runtime monitoring of a ADS. Although we did not test online runtime monitoring, our tool was build to accept and evaluate online monitors given any trace fed to the monitor.

Finally we implemented a tool that generates these runtime monitors checking any property respecting our grammar in the assumption of a trace passed into the tool. This tool allows the user to decide which method they wish to evaluate their monitor, differing in the way the monitoring model is generated. The unroll method only accepts finite traces where the unroll sequence is bounded to a value passed through the tool. The incremental method, which produced better results for the overall performance of the tool, accepts bi-infinite traces and is the better option for online runtime monitoring. An example of the usage of this tool may be seen in appendix 1.

## 6.2 Future Work

The current methods used to evaluate the monitor are valid but their performance has yet to be optimized mainly due to the unfolding used in the algorithm on behalf of the $\mathcal{U}$ and $\mathcal{S}$ operators that if strictly applied to finite traces does not adhere to the the monitors maxim of *anticipation*. This may lead to a more exhaustive process than actually needed to monitor a property.

As we can see in the results of section 5.5 for both methods the throughput when evaluating smaller traces is normally higher than for bigger traces. That difference can be more noticeable in the unroll method as expected. For this reason one way we could optimize results for this method would be partitioning the traces into smaller sub-traces and apply the unroll method on those smaller traces keeping track of a general correctness state updated at each evaluation of a sub-trace. We could also create and apply a machine learning algorithm that would apply the best bounding value depending on the trace size and monitor a property using the unroll method with that value. This way even applying the unroll method we would be applying the *anticipation* maxim of monitoring even if not as accurate as in the incremental method.

Z3 SMT solver allows us to configure its solver into using different types of satisfiability cores and decision algorithms applied to their models. We could look into it and configure our solve to use the best algorithms for the type of models we intend to verify. For our results we used the default configurations on the non-linear sat solver.

As for the expansion of our tool, we will eventually want for users to be able to provide a scenario in openScenario format, and for our tool to be able to recognize and encode the static elements of that scenario in order to evaluate safety properties in any provided scenario. As for now we can only evaluate properties in our own empirical scenario.

For now, our tool results evaluates properties over traces passed in as log files, they can be empirical traces or traces produced by a real or simulated environment as long as

presented in the predefined structure. Future works passes through adapting the *encode* function in order to encode incoming instrumentation from the simulator, for example in a stream form, and testing the tool with online monitoring of an ADS.

To finalize this tool a graphic user interface to simplify the process of the tools utilization would be of great work. A GUI where we could pass in an $LTL \times MS^{\leq}$ expression as input, a scenario and the trace and we could see a plot representing the trace footstep evolution while the property was being evaluated showing its final verdict when terminated.

# Bibliography


[1] M. Aiello, I. Pratt-Hartmann, and J. van Benthem. "Spatial Logic + Temporal Logic = ?" In: *Handbook of Spatial Logics*. Ed. by M. Aiello, I. Pratt-Hartmann, and J. van Benthem. Springer, 2007, pp. 497–564. DOI: 10.1007/978-1-4020-5587-4_1. URL: https://doi.org/10.1007/978-1-4020-5587-4%5C_1 (cit. on pp. 14–16, 18–21, 37).

[2] B. Alpern and F. B. Schneider. "Defining Liveness". In: *Inf. Process. Lett.* 21.4 (1985), pp. 181–185. DOI: 10.1016/0020-0190(85)90056-0. URL: https://doi.org/10.1016/0020-0190(85)90056-0 (cit. on p. 10).

[3] N. Aréchiga. "Specifying Safety of Autonomous Vehicles in Signal Temporal Logic". In: *2019 IEEE Intelligent Vehicles Symposium, IV 2019, Paris, France, June 9-12, 2019*. IEEE, 2019, pp. 58–63. DOI: 10.1109/IVS.2019.8813875. URL: https://doi.org/10.1109/IVS.2019.8813875 (cit. on p. 23).

[4] C. W. Barrett and C. Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Model Checking*. Ed. by E. M. Clarke et al. Springer, 2018, pp. 305–343. DOI: 10.1007/978-3-319-10575-8_11. URL: https://doi.org/10.1007/978-3-319-10575-8%5C_11 (cit. on p. 11).

[5] B. Bennett et al. "Multi-Dimensional Modal Logic as a Framework for Spatio-Temporal Reasoning". In: *Appl. Intell.* 17.3 (2002), pp. 239–251. DOI: 10.1023/A:1020083231504. URL: https://doi.org/10.1023/A:1020083231504 (cit. on p. 24).

[6] D. Bjørner and M. Henson. *Logics of Specification Languages*. Monographs in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 2007. ISBN: 9783540741077. URL: https://books.google.pt/books?id=pGrD8NtyR%5C_EC (cit. on p. 13).

[7] A. Dosovitskiy et al. "CARLA: An Open Urban Driving Simulator". In: *Proceedings of the 1st Annual Conference on Robot Learning*. Ed. by S. Levine, V. Vanhoucke, and K. Goldberg. Vol. 78. Proceedings of Machine Learning Research. PMLR, 13–15

Nov 2017, pp. 1–16. URL: https://proceedings.mlr.press/v78/dosovitskiy17a.html (cit. on p. 48).

[8] E. A. Emerson. "Temporal and Modal Logic". In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Ed. by J. van Leeuwen. Elsevier and MIT Press, 1990, pp. 995–1072. DOI: 10.1016/b978-0-444-88074-1.50021-4. URL: https://doi.org/10.1016/b978-0-444-88074-1.50021-4 (cit. on p. 14).

[9] D. M. Gabbay. *Many-Dimensional Modal Logics: Theory and Applications*. Elsevier North Holland, 2003 (cit. on p. 21).

[10] D. Gabelaia et al. "Combining Spatial and Temporal Logics: Expressiveness vs. Complexity". In: *CoRR* abs/1110.2726 (2011). arXiv: 1110.2726. URL: http://arxiv.org/abs/1110.2726 (cit. on pp. 19, 20, 22, 24).

[11] B. Gassmann et al. "Integration of Formal Safety Models on System Level Using the Example of Responsibility Sensitive Safety and CARLA Driving Simulator". In: *Computer Safety, Reliability, and Security. SAFECOMP 2020 Workshops*. Ed. by A. Casimiro et al. Cham: Springer International Publishing, 2020, pp. 358–369. ISBN: 978-3-030-55583-2 (cit. on p. 48).

[12] R. Gerth et al. "Simple on-the-fly automatic verification of linear temporal logic". In: *Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, June 1995*. Ed. by P. Dembinski and M. Sredniawa. Vol. 38. IFIP Conference Proceedings. Chapman & Hall, 1995, pp. 3–18 (cit. on p. 15).

[13] M. P. E. Heimdahl and C. L. Heitmeyer. "Formal Methods For Developing High Assurance Computer Systems: Working Group Report". In: *2nd Workshop on Industrial-Strength Formal Specification Techniques (WIFT '98), October 20-23, 1998, Boca Raton, FL, USA*. IEEE Computer Society, 1998, p. 60. DOI: 10.1109/WIFT.1998.766298. URL: https://doi.org/10.1109/WIFT.1998.766298 (cit. on p. 9).

[14] H. Kamp. "Tense Logic and the Theory of Linear Order". PhD thesis. Ucla, 1968 (cit. on p. 16).

[15] A. Kane. "Runtime Monitoring for Safety-Critical Embedded Systems". PhD thesis. Carnegie Mellon University, July 2018. DOI: 10.1184/R1/6721376.v1. URL: https://kilthub.cmu.edu/articles/thesis/Runtime%5C_Monitoring%5C_for%5C_Safety-Critical%5C_Embedded%5C_Systems/6721376/1 (cit. on pp. 1, 2, 7, 9, 12, 51).

[16] J. Kapinski et al. "Simulation-guided approaches for verification of automotive powertrain control systems". In: *American Control Conference, ACC 2015, Chicago, IL, USA, July 1-3, 2015*. IEEE, 2015, pp. 4086–4095. DOI: 10.1109/ACC.2015.7171968. URL: https://doi.org/10.1109/ACC.2015.7171968 (cit. on p. 23).

[17] J. C. Knight. "Safety critical systems: challenges and directions". In: *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*. Ed. by W. Tracz, M. Young, and J. Magee. ACM, 2002, pp. 547–550. DOI: 10.1145/581339.581406. URL: https://doi.org/10.1145/581339.581406 (cit. on pp. 1, 8, 9).

[18] A. Kurucz, F. Wolter, and M. Zakharyaschev. "Modal Logics for Metric Spaces: Open Problems". In: *We Will Show Them! Essays in Honour of Dov Gabbay, Volume Two*. Ed. by S. N. Artëmov et al. College Publications, 2005, pp. 193–108 (cit. on p. 18).

[19] O. Kutz et al. "Logics of Metric Spaces". In: *ACM Trans. Comput. Logic* 4.2 (Apr. 2003), pp. 260–294. ISSN: 1529-3785. DOI: 10.1145/635499.635504. URL: https://doi.org/10.1145/635499.635504 (cit. on p. 18).

[20] M. Leucker and C. Schallhart. "A brief account of runtime verification". In: *J. Log. Algebraic Methods Program.* 78.5 (2009), pp. 293–303. DOI: 10.1016/j.jlap.2008.08.004. URL: https://doi.org/10.1016/j.jlap.2008.08.004 (cit. on pp. 9, 12, 13).

[21] N. G. Leveson. *Safeware - system safety and computers: a guide to preventing accidents and losses caused by technology*. Addison-Wesley, 1995. ISBN: 978-0-201-11972-5 (cit. on pp. 9, 10).

[22] T. Li et al. "A spatio-temporal specification language and its completeness & decidability". In: *J. Cloud Comput.* 9 (2020), p. 65. DOI: 10.1186/s13677-020-00209-3. URL: https://doi.org/10.1186/s13677-020-00209-3 (cit. on p. 23).

[23] W. Lim et al. "Hybrid Trajectory Planning for Autonomous Driving in On-Road Dynamic Scenarios". In: *IEEE Transactions on Intelligent Transportation Systems* 22.1 (2021), pp. 341–355. DOI: 10.1109/TITS.2019.2957797 (cit. on p. 26).

[24] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[25] J. Martin. "Classical indeterminacy, many-valued logic, and supervaluations". In: *Proceedings of the sixth international symposium on Multiple-valued logic*. 1976, pp. 115–122 (cit. on p. 46).

[26] A. Mehmed. "Runtime Monitoring for Safe Automated Driving Systems". PhD thesis. Mälardalen University, 2020. ISBN: 978-91-7485-489-3. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:mdh:diva-51850 (cit. on pp. 1, 8, 12, 13, 51).

[27] L. M. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and J. Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24. URL: https://doi.org/10.1007/978-3-540-78800-3%5C_24 (cit. on pp. 11, 38).

[28] R. M. Murray. *Lecture 3 - Linear Temporal Logic (LTL)*. URL: http://www.cds.caltech.edu/~murray/courses/afrl-sp12/L3_ltl-24Apr12.pdf (cit. on p. 15).

[29] U. Nations. *Vienna convention on road traffic*. 1968 (cit. on pp. 2, 3, 5, 25, 30).

[30] PGDL. *Código da estrada [Lei n.o 72/2013 -Diário da República n.o 169/2013]*. URL: https://www.pgdlisboa.pt/leis/lei_mostra_articulado.php?nid=1991&tabela=leis&ficha=1&pagina=1 (cit. on pp. 2, 3, 25).

[31] S. Ratschan. "Efficient Solving of Quantified Inequality Constraints over the Real Numbers". In: *CoRR* cs.LO/0211016 (2002). URL: http://arxiv.org/abs/cs/0211016 (cit. on p. 14).

[32] S. Riedmaier et al. "Survey on Scenario-Based Safety Assessment of Automated Vehicles". In: *IEEE Access* 8 (2020), pp. 87456–87477. DOI: 10.1109/ACCESS.2020.2993730. URL: https://doi.org/10.1109/ACCESS.2020.2993730 (cit. on pp. 8, 23).

[33] J. Robinson. "Likert Scale". In: *Encyclopedia of Quality of Life and Well-Being Research*. Ed. by A. C. Michalos. Dordrecht: Springer Netherlands, 2014, pp. 3620–3621. ISBN: 978-94-007-0753-5. DOI: 10.1007/978-94-007-0753-5_1654. URL: https://doi.org/10.1007/978-94-007-0753-5_1654 (cit. on pp. 51, 57).

[34] P. Schnoebelen. "The Complexity of Temporal Logic Model Checking". In: *Advances in Modal Logic 4, papers from the fourth conference on "Advances in Modal logic,"held in Toulouse, France, 30 September - 2 October 2002*. Ed. by P. Balbiani et al. King's College Publications, 2002, pp. 393–436 (cit. on pp. 11, 14).

[35] S. Shalev-Shwartz, S. Shammah, and A. Shashua. "On a Formal Model of Safe and Scalable Self-driving Cars". In: *CoRR* abs/1708.06374 (2017). arXiv: 1708.06374. URL: http://arxiv.org/abs/1708.06374 (cit. on p. 23).

[36] A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. Santa Monica, CA: RAND Corporation, 1951 (cit. on p. 14).

[37] F. Wolter and M. Zakharyaschev. "Reasoning about Distances". In: *Proceedings of the 18th International Joint Conference on Artificial Intelligence*. IJCAI'03. Acapulco, Mexico: Morgan Kaufmann Publishers Inc., 2003, pp. 1275–1280 (cit. on p. 18).

[38] F. Wolter and M. Zakharyaschev. "Spatial Reasoning in RCC-8 with Boolean Region Terms". In: *Proceedings of the 14th European Conference on Artificial Intelligence*. ECAI'00. Berlin, Germany: IOS Press, 2000, pp. 244–248 (cit. on p. 17).

# A

# Tool usage example

Here we demonstrate how to download and use our tool for monitoring a simple spatio-temporal property in a small trace.

You can download the stem tool from [releases](https://github.com/anmaped/stem-binaries/releases) tab or try ‘stem-binaries-install.sh‘ script to automatically perform the download and instalation. The python libraries are built automatically by the stem tool or you can follow our guide to manually build the tool.

The safety property we will monitor describes that two cars, car 1 and car 2 never crash. For this we want to formally describe that for the whole trace the regions that represent car 1 and car 2 never overlap.

The input file (properties/no-colision.sprop) describing our property is a ‘.sprop‘ file (stem property file extension) with the content

Listing A.1: Property in $LTL \times MS^{\leq}$ language.

```
(property
    (always (not (overlap (prop "C1") (prop "C2"))))
)
```

First, we generate the monitor using the tool argument ‘-i‘ to pass the property file in ‘LTLxMS‘ language. Then, we describe our scenario in OpenScenario format (extension file .xodr) using the argument ‘-c‘. We also include flags ‘-u‘ to use the unroll method followed by a value to define the bound we desire or ‘-l‘ to set the incremental method.

To generate the monitoring model, we type

Listing A.2: Command to generate monitor

```
./src/stem.sh -g -u 3 -i ./properties/no-colision.sprop -c
./scenario/intersection_T.xodr -o monitoring-model.smt2
```

Note! The input file ‘intersection_T.xodr‘ illustrates the static objects of our scenario. Our tool identifies the argument to generate the monitor by the flag ‘-g‘. The output file is specified by ‘-o‘ and saves the file with the monitoring model containing the property to monitor.

To simulate the monitoring model and the trace satisfiability, type:

Listing A.3: Command to simulate monitor under a certain trace

```
./src/stem.sh -s -m monitoring-model.smt2 -t
./traces/no-colision-c1-c2-trace.json -n "[(1,'C1'),(2,'C2')]"
```

The flag '-s' instructs the tool to simulate the monitoring model '-m' under the trace '-t'. We also include flags '-n' that sets a map between the spatial variables (from the monitoring model) and the elements identifiers (e.g., 'ID') of the trace. Note that without this mapping, a spatial object cannot map to variables or spatial propositions.

The trace example file is available in (traces/no-colision-c1-c2-trace.json) or can be seen below in Listing A.4 and the footprint of this trace is illustrated in Figure A.1.

Listing A.4: Test trace

```
{
    "trace": [
    {
    "eventID" : 1,
    "timestamp" : "10:00:00",
    "elements" : [
            {   "ID" : 1,
                "type" : "Car",
                "position" :  {"x" : 0 , "y" : -8},
                "region": {"type": "circle", "radius": 0.66}},
                {       "ID" : 2,
                "type" : "Car",
                "position" : {"x" : 13 , "y" : 2},
                "region": {"type": "circle", "radius": 0.66} } ]
    },
    {"eventID" : 2,
    "timestamp" : "10:00:15",
    "elements" : [
            {   "ID" : 1,
                "type" : "Car",
                "position" : { "x": 0, "y": -4.5},
                "region": {"type": "circle", "radius": 0.66}},
                {       "ID" : 2,
                "type" : "Car",
                "position" : {"x" : 6 , "y" : 2},
                "region": {"type": "circle", "radius": 0.66}}]
    },
    {"eventID" : 3,
    "timestamp" : "10:00:30",
    "elements" : [
            {   "ID" : 1,
                "type" : "Car",
                "position" : {"x": 0, "y":  -2},
                "region": {"type": "circle", "radius": 0.66}},
                {       "ID" : 2,
                "type" : "Car",
                "position" : {"x" : 3 , "y" : 2},
                "region": {"type": "circle", "radius": 0.66}} ]
    },
    {"eventID" : 4,
    "timestamp" : "10:00:45",
    "elements" : [
            {   "ID" : 1,
                "type" : "Car",
                "position" : {"x" :0, "y" : 0},
                "region": {"type": "circle", "radius": 0.66}},
                {       "ID" : 2,
                "type" : "Car",
                "position" : {"x" : 2 , "y" : 2},
                "region": {"type": "circle", "radius": 0.66}} ]
```

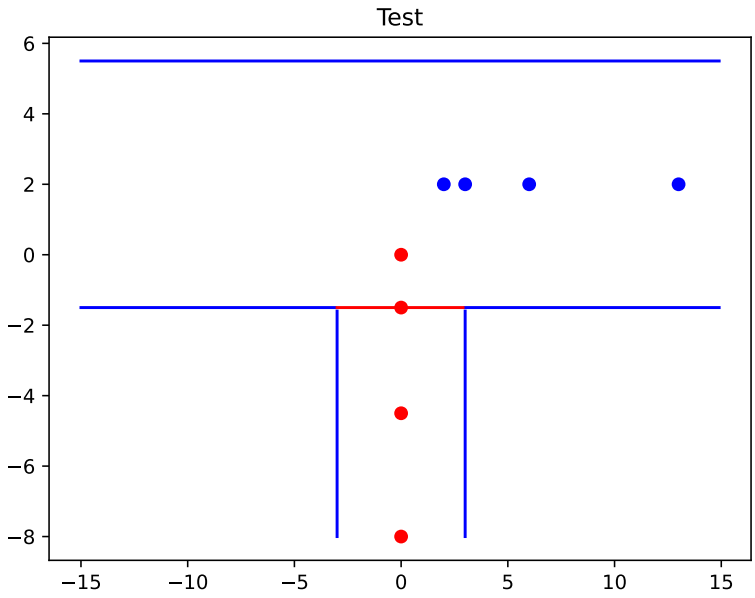



Figure A.1: Test empirical trace plot

```
    }]
}
```

After running these commands, the expected output should be similar to the one seen in Listing A.5.

Listing A.5: Shell output

```
Runtime = 0.13393616676330566
Checking satisfiability of the model... sat
(Time, Memory) = (0.014, 2.72)
```

Runtime means the time that the monitor generation process took excluding the time of the solver (in seconds). Memory (in megabytes) and time (in seconds) means the time and memory the solver spent.

If instead of the unroll method we use the incremental method running the command

Listing A.6: Command to generate monitor

```
./src/stem.sh -g -l -i ./properties/no-colision.sprop -c
./scenario/intersection_T.xodr -o monitoring-model.smt2
```

and then the command in Listing A.3, the expected output should be similar to the one seen in Listing A.7.

Listing A.7: Shell output

```
Runtime = 0.06212496757507324
Checking satisfiability of the monitoring model... sat
```

```
(Time, Memory) = (0.034999999999999996, 3.75)
The validity of this property regarding its infinite trace is yet unknown
Number of frames monitored to reach a final verdict: 3
```

In the incremental method output runtime means the time the monitor generation process took including the solver time. It evaluates the monitor presuming a infinite trace and also presuming a finite prefix of the infinite trace, passed as argument to the tool. Therefore obtaining two verdicts. It also presents the number of frames which encoding were necessary to reach a final verdict.