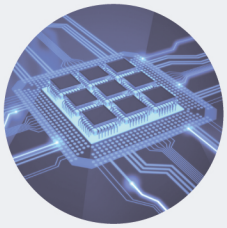
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA DE SISTEMAS COMPUTACIONAIS CRÍTICOS



Aplicação da Verificação de Modelos na Verificação Formal de Requisitos para um Sistema de Controlo de Velocidade em Ferrovias

CRISTIANO MANUEL GARCÊS COELHO
Junho de 2025

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Application of Model Checking in the Formal Verification of Requirements for a Speed Control System in Railway

Cristiano Manuel Garcês Coelho

Dissertation submitted in partial fulfilment of the requirements for the Master's degree in Critical Computing Systems Engineering

Supervisor: David Pereira

External-supervisor: Joaquim Tojal

Evaluation Committee:

President:

Luís Miguel Pinho

Members:

David Pereira

José Proença

Porto, June 28, 2025

Statement of Integrity

I hereby declare having conducted this academic work with integrity. I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

I declare that the work presented in this document is original and my own and has not previously been used for any other purpose.

I further declare that I have fully followed the Code of Good Practices and Conduct of the Polytechnic Institute of Porto.

ISEP, Porto, June 28, 2025

Abstract

Railway speed control systems are essential for the safety and efficiency of railway transportation systems, as they ensure compliance with regulations and prevent accidents. As the demand for rail transport grows, rigorous verification of these systems is becomes increasingly critical. In such projects, errors are sometimes only detected during the testing phase, leading to costs that could have been avoided if these errors were identified in the early stages of the project.

This thesis focuses on the topic of the formal verification of speed control system, specifically using model checking, to ensure that these systems meet safety requirements as well as performance standards. As the basis for the work presented in this thesis, we will adopt the EBICAB 700 architecture, and for which a small, yet representative, set of safety, operational, or regulatory requirements will be rigorously specified using well-known temporal logic languages, and verified against the system models using the model checking tools NuSMV and UPPAAL.

This thesis work aims at contributing to the effort of demonstrating the feasibility and benefits of applying formal verification to railway systems, not only in terms of safety assurance but also as a strategy for reducing late-stage development risks and increased costs. It highlights the value of incorporating formal methods into the requirements and design phases of critical system engineering.

Keywords: Speed control systems, Linear Temporal Logic, Computation Tree Logic, Model checking

Resumo

Os sistemas de controlo de velocidade ferroviária são essenciais para a segurança e eficiência dos sistemas de transporte ferroviário, pois garantem o cumprimento das regulamentações e previnem acidentes. À medida que cresce a procura pelo transporte ferroviário, a verificação rigorosa destes sistemas torna-se cada vez mais crítica. Em projetos deste tipo, os erros são por vezes detetados apenas na fase de testes, o que leva a custos que poderiam ter sido evitados se esses erros tivessem sido identificados nas fases iniciais do projeto.

Esta tese foca-se na verificação formal de sistemas de controlo de velocidade, recorrendo especificamente à técnica de model checking, para garantir que estes sistemas cumprem os requisitos de segurança, bem como os padrões de desempenho. Como base para o trabalho apresentado nesta tese, será adotada como exemplo a arquitetura EBICAB 700, para a qual será especificado um conjunto pequeno, mas representativo, de requisitos de segurança, operacionais ou regulamentares, utilizando linguagens de lógica temporal bem conhecidas, e verificados contra os modelos do sistema através das ferramentas de model checking NuSMV e UPPAAL.

O trabalho desta tese pretende contribuir para o esforço de demonstrar a viabilidade e os benefícios da aplicação da verificação formal em sistemas ferroviários, não apenas em termos de garantia de segurança, mas também como uma estratégia para reduzir riscos de desenvolvimento em fases tardias e os custos acrescidos. Destaca-se o valor da incorporação de métodos formais nas fases de requisitos e conceção da engenharia de sistemas críticos.

Palavras-chave: Sistemas de controlo de velocidade, Lógica Temporal Linear, Lógica Árvore de Computação, Verificação de modelos

Acknowledgements

I would like to thank everyone involved in making this thesis possible, namely:

- My parents, for always encouraging me to grow.
- My supervisor and external supervisor, David Pereira and Joaquim Tojal, who guided and helped me through this journey.
- My work colleagues at Critical Software, with special mention to Eduardo Pacheco and João Noronha, especially for having encouraged and motivated me throughout, without disregarding the rest of my colleagues and friends who also supported me during this journey.
- My professors over the years, without whom I wouldn't have the knowledge I have and wouldn't be able to achieve the development of this project.

To everyone involved, a sincere thank you. This phase of my life will always be remembered, not only for the extreme amount of work I undertook, but also because of everyone who supported me and made me see how important it is to have close ones to keep me together in times of stress and to achieve great things.

Table of Contents

List of Figures

List of Tables

Acronyms and Symbols

List of Acronyms

ISEP	*Instituto Superior de Engenharia do Porto*
ATP	Automatic Train Protection
ERTMS	European Rail Traffic Management System
ETCS	European Train Control System
GSM-R	Global System for Mobile Communications – Railway
CONVEL	*Controlo Automático de Velocidade*
LTL	Linear Temporal Logic
CTL	Computation Tree Logic
ATC	Automatic Train Control
ATS	Automatic Train Supervision
ATO	Automatic Train Operation
IECS	Integrated Electronic Control System
ATB NG	Automatische Trein Beïnvloeding (Next Generation)
KVB	Contrôle de Vitesse par Balises
TBL	Transmissie Baken Locomotief
LZB	Linienförmige Zugbeeinflussung
STM	Specific Transmission Module
RBC	Radio Block Centre
TSI	Technical Specification for Interoperability
BLTL	Bounded Linear Temporal Logic
MDA	Model-Driven Architecture
OBDD	Ordered Binary Decision Diagrams
SMV	Symbolic Model Verifier

CMU	Carnegie Mellon University
ITC-IRST	Institute for Scientific and Technological Research
PSL	Property Specification Language
BDDs	Binary Decision Diagrams
RTCTL	Real-Time CTL
FSM	Finite State Machine
VIS	Verification Interacting with Synthesis
SMC	Statistical Model Checking
FRET	Formal Requirements Elicitation Tool
VIS	Verification Interacting with Synthesis

1 Introduction

1.1 Context

In the railway sector, trains are subject to various factors that can lead to accidents, such as collisions and derailments. These can be caused by speeding, defective tracks, obstacles on the track, and faulty rolling stock, representing a serious threat to passenger safety. To mitigate these risks, the Automatic Train Protection (ATP) system was designed to prevent speeding and collisions. Speeding occurs when a train travels above the permitted limit on a given section of track, with these limits being established based on various track characteristics, such as its geometry, gradient, and curves. A railway is divided into blocks, with signals installed at the beginning of each block to ensure that only one train occupies a block at a time. Train collisions occur when two trains accidentally occupy the same block simultaneously. (H.Song, et al., 2018).

To manage the occupancy of the blocks, light signals are used to inform the driver about their occupancy status. These signals display different colors:

- **Green:** Indicates that at least the next two blocks are free.
- **Flashing Yellow:** Alerts the driver about the possibility of having to stop in the next two blocks.
- **Steady Yellow:** Indicates that the next block is occupied.
- **Red:** Indicates that the block is occupied and the train must stop.

These signals are essential to ensure safety and prevent train collisions.

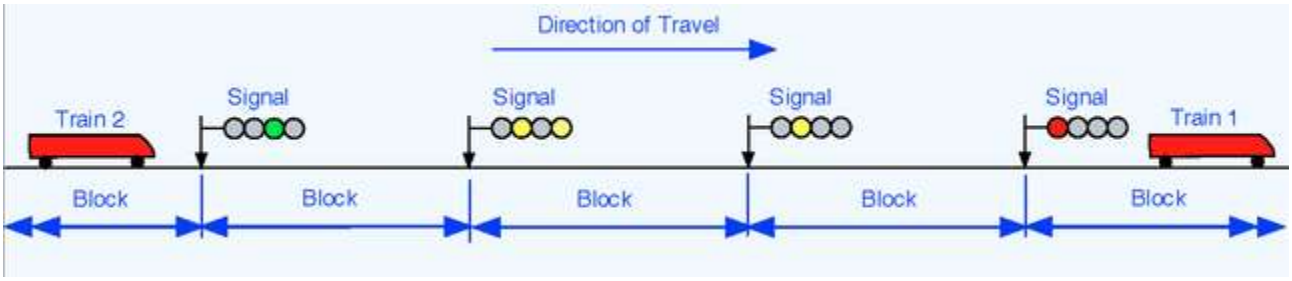


Figure 1.1: Aspect operating sequence. (Admin, 2022)

The demand for the use of trains as a means of transportation, both for passengers and goods, has increased significantly in recent decades. This growing demand required the implementation of signalling systems and speed limits, ensuring not only a safer railway system but also a more efficient one, allowing for a greater supply of this type of transportation to passengers.

Since human errors can occur, and the driver has a vast set of tasks to manage, this fact can lead to failures that result in accidents and potentially cause human casualties. It was then, due to market demand and to assist the driver, that the ATP system emerged in the 1980s. This system was widely implemented to increase the safety and efficiency of railway operations (Evans, 1996).

These systems regularly monitor track conditions, thanks to a set of sensors on the track that transmit information to the train, allowing the system to take the necessary precautions. Subsequently, the onboard system receives the information provided by the track through an antenna installed on the train, along with the train's characteristics provided by the driver, such as length, maximum speed, and brake reaction time, which will allow the system to define a braking curve for that section of the track until the next information is received (Martins, 2017).

The calculated braking curve (Figure 1.2) establishes the continuous evolution of speed, automatically activating the service or emergency brakes whenever speed limits are exceeded (Yuan, et al., 2021).

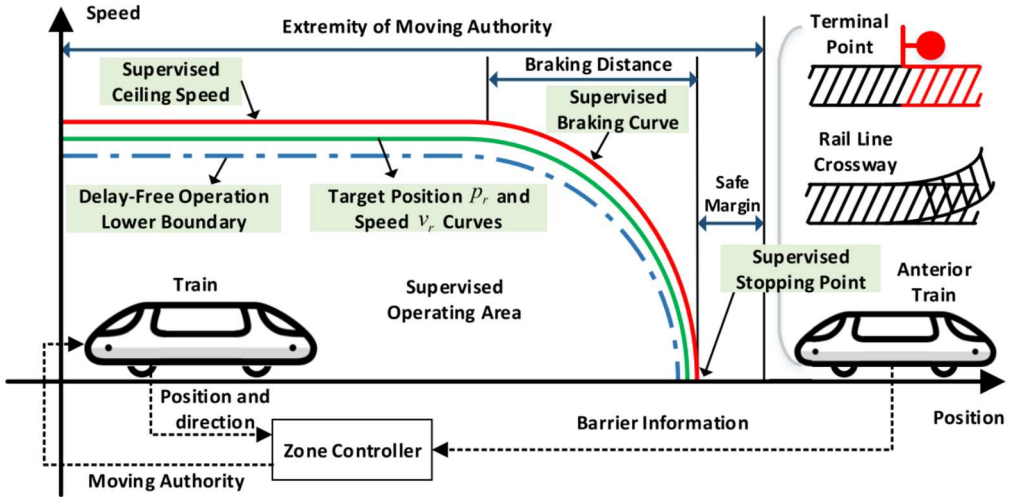


Figure 1.2: Illustration of protection constraints for automatic train operation including braking curve profile. (Yuan, et al., 2021)

Since this system is considered reliable, over the years several ATP systems have been installed in Europe, as illustrated in figure 1.3, which depicts the various ATP systems used in Europe. However, each country has adopted its own system, resulting in several differences between them, making it difficult to achieve the desired interoperability in the European railway.

As a possible solution to this problem, the European Rail Traffic Management System (ERTMS) emerged, a unique signaling and speed control system that ensures the interoperability of national railway systems.

This system allows for the reduction of acquisition and maintenance costs of signaling systems, increases the maximum speed limit of trains, and improves the level of safety in railway transport. The ERTMS is composed of the European Train Control System (ETCS), which includes an onboard signaling system with ATP, the Global System for Mobile Communications – Railway (GSM-R), and operational rules (Railways, s.d.).

In Portugal, the ATP system used is the EBICAB 700, better known as CONVEL (Controlo Automático de Velocidade), and it is installed in all passenger and freight trains. (Transportes, s.d.).

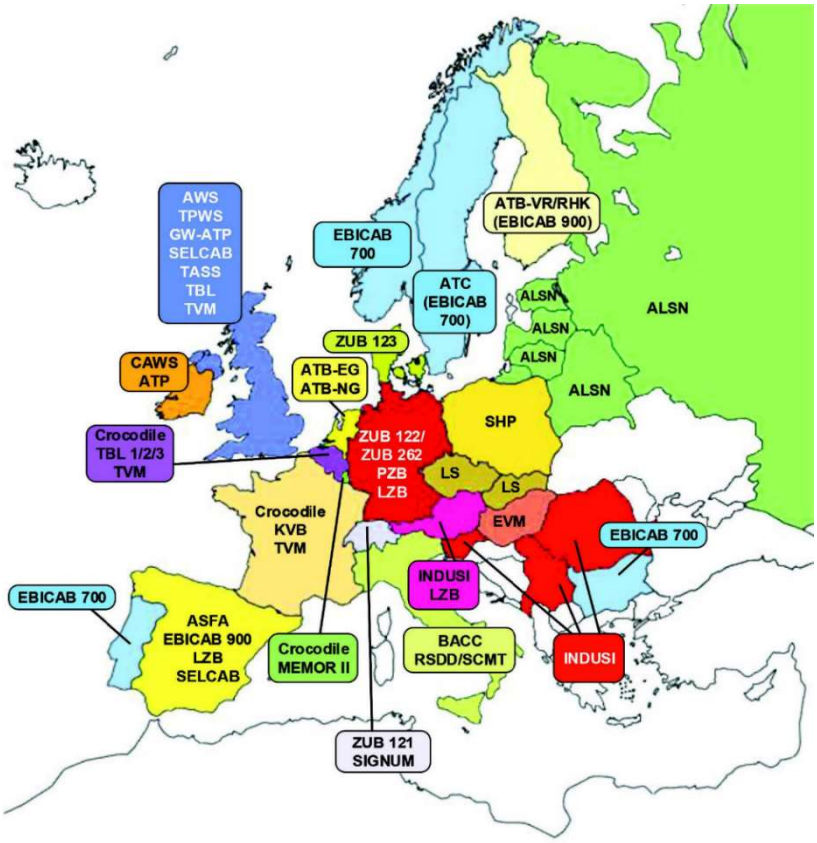


Figure 1.3: Train Control Systems in Europe. (Trackopedia, s.d.)

1.2 Motivation

Railway safety is a primary concern worldwide, especially in Europe, where the density of railway traffic is high and interoperability between different national systems is essential. ATP systems play a crucial role in preventing accidents, ensuring that trains operate within established safety limits. However, the diversity of ATP systems used in different European countries presents significant challenges for interoperability and uniformity of railway operations.

This thesis aims to address the various ATP systems implemented in Europe, analyse some of their characteristics, as well as their advantages and limitations. It also intends to test a small part of the requirements of an ATP system using formal verification with Model Checking. Formal verification is a technique with added advantages, allowing to ensure that a system strictly meets its requirements through an exhaustive analysis of all possible states of the system.

The motivation for this research lies not only in the need to ensure the continued safety and efficiency of European railway systems but also stems from my professional experience as a software tester in the railway sector. It further aims to promote the use of formal requirement verification through Model Checking techniques. These techniques allow for the identification of potential system failures at an early stage of a project, which might otherwise only be detected during the testing phase, resulting in associated costs. Furthermore, this approach will contribute to the reduction of railway accidents and the creation of a safer and more reliable transportation environment for passengers and goods.

It is expected that, through this thesis, a comprehensive analysis of ATP systems in Europe will be provided and the effectiveness of formal verification as a tool for the continuous improvement of railway safety will be demonstrated.

1.3 Problem

1.3.1 Problem Context

Model Checking tools are computer-based systems that analyze the requirements of a system in real-time, with the aim of ensuring they meet safety and performance criteria. Requirement verification through Model Checking allows for the detection of undesirable patterns that can result in safety and requirement failures, providing an additional guarantee of safety that might otherwise be overlooked.

1.3.2 Problem Characterization

The use of Model Checking is crucial in requirement verification because, in complex systems, manual analysis may be insufficient to ensure compliance with safety and performance criteria. In this sense, these tools detect undesirable behaviors or failures in the implementation of system requirements at an early stage of the project. Thus, it not only ensures the safety and reliability of the system but also allows for cost reduction in project implementation, as failures detected at a later stage incur additional costs. This approach is essential to ensure the robustness and reliability of critical systems, where safety is a top priority.

1.4 Objectives

The main objective of this project is to investigate and analyze speed control systems, with an emphasis on their importance, operation, and applications, particularly in the railway sector. The goal is to explore formal methods to model and verify these systems, ensuring their safety, reliability, and efficiency. This thesis will apply model checking techniques to ensure that speed control systems meet formal requirements and specifications. The study's architecture will highlight the use of formal verification tools and techniques to ensure rigorous system validation.

The planned activities include:

- **Objective 1:** Analyze and describe in detail the existing speed control systems, with a main focus on their importance and impact on the safety and efficiency of railway transport.
- **Objective 2:** Conduct a literature review on model checking and its application in speed control systems.
- **Objective 3:** Collect and analyze the requirements of a speed control system, identifying needs and specifications for effective implementation.
- **Objective 4:** Model a small part of a speed control system using formal methods with temporal logics such as LTL and CTL.
- **Objective 5:** Research the most suitable model checking tools for the case study.
- **Objective 6**: Apply model checking tools to identify discrepancies or potential risks.
- **Objective 7:** Evaluate the efficiency of model checking in the context of speed control systems, discussing the strengths and weaknesses of the approach.
- **Objective 8:** Present the study results, providing a detailed analysis of the verification process and the overall system performance.
- **Objective 9:** Propose improvements and optimizations based on the verification results, ensuring compliance with safety and performance standards.
- **Objective 10:** Suggest future research directions and possible improvements based on the study findings, including recommendations for further development and testing.

These objectives aim to provide a comprehensive exploration of speed control systems and their formal verification, ultimately contributing to the development of safer and more efficient transportation systems through the use of formal methods and model checking techniques. To obtain a more detailed perspective on the achievement of these objectives, the thesis will be based on the Gantt chart (Figure 1.4).

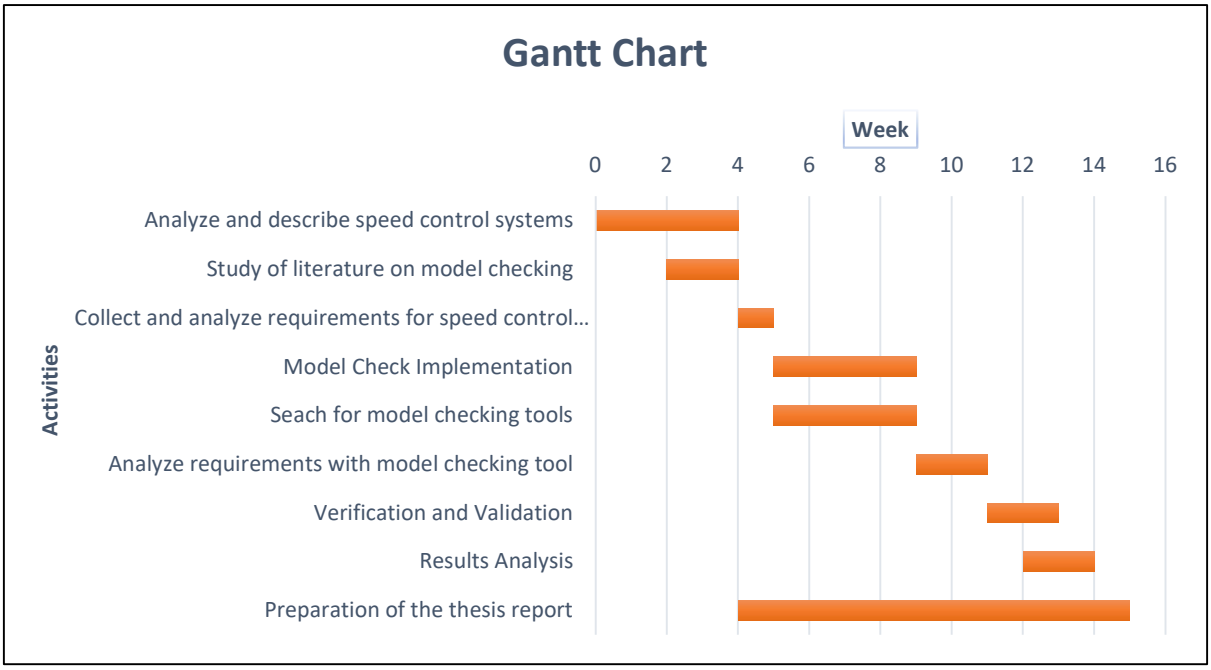


Figure 1.4: Gant Chart.

1.5 Used Methodologies

This subchapter aims to outline the methodologies that will be used to achieve the proposed objectives.

Firstly, an explanation of the existing ATP systems in Europe and their importance in railway transport will be provided. This analysis will include a discussion on the value of these systems in preventing railway accidents and ensuring passenger safety.

Next, a literature review on Model Checking and existing tools will be conducted. This review will involve analyzing relevant articles and documents to understand the state of the art and best practices used in these areas.

The modeling of a small part of an ATP system will be the next step. This modeling will involve developing a representative model of a part of the system, using modeling tools to create an accurate and functional representation.

Requirement verification will be carried out in several stages. Firstly, the requirements of the ATP system will be gathered. Then, the requirements will be classified into clear patterns and specifications. Subsequently, the requirements will be mapped to formal languages such as LTL and CTL. Finally, verification will be performed using a tool (yet to be defined) to ensure that the ATP system model meets the specified requirements.

Finally, a discussion of the results obtained from the formal verification will be conducted. This discussion will include an analysis of the ATP system's efficiency and the improvements identified through verification with model checking tools. Conclusions will be drawn about the system's robustness and reliability, as well as recommendations for future work.

1.6 Structure

This report is structured to address the use of Model Checking in the formal verification of critical systems, such as railway control systems. Below is a summary of its structure, highlighting the main topics covered in each chapter:

- **Introduction:** The first chapter provides an overview of the report's topic, explaining the motivation for using formal verification techniques, such as model checking, in critical systems. The main research objectives and the problems that the use of these techniques aims to solve are discussed. This chapter also introduces the importance of safety in railway systems.

- **State of the Art:** This chapter provides a detailed introduction to railway signaling systems, as well as their history. It also covers the history of ATP systems and their operation. The concept of model checking, including its fundamentals, advantages, disadvantages, and general application in system verification, is another topic addressed in this chapter. It also discusses the use of temporal logics and the phases of the verification process. Additionally, a study on the advantages and limitations of using model checking, especially in critical systems, is provided.

  - Historical Evolution of Signaling Systems: Covers the history of railway signaling, from the first signals to the emergence of ATP systems.

  - ATP Systems in Railway: Brief definition of ATP systems and their historical evolution. It also covers the types of existing ATP systems and some application examples. The integration of ATP systems with new technologies is also mentioned.

  - ERTMS/ETCS: Discusses the ERTMS/ETCS system, focusing on its main role and why it emerged.

  - Model Checking: Delves into the advantages and disadvantages of using model checking in critical systems. Topics such as state explosion, the need for abstraction in models, and limitations related to

the applicability of the method in systems with infinite states are addressed. The importance of model accuracy and the need for adequate tools to ensure efficient verification are also discussed.

- Phases of Model Checking Process: Explains the phases of the model checking process.

- Verification Tools: Explores the most commonly used model checking tools in formal verification, including their functionalities, supported temporal logics, and application domains.

- Model Checking vs Testing: Compares formal and practical verification approaches, highlighting how model checking exhaustively analyzes all possible system behaviors, while testing focuses on specific scenarios with real inputs—both being complementary in software development.

- Selected Model Checking Tools: It addresses the choice of the tools UPPAAL and NuSMV to verify the requirements of an ATP system.

- Formal Specification for Model Checking: This section addresses the formal specification of systems, explaining the importance of formal languages and mathematical techniques in the process of modeling and verifying properties. It details the differences between LTL and CTL logics, discussing their advantages and limitations in railway systems.

- Some Application of Model Checking in Railway Systems: This chapter focuses on the practical application of model checking in railway systems. Three relevant case studies are presented that exemplify how formal verification is used in railway systems.

- **Model checking Tools:** This chapter introduces the tools UPPAAL and NuSMV, focusing on their application in the modeling and verification of systems. It explores the fundamentals of these tools and their general use in system verification.

- NuSMV: In this subsection, the NuSMV tool is presented, highlighting its functionalities, architecture, as well as the specifications used by the tool.

- UPPAAL: In this subsection, the UPPAAL tool is presented, with an emphasis on its specifications and its origins. The concept of time used in this tool for modeling is also addressed.

- **Automatic Train Protection:** In this chapter, an example of ATP system architecture is demonstrated, using the EBICAB 700 as a case study.

  - System Architecture: In this subsection, the architecture of the EBICAB 700 system is demonstrated.

  - Requirements: In this subsection, the requirements that were subject to modeling are demonstrated.

- **Models, Verification, and Results:** Formal models were developed in UPPAAL and NuSMV to verify critical requirements of the railway system, ensuring safety and the absence of deadlocks. The analysis confirmed full compliance and highlighted the comparative efficiency of both verification tools.

  - Model of ATP System in NuSMV: In this subsection, the ATP system model was formalized using NuSMV, with a precise definition of the system states and transitions, as well as the conditions for emergency brake activation.

  - Model of ATP System in UPPAAL: In this subsection, the ATP system model was developed in UPPAAL using timed automata to represent states and transitions, including real-time braking speed calculations.

2 State of the Art

The development and application of formal verification techniques have evolved substantially in recent decades, especially in the context of critical systems, such as railway control systems. Formal verification has become an essential tool to ensure the safety, reliability, and accuracy of these systems, providing a rigorous way to detect errors and design flaws in the early stages of development.

This chapter addresses the state of the art in railway signalling systems, as well as in the field of model checking and its applications in critical systems. Initially, an analysis of the main concepts and methods of formal verification will be conducted, focusing on temporal logics such as LTL and CTL, which are widely used in model checking tools.

Additionally, the chapter covers the verification tools for railway systems, presenting some examples of how model checking has been successfully applied to increase safety and reduce design errors in these systems.

With this, we aim to provide a comprehensive overview of current solutions, particularly regarding the application of model checking in railway control systems.

2.1 Historical Evolution of Signaling Systems

2.1.1 Signalling of the First Trains

Railway signalling emerged with the first use of George Stephenson's steam locomotive. At that time, trains were controlled solely by the driver's vision, with the braking process initiated only upon sighting another train.

Previously, around 1806, wagons were pulled by animal traction in mines and quarries. Some records from that time indicate that manual and arm signals were used to direct the movement of these early trains (Ganguly, 2001).

Over the years, train speeds increased, which resulted in longer braking distances, making it unsafe to stop the train after spotting an obstacle on the track. New rules were introduced to ensure a safe distance between moving trains through signals for the driver along the track. These problems were largely caused by the lack of adhesion between the train wheels and the track, as well as the existence of inefficient brakes.

Initially, trains were operated based on a time interval system, mostly of ten minutes, where a train should leave the station at an exact minute, immediately after the previous

train traveling in the same direction. A train could only travel at maximum speed ten minutes after the departure of the previous train (Site, 2023). However, this method had some limitations, such as when delays occurred with other trains, which triggered a chain of delays (Shedd, et al., 2024).

To increase railway safety and efficiency, the first signaling systems were introduced, which divided the track into blocks and used manual signals. The Railway policeman, as they were called, provided indications to the drivers through hand gestures and were positioned at the beginning of each block.

A policeman, standing in front of the train with an outstretched arm, indicated to the driver whether the track ahead was clear or not. After the train passed and entered the block, the policeman assumed a resting position. If another train approached, the policeman continued to signal an obstruction until a time interval, usually seven-eight minutes, had passed. Only then did he allow the next train to proceed, but with caution. This method of signaling and separating trains helped maintain safety on the railway tracks.

The Railway Policeman used yellow, red and green flags, as shown in figure 2.1, to indicate to the driver how to proceed. The red flag was used in the first five minutes immediately after the train's departure. In the event of a train arriving with a five-minute delay, a yellow flag was shown to the driver, while the green flag was shown after the ten minutes had elapsed (Site, 2023).

Figure 2.1: Flags Used for Indicating to the Driver. (Ferroviário, 2021)

In the event of a train breakdown and the need to stop it outside the field of vision of the two-railway policeman, a crew member had to walk along the track as far as possible to signal the next train to stop.

At the end of the 1830s, the railway experienced a new advancement with the introduction of the first optical signals. This era marked the replacement of manual signals with fixed signals, leading to the emergence of the semi-automatic block, resulting in a significant improvement in communication and safety in railway operations (Clark, 2024).

To improve railway efficiency and safety, mechanical signals were introduced in the early 20th century. The telegraph, and later the telephone, facilitated communication between line managers, allowing messages to be sent, initially by bell rings and later by calls, to indicate the line's occupancy status. (Palumbo, 2013).

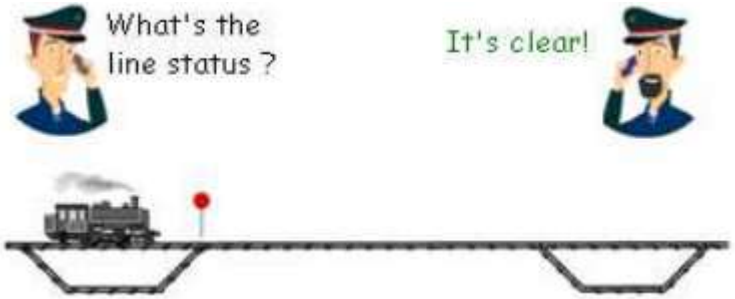


Figure 2.2: Phone Block System. (Palumbo, 2013)

2.1.2 Fixed Signals

At the end of the 1830s, manual signals began to be replaced by fixed signals, mostly consisting of flags or movable discs and light signals, installed on tall posts, operated by a railway policeman. At night, to avoid confusion with external lights, white light signals were used to indicate the "clear" state.

In 1841, Charles Hutton Gregory designed a semaphore signal (Figure 2.3) for the London & Croydon Railway, which would become the first example of a "railway signal" currently in operation. In 1843, Gregory developed a system of levers and rods that allowed the operation of multiple signals from a single location, to prevent derailments or collisions, which would become the basis of the "interlocking" system. However, true "interlocking," where one lever movement must be completed to allow the movement of another lever, only emerged in the 1860s (Clark, 2024).

Figure 2.3: Model of Signal Implemented by Charles Hutton Gregory. (Group, s.d.)

2.1.3 Telegraph

The need to ensure the distance between trains by time intervals remained despite the development of signals. Therefore, it would be useful for railway policeman to communicate with each other in a simple and reliable manner, ensuring the circulation of trains by time interval. This growing need led to the development of the Cooke & Wheatstone electric telegraph, first demonstrated in 1837.

These instruments (Figure 2.4) used a pointer or needle, which moved to the left or right to allow the exchange of messages, known as telegraphic code. The words in these messages were spelled out letter by letter, facilitating communication between the railway policeman, who indicated when a train entered or left the block.

This form of communication led to the implementation of a system known as the "Absolute Block," which consisted of ensuring that every train entering a block must be observed leaving it before another train is allowed to enter the same block (Clark, 2024).

Figure 2.4: Telegraphic Block Instrument. (Group, s.d.)

2.1.4 Blocks

Trains cannot collide with each other if they are not permitted to occupy the same section of track at the same time (Wikipedia, 2024).

To ensure a safe distance between trains and avoid collisions, railways are divided into sections or "blocks". Each block has a signal at its entrance that can be red, indicating the track is occupied and the driver must stop the train, or green, indicating the track is clear and the driver can proceed.

This system, implemented by Dr. William Robinson in 1972, revolutionized railway signaling with the closed track circuit. With two mechanical sensors installed, one at each end of the block, which activated the respective signal. (Ganguly, 2001).

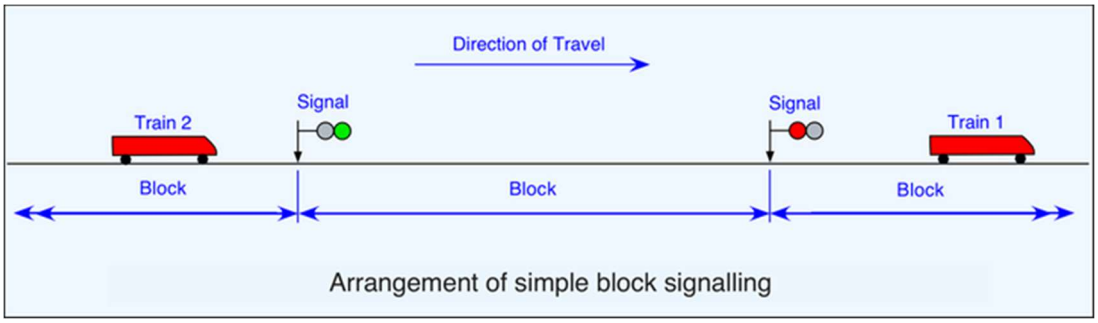


Figure 2.5: Block Signalling Scheme. (Site, 2023)

This system ensures the crucial signaling rule, where only one train is allowed in a block at a time, as we can see in figure 2.5 above. The block occupied by “train 1” is protected by the red signal at the block entrance, indicating the block is occupied. The block immediately before is clear, so the installed signal shows green, allowing “train 2” to enter that block (Site, 2023).

Currently, railway lines are equipped with automatic blocks, whose blocks have a length never less than the braking distance of the fastest train running on that track.

To detect the passage of the train in a given block, a relay is used, which is energized as soon as a train passes over the track, which in turn will activate the occupied or red signal of that block.

In the case of more recent lines, there are devices called Axle Counters (Figure 2.6) installed at the beginning and end of the track, responsible for detecting all the train axles entering the block and comparing the total axles detected at the beginning of the block with those detected at the end. If the number of axles detected at the end of the block matches the number of axles in the beginning of the block, the clear track state will be activated (Palumbo, 2013).

Figure 2.6: Axle counter installation on the track. (Argenia, s.d.)

As we can see in figure 2.7, block “TS1” will be considered clear as soon as the number of axles counted at “Ax2”, at the block exit, matches the number at the block entrance, that is, at “Ax1” (Palumbo, 2013).

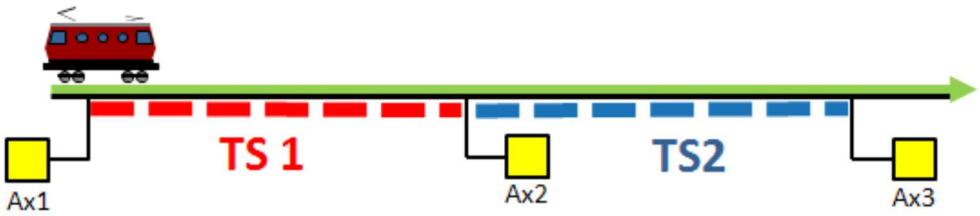


Figure 2.7: Axle Counters for train detection. (Palumbo, 2013)

2.1.5 Interlocking

The previously mentioned system proved reliable for cases where all trains travel on the same line. But for bidirectional traffic, safety systems were needed to ensure that there would be no possibility of head-on collisions. This led to the introduction of a new safety measure in the mid-19th century, known as Interlocking.

The purpose of this system was to prevent, in places with line intersections, a configured train route and its clear signal, where the train can proceed, from conflicting with another configured train route also with the clear signal activated, thus avoiding head-on collisions. This was achieved through mechanical systems, consisting of levers and rods (Figure 2.8), capable of activating the signals in the signal cabin.

Figure 2.8: Principles of railway interlocking. (railwaysignalling.eu, 2015)

With the development of signaling technologies, mechanical systems began to be replaced by small levers or even buttons, with panels being replaced by electromagnetic relays, used in series to ensure the correct configuration of routes. Currently, most of these systems have been computerized.

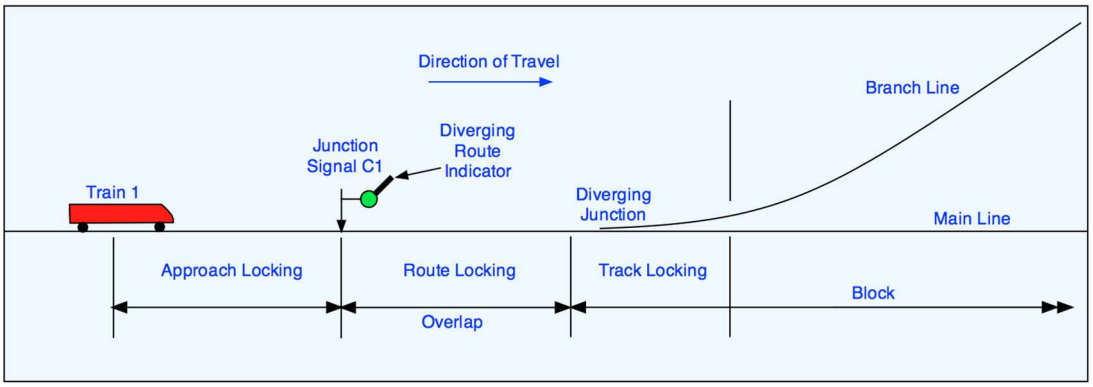


Figure 2.9: Interlocking. (Website, 2023)

2.2 ATP Systems in Railway

We can see that the previous analysis of signaling focused only on warnings and restrictions to the driver, without considering a factor that should not be overlooked, the braking distance. With the evolution of locomotives, the maximum speed they can reach has also increased significantly, which in turn translates into an increase in the distance required to make a safe stop. For example, a train traveling at a speed of 160 km/h may need approximately 1.5 km to make a safe stop. (Site, 2023).

Train protection systems aim to mitigate or even eliminate the possibility of driver error, in order to prevent accidents that can be caused by the driver's non-compliance with a signaling indication.

Initially, these systems only alerted the driver that they were approaching a signal with restrictions, requiring them to acknowledge the warning. In the event that the driver did not acknowledge the warning, the system would activate the train's brakes. Later, largely due to imposed speed restrictions and limitations, these systems were improved to meet the speed limits imposed by the restrictions. For this, it was necessary to install information transmission systems, such as permanent magnets, electromagnets, and coded circuits.

Currently, these systems are fully automatic and are therefore designated as ATP, where they impose speed limits and train movement actions on all types of restrictive signals, with or without line signals present. With this system, the driving is fully manual,

with the system only having the braking capability according to the imposed restriction (Connor & Schmid, 2023).

2.2.1 Definition and Purpose of ATP Systems

Railway signaling systems in many countries rely on the driver's reaction to indications displayed by light signals, such as traffic lights, adjusting the train's speed according to the instructions. Over the years of using railway signaling, driver's failure to respond to commands indicated by wayside systems has, in some cases, led to several accidents, some of which resulted in a large number of fatalities. To address the ongoing need to reduce the risks generated by driver failure, various types of driver warning devices have been developed, capable of continuously monitoring the train's speed and imposing speed limits. These systems are known as ATP (Connor & Schmid, 2023).

ATP systems are a subsystem of the Automatic Train Control (ATC) system and are capable of supervising train speeds in relation to a permitted speed limit, which is automatically determined by the onboard equipment based on information received from the ground signaling subsystem. The onboard control system, installed in the train cabin, aims to ensure compliance with speed limits by calculating the so-called "braking curves" to allow the train to decelerate and stop before any stop signal or emergency condition (Flammini, 2013).

2.2.2 Historical Evolution of ATP Systems

In the early 1900s, the first automatic train control system, known as Automatic Train Stop (ATS), was introduced with the aim of preventing trains from passing signals indicating that the track was occupied by another train. The ATS used electromechanical devices that communicated with the train's braking system, automatically applying the brakes if the driver did not stop at the signal.

Later, in the 1930s, this system was improved and became known as ATC. The ATC used radio communication between the train and the control center to monitor the train's movement and regulate its speed. This system automatically applied the brakes if the train exceeded the speed limit or if the driver did not stop the train at a red signal (Rail, 2024).

ATC consists of all vital and non-vital functions responsible for ensuring the safe operation of trains. Currently, this system is composed of three main subsystems (Software, 2024):

- Automatic Train Supervision (ATS).
- Automatic Train Operation (ATO).

- Automatic Train Protection (ATP).

In the 1960s, the Integrated Electronic Control System (IECS) was introduced, representing a significant advancement in the evolution of ATC. The IECS used computers and computer networks to control trains, allowing for more precise control of speed and movement, as well as communication between trains and the control center, which significantly improved safety and efficiency.

In the late 1960s, fully automatic ATP systems first appeared in subways and are now installed in similar systems worldwide. In 1964, this system was also implemented on the Japanese high-speed Shinkansen route (Connor & Schmid, 2023).

The introduction of the ETCS in the 1990s marked a significant advancement in railway signaling. This system uses digital radio communication between the train and the control center, providing more precise regulation of train speed and movement (Rail, 2024).

2.2.3 Automatic Train Supervision - ATS

Like ATP, ATS is a subsystem of the ATC system. It is a non-vital system responsible for managing all line traffic. It is capable of monitoring and controlling all existing traffic to ensure safe operation, such as route definition and maintaining a train within the allowed time. It is also capable of managing passenger information (Software, 2024).

Figure 2.10: ATS System. (Railworks, 2024)

2.2.4 Automatic Train Operation - ATO

Similarly, this non-vital system is a subsystem of the ATC system and is mainly located onboard, providing functions for driverless operation. It is responsible for ensuring the correct stopping of the train at stations or in front of a stop signal, as well as maintaining the speed profile under ATP control. It is also capable of opening and closing doors on the correct side of the train once it determines its position, ensuring that no passenger exits the train until it is fully immobilized. It must also report the train's status to the control center (Software, 2024).

2.2.5 Types of ATP Systems

The ATP system has two distinct implementations: intermittent and continuous. Intermittent systems can use inductive or radiofrequency beacons, known as "balises" (Figure 2.11). Continuous systems, through various means such as electrical inductive couplings, have permanently active monitoring and data transmission.

Figure 2.11: Passive Balise. (Pandrol, 2024)

2.2.6 Examples of ATP System Implementations

In this section, we will discuss some examples of ATP systems implemented in Europe.

Automatische Trein Beïnvloeding (ATB NG)

The ATB NG system, Automatic Train Control, is an ATP system implemented in the Netherlands in the mid-1990s. It consists of beacons installed on the track and onboard equipment. The transmission is done between the active beacon and an antenna installed on the train.

In this system, the driver must input relevant data into the system, such as maximum speed and braking characteristics. The train is equipped with a screen that displays the maximum permitted speed on the line, the target speed, the distance to the destination, and the braking curve. The driver also receives audible and visual warnings of overspeed to mitigate possible errors that could cause accidents, activating the emergency brake whenever a restriction is violated or if the driver does not respond to the warnings (Connor & Schmid, 2023).

Ebicab

The Ebicab is an ATP system currently in operation in Sweden, Norway, Portugal, and Bulgaria. The system installed in Sweden and Norway uses the same software, allowing cross-border operation without the need to change trains or even drivers. In Portugal and Bulgaria, different software versions are used.

The Ebicab has two versions, the Ebicab 700 and Ebicab 900, with identical safety functions. In this system, signal encoders are installed to send data to passive beacons located on the track, which in turn transmit to the vehicle via an antenna installed on it, energizing the beacon as it passes by.

In the Ebicab system, some relevant data for the system's operation must be entered by the driver. It also has screens that display the maximum line speed, the target speed, among other information. In the event of overspeed, the time for service brake intervention is displayed, as well as three audible warnings.

Service braking starts as soon as the train's speed exceeds 10 km/h and 5 km/h in the case of the Ebicab 900, relative to the maximum permitted speed, with the brakes being released as soon as the speed is equal to the permitted speed. In the event that the emergency brake is activated due to non-compliance with the maximum permitted speed, the brake will only be released once the train comes to a complete stop (Connor & Schmid, 2023).

KVB (Contrôle de Vitesse par Balises)

The KVB, Beacon Speed Control, is an ATP system used in France, technically very similar to the Ebicab and installed throughout the railway network.

Similarly to the Ebicab, in this system, data is transmitted to the vehicle inductively between beacons installed on the track and the antenna installed on the train whenever it passes over a beacon. Since data transmission in this system is limited, additional beacons are installed.

If a fixed unit train is used, its data is automatically entered into the system. Otherwise, the driver must input all the necessary information for the system to function correctly.

Like the EBICAB 700 and 900, if the maximum speed is exceeded, audible warnings will be issued, followed by the application of the emergency brake, which can be released once the train comes to a complete stop (Connor & Schmid, 2023).

TBL (Transmissie Baken Locomotief) 2

The TBL 2 system is installed on all lines in Belgium that allow trains to travel at speeds exceeding 160 km/h. Similar to the ATP system installed in the United Kingdom, it features powered beacons installed on the track in the form of steel loops. Since this system is direction-sensitive, the beacons are installed slightly off-center in relation to the rails.

On the onboard system, the driver can view the maximum permitted speed, the target speed, the target distance, as well as the train's instantaneous speed. The driver will receive warnings whenever restrictions are violated, with the brakes being activated if the warnings are not acknowledged by the driver (Connor & Schmid, 2023).

Linienförmige Zugbeeinflussung (LZB)

The LZB, from the English Linear Train Control, is a continuous ATP system installed on all lines in Germany with speed limits exceeding 160 km/h. This system can also be found on some lines in Austria and on the Spanish high-speed line between Madrid and Seville. In this system, data transmission between the track and the train is carried out through inductive cable loops and antennas installed on the train.

As with the systems described above, the driver must input the train's length, maximum speed, and braking characteristics. Onboard, this system features a two-pointer speedometer that displays the maximum permitted speed and the actual speed. On a separate display, it is possible to view the mode of operation, the status of data transmission, as well as the target speed and the distance to the destination. It is also

capable of monitoring the permitted speed on a given section of track, the train's maximum speed, and its direction, activating the emergency brake when speed limits are not adhered to (Connor & Schmid, 2023).

2.2.7 Integration of Traditional ATP Systems with New Technologies

In addition to operating autonomously, ATP systems, specifically the Convel system, will be integrated with other railway control and signaling systems such as ETCS, forming an integrated safety ecosystem called Specific Transmission Module (STM-PT). This system is being developed by Critical Software, Medway, Alpha Trains, and Thales, and will allow trains equipped with ETCS to operate on tracks equipped with ATP systems (T&N, 2023).

Given the various types of ATP systems existing in different European countries and the difficulties encountered in cross-border train operations, as each ATP system has its specificities, the ETCS emerges to solve these complications and contribute to interoperability, thus allowing standardization in protection systems.

Since a large part of the railway tracks still use the old ATP system and the new trains are already equipped with the current ETCS system, there is a need to create a system capable of bridging the ATP systems and the current ETCS.

Thus, the STM emerges, serving as an intermediate communication system between ATP and ETCS systems, allowing trains equipped with this current system to operate on tracks equipped with ATP.

Its main function is to read the data sent by the beacons of the old ATP system installed on the track, interpret this collected information so that it can be recognized by the new ETCS system.

This new system presents itself as a solution for any ETCS system supplier, allowing the correct migration of the system and greater signaling openness in Portugal. This system can also be removed from the train once the railway lines have the ETCS system installed.

Given this, we can see that this project represents a technological advancement in Portugal, as it allows for an increase in service quality as well as technical support.

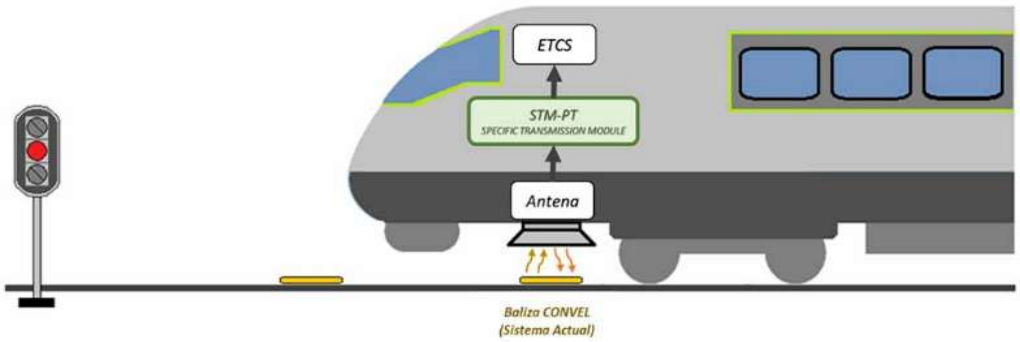


Figure 2.12: Integration of CONVEL with ETCS. (T&N, 2023)

2.3 ERTMS / ETCS

Before the ERTMS was developed, most European countries had their own ATP system, which were largely incompatible with each other, making cross-border train operations impossible. To achieve this, trains would need to be equipped with multiple ATP systems. With the increasing demand for this mode of transport, the need to implement a single ATP system across Europe grew, to replace the old ATP systems, promoting interoperability and efficiency in this transport (Commission, s.d.).

The ERTMS is composed of the ETCS, an onboard signaling system that includes the ATP system, the GSM-R, a system that allows communication between trains, railway infrastructure, and command centres. It also includes a set of operational rules (Railways, s.d.).

2.3.1 ETCS

The ETCS is divided into different functional levels, with these levels defined according to the railway infrastructure equipment and how information is transmitted to the train. The ETCS specification has two substantially different ATP operation levels, providing full speed supervision and varying amounts of information to the driver during the journey, and can be summarized as follows:

ETCS LEVEL 1

At this level, the system is capable of constantly supervising the maximum permitted speed and calculating the braking curve to the authorized location for the train to proceed, known as the limit of movement authority. Similar to ATP, communication between the train and the infrastructure is done through beacons, with the only difference being their designation, as in ETCS they are called Eurobalises.

This level does not dispense with trackside signals, except when data is transmitted from the infrastructure to the train intermittently via Euroloop or Radio Infusion Unit, allowing the train to receive data about its future movement (Commission, s.d.).

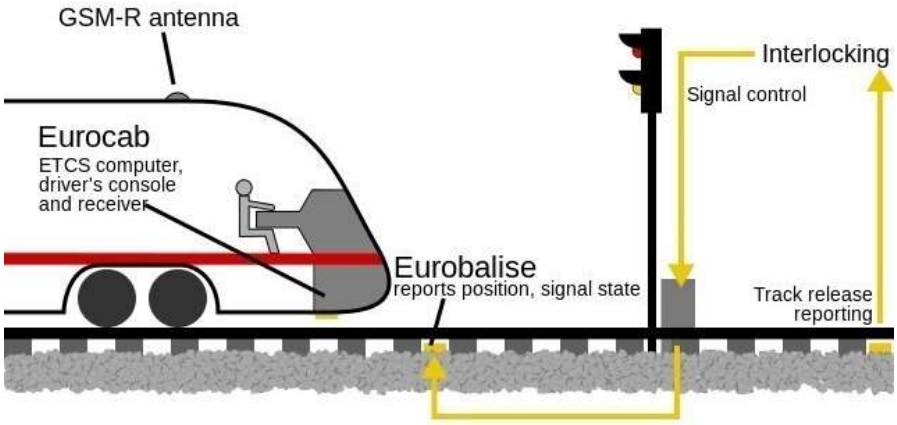


Figure 2.13: ETCS Level 1. (Commission, s.d.)

ETCS LEVEL 2

At this level, continuous supervision of the train's movement and permanent communication between the train and the railway infrastructure via GSM-R antenna is available. More recently, this level has integrated the concepts of ETCS Levels 2 and 3.

Trackside signals are optional, but train detection and train integrity supervision remain the responsibility of the track.

The onboard equipment is responsible for continuously monitoring the data from the beacons and the RBC (Radio Block Centre) to calculate its position and the maximum permitted speed. In this way, it provides the driver with relevant data so that the train can reach the ideal speed and maintain a safe braking distance (Commission, s.d.).

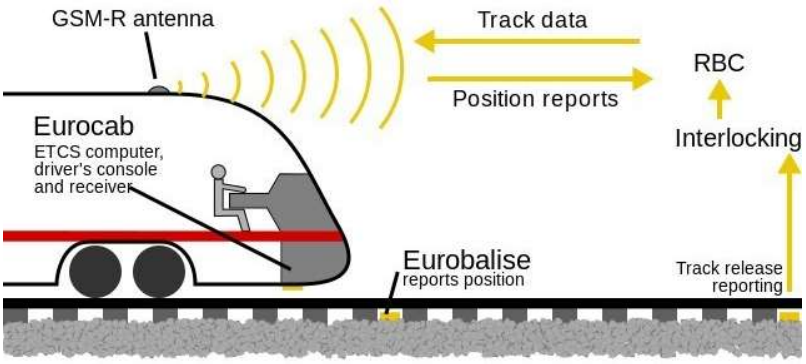


Figure 2.14: ETCS Level 2. (Commission, s.d.)

2.4 Model Checking

2.4.1 About Model Checking

Model Checking is a formal technique used to ensure that concurrent and distributed systems function correctly. It emerged as a response to the need to solve verification problems of concurrent programs, which are notoriously difficult to test due to the difficulty of reproducing concurrency errors. The rudimentary way of verifying such programs involved manually constructing proofs using formal logics, such as Floyd-Hoare logic. However, the scalability of these manual proofs was a significant concern.

In the 1970s, researchers like Owicki and Gries developed formal methodologies to analyze conditional critical regions, while Pnueli and others proposed the use of Temporal Logic to specify concurrent programs. Temporal Logic proved to be ideal for expressing concepts such as mutual exclusion, absence of deadlock, and absence of starvation. Absence of starvation is the condition where, in concurrent systems, all threads or processes are guaranteed access to the necessary resources within the required time.

Model Checking combines state exploration with Temporal Logic efficiently, allowing for the automatic verification of properties of concurrent systems. This approach has proven effective in solving non-trivial problems and is widely used in the verification of communication protocols, embedded systems, and other critical systems (Clarke, 2007).

2.4.2 What is Model Checking

Model checking is a technique used to ensure that a finite state model of a system meets a predefined specification, which can be a hardware or software system. It analyzes whether the model meets a set of properties, which can be safety or quality guarantees, i.e., whether the system performs as expected without getting stuck in a state. This technique uses automatic tools to verify if concurrent systems function correctly according to the specifications.

The main aspects of model checking include the creation of a model, which is a mathematical representation of the system's behavior, and can be represented in the form of a graph or state machine. As for verification, this is the process of ensuring that the model meets the established properties, using tools that automate this process.

This method has various applications, such as error identification, implementation testing, and system correctness verification. It is currently widely used in artificial intelligence to verify the correctness of proposed solutions. Despite all the advantages this method presents, it has some limitations, such as the state space explosion problem and the challenges associated with temporal logic (II, s.d.).

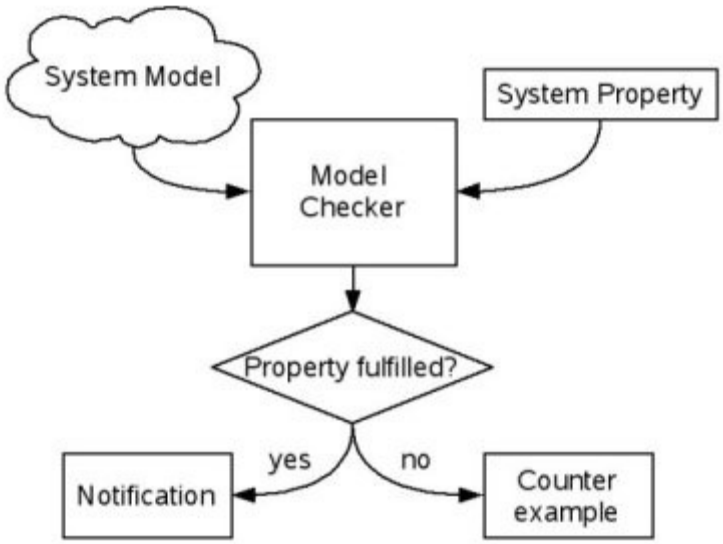


Figure 2.15: Model checking flow. (Wien, s.d.)

2.4.3 Advantages in use Model Checking

Model checking has several advantages, including the fact that the technician verifying the models does not need to create correctness proofs.

Another advantage is its performance, compared to other methods, which can take months of work.

Diagnostic counterexamples are another advantage. In the event that the specification is not met, this method will create a vast sequence of counterexample execution, showing the reason why the specification is not valid, which is very useful in debugging complex systems.

Since it is not necessary for the program to be fully specified, it is possible to analyze its properties during the design phase, meaning it does not present any problems with partial specifications.

Another advantage we can find is the ease with which temporal logics can express properties relevant to the analysis of concurrent systems (Clarke, 2007).

2.4.4 Disadvantages of Model Checking

This method has some disadvantages, which we will describe below:

Suitable for applications: Due to the volume of data that can grow infinitely, this method is more suitable for control-intensive applications and not as appropriate for data-intensive applications.

Decision issues: The application of this method in infinite state systems in abstract data types can become a problem regarding the applicability of this method, as it is limited by decision issues, not being computable efficiently (Clarke, 2007).

Model Checking and Real Systems: Since this method verifies system models and not real systems, the results largely depend on the accuracy of the model used. It is crucial to ensure that the model is as identical as possible to the real system so that the results are valid.

System requirements already specified: Since it only verifies the declared specifications and this method is limited to the requirements, verification can only ensure that the system meets the specified requirements, so it does not guarantee that all possible system behaviors have been considered.

State Explosion: State explosion, which can occur whenever the number of states needed to accurately model the system exceeds the amount of memory available on the machine where the verification is being performed, is one of the problems of this method.

Experience: The application of this method requires some experience in finding appropriate abstractions to create smaller system models and declare properties in logical formalization, ensuring accurate and efficient verification.

Error-free results: Since the tools used are composed of software, they may contain implementation errors, which in turn can lead to incorrect results.

Generalization Verification: It does not allow the verification of generalizations, that is, it does not allow systems that can have an arbitrary number of components or that are parameterized. However, it can suggest results for arbitrary parameters that can, in turn, be verified using proof assistants, which are formal tools that assist in the implementation and verification of mathematical proofs (Baier & Katoen, 2008).

2.5 Phases of Model-Checking Process

Following are the different phases in model checking process:

Modeling: Modelling a system requires the creation of finite state machines to specify the desired behavior. The first step is to convert the system design into an abstract model accepted by a model checking tool, in order to eliminate irrelevant details. In some cases, this conversion can be automatic; however, it still requires human guidance and assistance (Baier & Katoen, 2008).

Properties definition: It involves the precise and unequivocal specification of the system properties to be verified against the system model. Temporal logic constraints, such as LTL or CTL, are generally used to describe a wide range of behaviors and characteristics of the system, such as functional accuracy, accessibility, integrity, continuity, fairness, and temporal properties. Precision in defining these properties is crucial to ensure rigorous and effective verification.

Run Model Checker: The model checker is used to ensure the validity of the properties defined in the system model. Then, the results should be saved for future analysis.

Results Analysis: In the phase of analyzing the results of model checking, the obtained results are carefully examined to verify if the defined properties are guaranteed by the system model. Whenever a property is satisfied, the next property is verified. When a property is violated, the generated counterexample or error path is analyzed by simulation. After the analysis, the model, system design, or the property itself is adjusted. This process is repeated iteratively until all properties are satisfied (Baier & Katoen, 2008).

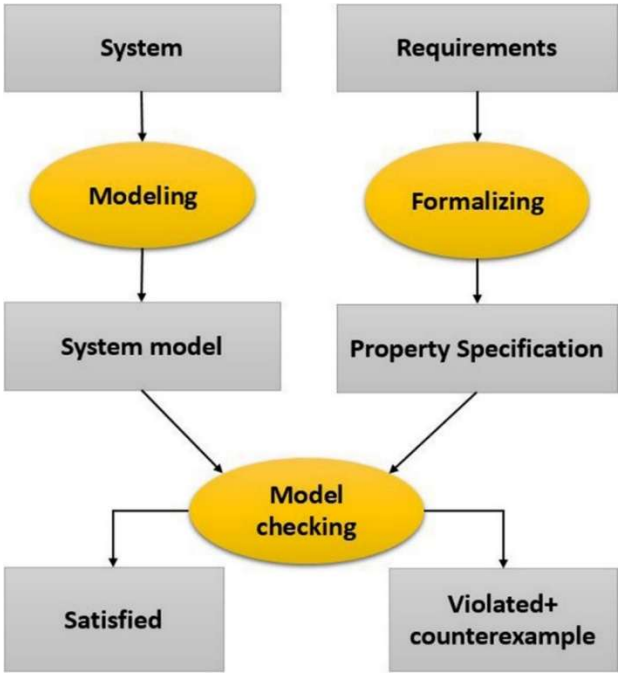


Figure 2.16: Model checking architecture. (Almakhour, et al., 2020)

2.6 Verification Tools

Model Checking tools are essential for the formal verification of complex systems, allowing the detection of faults and the verification of properties before actual implementation. These tools are applied in various domains, such as communication, embedded systems, Software Engineering, hardware, and healthcare. They use different types of temporal logics, such as linear-time logics and branching-time logics, to express system properties as logical formulas that are automatically verified. Each tool may use different types of logic to express properties, depending on the system's characteristics and the application domain. Table 2.1 presents a list of some of the most commonly used tools.

Table 2.1: List of some available Model Checking tools.

Tool	Full Name	Main Features	Supported Temporal Logics	Platform	Applications
SPIN (Holzmann, 1997)	Simple Promela Interpreter	Used for model checking of concurrent or asynchronous processes.	LTL	Windows, Unix	Communication protocols, distributed systems
NuSMV (Cimatti, et al., 2000)	New Symbolic Model Verifier	Open-source symbolic model checker for both LTL and CTL logics.	LTL, CTL, RTCTL, PSL	Windows, Unix, MacOS	Digital Circuits
UPPAAL (UPPAAL, s.d.)	UPPAAL Model Checker	Verification of temporal and real-time systems using timed automata.	TCTL	Windows, Linux	Real-time control systems
PRISM (Department of Computer Science, s.d.)	Probabilistic Symbolic Model Checker	Focused on probabilistic systems and quantitative logics.	PCTL, CSL, LTL, PCTL*	Windows, Linux, MacOS	Systems with uncertainty, communication networks
Cadence SMV (Mir, et al., 2000)	Cadence SMV Model Checker	Commercial version of NuSMV, with advanced features for complex systems.	CTL, LTL	Linux, Unix	Hardware verification, critical systems
DVE (DiVinE) (Basit-Ur-Rahim, et al., 2014)	Distributed Verification Environment	Distributed verification environment integrating multiple tools.	Varies based on integrated tool	Unix	Distributed systems
Simulink Design Verifier (Simulink, s.d.)	Simulink Design Verifier (Prover)	Used to verify models created in Simulink, a data-flow and state-machine simulation	-	Windows, Linux	Embedded systems
FDR (SCIENCE, s.d.)	FDR Model Checker	Used to model asynchronous systems and check CSP refinement.	CSP refinement, temporal	Windows, Unix	Communication protocols, concurrent systems

2.7 Model Checking vs Testing

The difference between model checking and testing primarily lies in the approach and scope of analysis that each technique offers.

Model checking is a formal technique designed to verify software system properties exhaustively. It builds a mathematical model of the system and evaluates all possible executions of the program, using logic and algebra to verify properties such as the absence of deadlocks or conformity with specifications. This technique ensures comprehensive analysis, covering all possible scenarios regardless of specific inputs, but it can be computationally expensive, especially for complex systems (Gunter & Peled, 2005).

On the other hand, testing presents itself as a practical and intuitive approach, more focused on executing the software with a limited set of test cases. It evaluates the actual behavior of the program with different inputs to identify failures. Although it is more flexible and easily applicable in everyday development, testing does not cover all possible system executions, which can leave rare errors undetected.

In summary, while model checking seeks to prove the correct implementation of the system with a mathematical approach, testing is more focused on finding problems in specific and practical scenarios. Both play complementary roles in software development, where we conclude that model checking is essential for critical systems requiring high reliability, and testing is more utilized for quick and practical validations during the development process. Combining these two techniques can offer a more robust and detailed analysis of software behavior (Beyer & Lemberger, 2017).

2.8 Selected Model Checking Tools

During this systematic literature review, tools were identified that can be used for model checking. For this master thesis project, UPPAAL and NuSMV were selected to perform model checking on the requirements of an ATP system.

UPPAAL is a model checking tool particularly suited for modeling, simulation, validation, and verification of real-time systems. It is one of the most widely used, frequently updated, and continuously evolving tools in this domain. On the other hand, NuSMV is a symbolic model checker known for its powerful verification capabilities and extensive support for temporal logics. Together, these tools offer complementary strengths for addressing the challenges of modeling and verifying real-time systems, such as the ATP system.

The use of these tools facilitates the efficient modeling of the system's requirements and ensures that its behavior complies with the intended specifications. By using these tools, this thesis aims to demonstrate the feasibility and efficiency of applying UPPAAL and NuSMV to model and verify the critical requirements of an ATP system.

2.9 Formal Specification for model checking

Formal specification defines the requirements and behavior of software systems using mathematical and logical techniques. While informal specifications rely on natural language descriptions and diagrams, these use precise languages to describe exactly what the system should do, thus conveying a clear description of the intended behavior (Badshah, 2024).

Formal specifications describe how the values of state variables should evolve over time. Temporal logics, such as LTL, are used to analyze these changes, allowing the formulation of atomic, boolean and temporal propositions. However, LTL has some limitations when working with finite sequences of states in complex systems. To overcome these limitations, extensions such as Bounded Linear Temporal Logic (BLTL) have been developed, which allow writing logical properties limited to specific time intervals (Van Lamsweerde, 2000).

2.9.1 LTL

LTL is a form of temporal logic where modalities refer to time. LTL is a formal system capable of modelling the temporal space, allowing the representation of statements qualified in temporal terms. While in classical logic, formulas are restricted to a single fixed point, temporal logic seeks to evaluate statements within a set of possible cases. For example, in classical logic, a statement like "the traffic light is green" can only be "true" or "false." However, in temporal logic, we can consider different moments in time, where in the same example, the state can be true at one exact moment but false a few seconds after the traffic light changes to red (Arzaghi, 2021).

In the 1970s, Pnueli applied temporal logics to the formal verification of complex computer systems. These include operators such as:

- ◇ “eventually”.
- □ “always”.
- p ∈ P.
- ϕ & ψ “conjunction”.
- ϕ | ψ “disjunction”.
- !ϕ “negation”.
- ϕ → ψ “implication”.
- ϕ ↔ ψ “equivalence”.
- Gϕ “globally, in all states”.
- Xϕ “in the next state”.
- Fϕ “finally/in the future”.
- ϕ U ψ “ϕ until ψ”. (Pereira, et al., 2024).

Temporal logics can be defined as linear or branching, with LTL adopting a linear time view, with a single, sequential future (Figure 2.17), where each state has only one successor. On the other hand, CTL is based on a branching time view, which allows for multiple possible futures from the same point in time.

One of the advantages of LTL is the ability to express fairness assumptions without the need for additional tools. However, LTL has limitations, such as the inability to specify the exact timing of events, being restricted only to their relative order (Baier & Katoen, 2008).

Figure 2.17: Linear Temporal Logic. (SlideServe, 2014)

Example:

As an example of application, we have the traffic light, which has states such as "Green," "Yellow," and "Red."

The requirement*, "once red, the light always becomes green eventually after being yellow for some time",* can be interpreted in LTL logic as:

$\square(\boldsymbol{red} \rightarrow \bigcirc\left(\boldsymbol{red}\ \boldsymbol{U}\ \left(\boldsymbol{yellow} \wedge \bigcirc(\boldsymbol{yellow}\ \boldsymbol{U}\ \boldsymbol{green})\right)\right))$ (Baier & Katoen, 2008)

Where:

- $\square(\boldsymbol{red} \rightarrow \cdots)$: It means "always," so we can say that whenever the traffic light is red, the following condition is true.
- $\boldsymbol{red} \rightarrow$**…:** Starting point.
- $\bigcirc\left(\boldsymbol{red}\ \boldsymbol{U}\ \left(\boldsymbol{yellow} \wedge \bigcirc(\boldsymbol{yellow}\ \boldsymbol{U}\ \boldsymbol{green})\right)\right)$**:** The operator "◯" means "in the next state." The operator "**U**" (until) means that the state on the left will continue to be "true" until the condition on the right becomes true.
- "**Yellow**" means that the traffic light should be in that state at some point in the future.
- **"◯ (yellow U green)**" means that after the yellow state, in the next state, the traffic light will turn green after some time in the previous state.

2.9.2 CTL

As mentioned in subchapter 2.9.1, the LTL introduced by Pnueli is considered linear since it can treat time as a single path. However, LTL has limitations in expressing certain properties, such as the ability to return to the initial state in all computations.

Clarke and Emerson introduced a new branching temporal logic (Figure 2.18), represented as an infinite tree of states, where each moment can branch into multiple possible futures to overcome these limitations. CTL is used to specify system properties. It is a combination of branching and linear time operators. This means that its time model is a tree-like structure where the future is not determined, so there are different paths in the future, and any of these paths can be the true path that is realized (Baier & Katoen, 2008).

These logics include operators such as:

- p ∈ P.
- ϕ & ψ “conjunction”.
- ϕ | ψ “disjunction”.
- !ϕ “negation”.
- ϕ → ψ “implication”.
- ϕ ↔ ψ “equivalence”.
- EGϕ “exists a path where in all states ϕ holds”.
- AGϕ “ϕ holds in all states of all paths”.
- EXϕ “exists a path where ϕ holds in the next state”.
- AXϕ “ϕ holds in all next states of all paths”.
- E(ϕ U ψ) “exists a path such that ϕ holds until ψ becomes true”.
- A(ϕ U ψ) “in all paths ϕ holds until ψ becomes true”. (Pereira, et al., 2024)

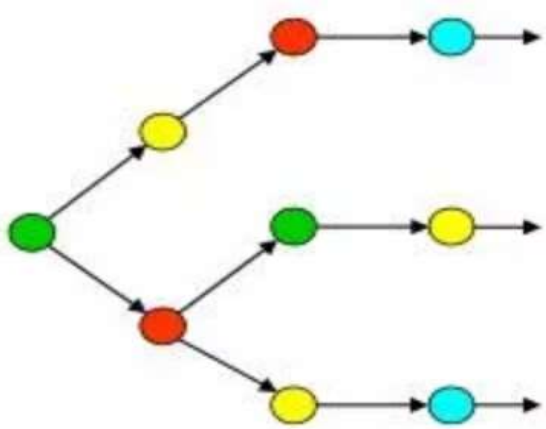


Figure 2.18: Computation Tree Logic. (SlideServe, 2014)

Example:

As an example of CTL specifications application, we have the case of a coffee and tea machine, as we can observe from the automaton (Figure 2.19) where the user needs to insert coins. In the case of one coin, the machine will serve coffee and in the case of two coins, it will serve tea. Once the request is completed, the machine returns to its initial state.

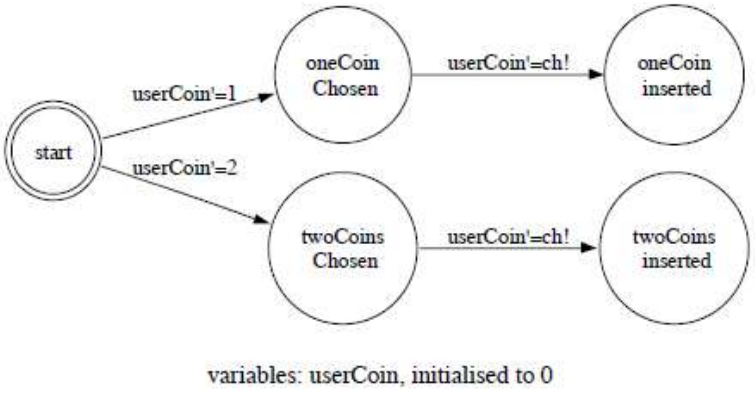


Figure 2.19: Automata for coffee and tea machine (Johnson, 2007)

$$AG(\text{“}oneCoinSelected\text{”} \rightarrow AF(\text{“}coffee\text{”}))$$

Where:

- **AG “always globally”**, means that the property must be true in all states. In other words, in all situations (or possible paths) over time, the associated condition must remain true.
- **“oneCoinSelected**, checks if the current state is "oneCoinSelected."
- $\rightarrow$, implication logical operator, it means that if the condition on the left is true, then the condition on the right must also be true.
- **AF “Eventually finally”**, states that there is a future state where "coffee" is true, and a coffee it will be served.

These conditions ensure that once the user has chosen to insert one coin coffee must be served (Johnson, 2007).

We can conclude that LTL logic is ideal for deterministic systems, as it allows expressing properties that depend on temporal sequences, ensuring that something will always or eventually occur. On the other hand, CTL logic is used for non-deterministic systems, being necessary to express properties that depend on multiple possible paths. The application of these two techniques will depend on the system to be modeled and the desired behavior.

The combination of the two logics can be extremely useful in situations where the system's behaviour needs to verify global properties of a sequence of states, as well as properties of branched paths. This offers a more robust verification of the system.

2.10 Some Application Examples in Railway

This chapter addresses the importance of railway safety and the application of formal verification techniques, such as model checking, to ensure the efficiency of control systems. Three significant examples are presented to highlight how these techniques allow detecting and correcting design flaws in the early stages of development, ensuring the accuracy, safety, and reliability of critical systems.

Article 1: ***Verification of Railway Control Systems Using Model Checking and CTL, Explained Through a Case Study*** (Lukács & Bartha, 2024)

This article investigates the use of model checking to assist railway engineers in the design and verification of the safety of railway control systems. For this purpose, the UPPAAL framework was used to demonstrate the use of this technique through a case study of a vehicle detection point. UPPAAL allows verifying formal properties of the system and generating counterexamples in case of violations, which in turn provides designers with relevant information about the system's behavior, allowing the identification of flaws or defects in the design.

One of the main advantages of this approach is the abstraction of mathematical details that may cause doubts, allowing railway engineers to better understand the results of formal verification. However, the application of model checking is subject to various constraints, depending on the method, the tool used, and the specifications of the modelled system.

Article 2: ***Modeling and Verification of an Automatic Train Protection System*** (Xionga, et al., 2010)

This article highlights the importance of ATP systems and the need to ensure the efficiency of software models. It proposes the integration of modelling and verification techniques in Model-Driven Architecture (MDA), using tools such as UPPAAL for formal verification. This approach allows modelling the ATP system from different perspectives and levels of abstraction, which will reduce complexity and ensure the correctness of the system's essential properties. The utility of model checking should be highlighted for its ability to detect design errors in the early stages of development, ensuring the safety and efficiency of the system, despite the challenges and costs associated with formal verification.

Article 3: ***Verification of railway interlocking systems*** (Busard, et al., s.d.)

This article highlights the importance of interlocking systems in the computerized control of signaling devices to ensure the safe operation of trains. It proposes the development of an executable model in NuSMV, based on application data, and the development of a tool to automatically translate this data into the model. The verification of safety properties is carried out using the PyNuSMV library, which allows detecting design errors in the early stages of development and improving the efficiency of the validation process. Model checking is essential to ensure the correctness and safety of ATP systems, significantly reducing the reliance on manual testing and contributing to the management of software complexity. The proposed approach demonstrates how the integration of modelling and verification techniques can significantly enhance the safety and efficiency of interlocking systems.

Comparative analysis

The three articles address the use of formal verification techniques, such as model checking, in verifying the safety and efficiency of ATP systems. The first article uses the UPPAAL framework to verify safety in vehicle detection, highlighting the abstraction of mathematical details to facilitate understanding of the results. The second article focuses on ATP systems, proposing the integration of modeling and verification techniques to reduce complexity and ensure the correctness of essential properties. Finally, the third article addresses interlocking systems, proposing the creation of an executable model in NuSMV and the use of the PyNuSMV library to verify safety properties, in order to improve the efficiency of the validation process.

3 Model Checking Tools

3.1 NuSMV

NuSMV is an advanced symbolic model-checking tool based on OBDD (Ordered Binary Decision Diagrams), capable of describing finite-state systems using a structured language. It is primarily focused on the verification of synchronous systems, enabling the validation of models with specifications in the temporal logics CTL and LTL. If a specification is not met, the tool generates a counterexample, facilitating system analysis and debugging. Additionally, NuSMV provides the functionality to simulate the specified model, promoting an interactive approach to formal verification (Cimatti, et al., 1999).

As a reengineering and extension of the SMV (Symbolic Model Verifier), developed by K. McMillan at Carnegie Mellon University (CMU) in the 1990s, NuSMV preserves the robustness of the original system but brings significant improvements. The SMV, the first symbolic model checker, had a great impact on the field of formal verification, and NuSMV emerges as an enhancement of this concept. Developed as part of a collaborative project between CMU and ITC-IRST (Institute for Scientific and Technological Research), NuSMV benefited from the contributions of researchers such as A. Cimatti, E. Clarke, F. Giunchiglia, A. Morichetti, and M. Roveri. Constantly evolving, NuSMV remains an essential tool for verifying dynamic systems, expanding the functionalities of SMV and offering new capabilities.

In addition to its focus on formal verification, NuSMV stands out for its flexibility in describing systems, which can range from fully synchronous to fully asynchronous. Its language allows the creation of systems in a modular and hierarchical manner, facilitating the definition of state transition relations. Among its additional functionalities compared to SMV, NuSMV offers:

- **Interactivity:** An interactive shell that enables the execution of separate computation steps.
- **Invariant Analysis:** Allows real-time verification of invariants during accessibility analysis.
- **Partitioning Methods:** Offers the possibility to partition the model conjunctively or disjunctively, with inspection and ordering options based on heuristics.
- **Verification of LTL and PSL:** Supports specifications in LTL and PSL (Property Specification Language), including automatic conversion of LTL formulas to CTL.
- **SAT-Based Verification:** Utilizes satisfiability techniques to perform bounded model verification (NUSMV, 2014-2023).

The NuSMV offers two main modes of interaction with the user: batch and interactive. The batch mode (Figure 3.1) follows a predefined algorithm and is similar to the original CMU SMV, allowing for the automatic execution of verification tasks. The interactive mode, on the other hand, provides greater flexibility, enabling the user to control and activate calculation steps independently through shell commands.

In the interactive shell, we can perform steps such as reading the model, transforming hierarchical descriptions into a flattened form, encoding scalar variables into boolean ones, and building models using BDDs (Binary Decision Diagrams). The shell also allows for the configuration of BDD-related options and supports a scripting language for developing customized verification techniques.

Binary decision diagrams are data structures used to represent boolean functions. The non-terminal nodes in a diagram are called decision nodes, and each decision node is labeled by a boolean variable and has two child nodes, referred to as low child and high child. The terminal nodes can be of two types: 0-terminal, representing the Boolean value false, and 1-terminal, representing the boolean value true (Bertino, 2012).

These features make NuSMV a highly flexible and powerful tool for the formal verification of models in dynamic systems, meeting the needs of both users who seek automated solutions and those who desire greater control and customization during the verification process.

```
NuSMV > go
NuSMV > pick_state -r
NuSMV > print_current_state -v
Current state is 1.1
speed = 0
mode_atp = initial
check_failure = FALSE
NuSMV > simulate -r -k 3
********  Simulation Starting From State 1.1   ********
NuSMV > show_traces -t
There is 1 trace currently available.
NuSMV > show_traces -v
    <!-- ################## Trace number: 1 ################## -->
Trace Description: Simulation Trace
Trace Type: Simulation
  -> State: 1.1 <-
    speed = 0
    mode_atp = initial
    check_failure = FALSE
  -> State: 1.2 <-
    speed = 1
    mode_atp = initial
    check_failure = FALSE
  -> State: 1.3 <-
    speed = 2
    mode_atp = initial
    check_failure = FALSE
  -> State: 1.4 <-
    speed = 3
    mode_atp = initial
    check_failure = FALSE
NuSMV >
```

Figure 3.1: NuSMV Interactive shell

Regarding the batch mode of NuSMV, it is based on the original CMU SMV, which performs a set of predefined steps in the verification process automatically. This mode includes the first five phases, such as reading the model, hierarchical transformation, and building models using BDDs. Variable reordering can be enabled through the "-reorder" option.

On the other hand, the NuSMV graphical interface complements the interactive shell, providing a visual method for editing and modifying models. The graphical interface includes a formula editor with dedicated buttons, making it easier to create verification specifications. As we can see from the example in figure 3.2, where a simple binary counter is implemented using modules, the “main” module defines three variables, bit0, bit1, and bit2, which are instances of the “counter_cell” module. Each bit transmits a carry signal (carry_out) to the next bit as input (carry_in). The property to be verified, SPEC, ensures that the carry signal of the third bit (bit2.carry_out) will eventually be activated, that is, it will be true at some point in the future (Cimatti, et al., 2000).

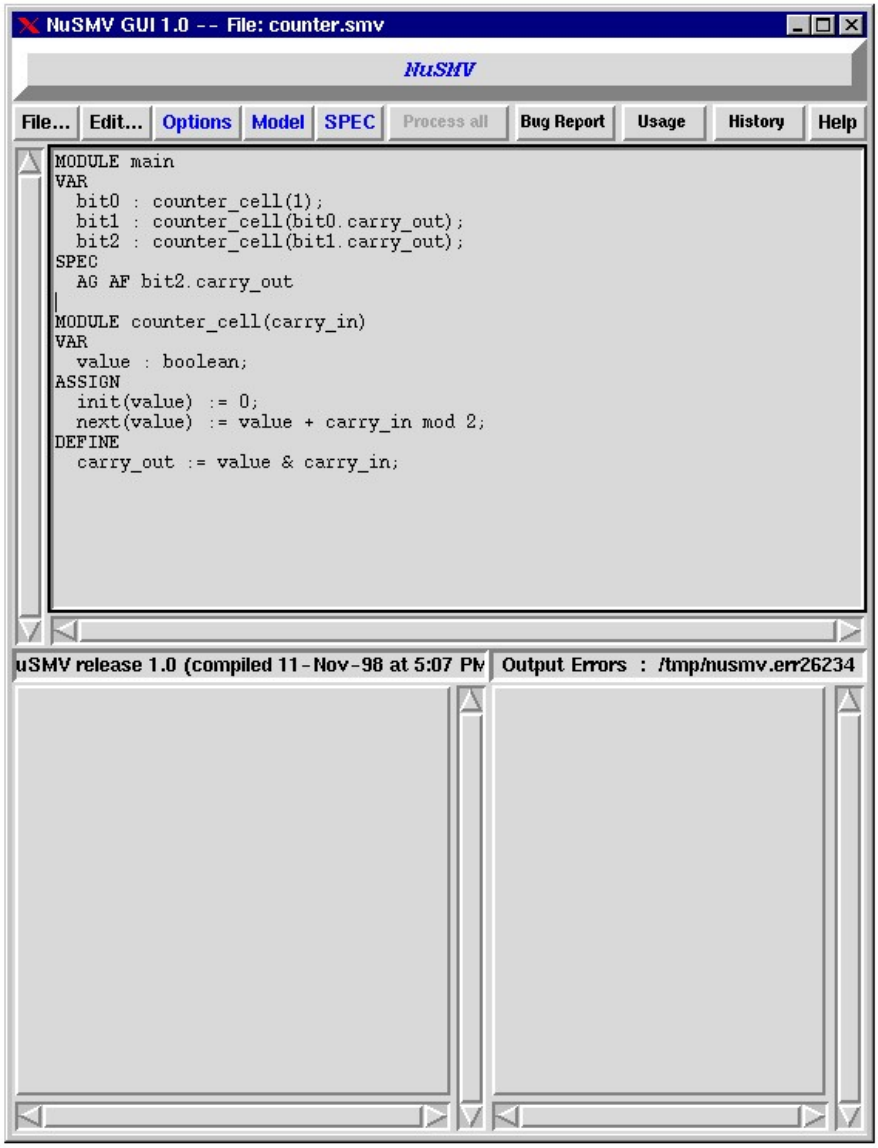


Figure 3.2: NuSMV Graphical Interface (Cimatti, et al., 2000)

NuSMV provides advanced functionalities for model verification, such as specialized algorithms for invariant verification and heuristics that help mitigate the state explosion problem. It supports RTCTL (real-time CTL) specifications, allowing the definition of time intervals for events. Additionally, it verifies invariants using fixed-point techniques and "on the fly" verification, where invariants are checked during the model exploration process without the need to explore all system states.

This tool enables the calculation of quantitative information about the Kripke structure, such as delay bounds between events. It also supports LTL model checking and allows the partitioning of the transition relation into small BDDs, which facilitates verification and reduce computational complexity (Cimatti, et al., 2000).

3.1.1 Functionalities

In terms of features, NuSMV offers an encoding that converts non-boolean variables into boolean variables, which represents systems with enumerated types. The encoding method is inspired by CMU SMV, grouping boolean variables into blocks that are organized into packages of BDDs.

Regarding reachability analysis, NuSMV uses an advanced algorithm to compute reachable states from initial states. Future improvements are planned, such as the use of "don’t care" sets to simplify this process.

When it comes to property verification with CTL, NuSMV employs efficient fixed-point algorithms. Whenever the reachable states have been previously computed, performance is optimized by restricting the verification to only the relevant states. NuSMV also includes a feature for verifying the completeness of the transition relation, useful for identifying the presence of "deadlock" states. This process is carried out through fundamental model-checking operations.

NuSMV allows the generation of counterexamples when a property is not satisfied, using algorithms like those used by CMU SMV. This functionality allows for the navigation and inspection of counterexamples for unverified properties. Similar to a programming debugger, the user can switch between different states or counterexamples and evaluate CTL expressions in specific states through commands in the interactive shell, providing a dynamic and efficient interaction in the verification process.

Another feature of NuSMV is the ability to dynamically modify the model without needing to restart the calculations. Through commands in the interactive shell, it is possible to adjust initial states, invariants, transition relations, and even add new fairness constraints. All these changes can be reversed, allowing the system's original state to be restored, facilitating the exploration of different modifications.

Finally, NuSMV includes an integrated help functionality that provides concise explanations for each command in the interactive shell. It also offers comprehensive documentation in text format and HTML format, accessible via a web browser, making the learning process more practical and accessible (Cimatti, et al., 2000).

3.1.2 System architecture

The following section will present the description of the architecture of the NuSMV system (Figure 3.3). This architecture is organized into blocks, with each block designed to implement a specific set of functionalities, while ensuring efficient communication with the other system modules.

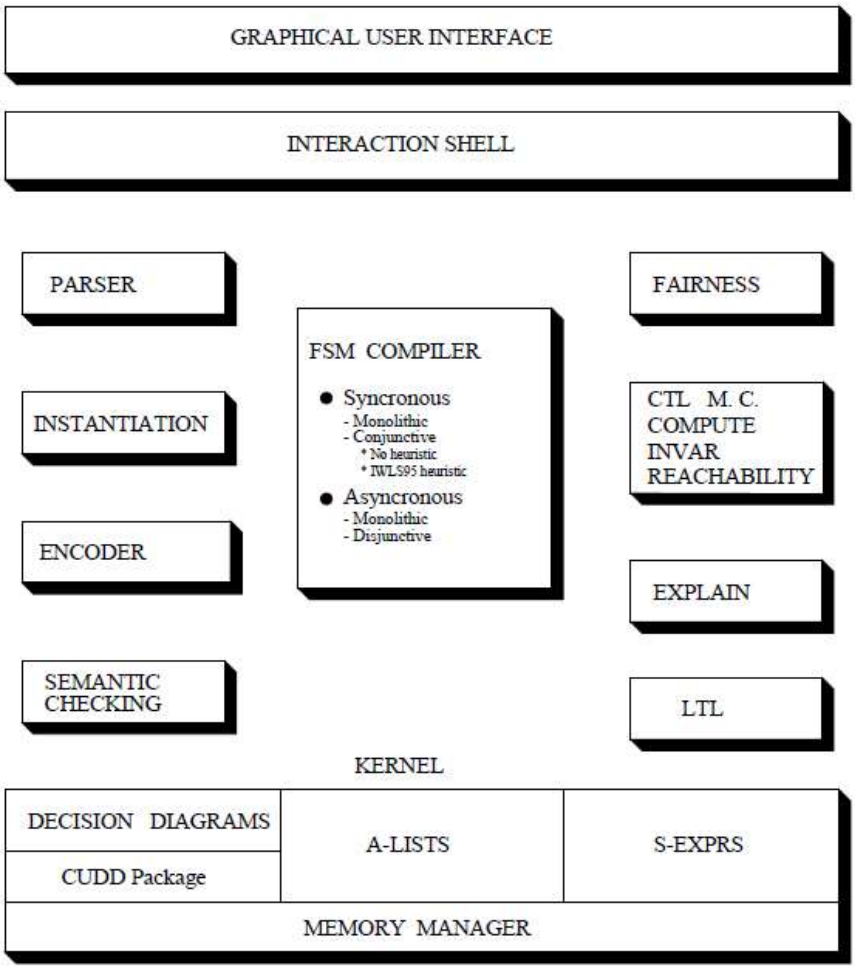


Figure 3.3: System Architecture (Cimatti, et al., 2000)

- **Parser:** This module is responsible for processing files written in the NuSMV language, verifying their syntactic correctness, and building a parse tree that represents the internal format of the file. Additionally, these routines are used for commands in the interactive shell, allowing the specification of new properties directly from the command line.

- **Instantiation:** This module is responsible for instantiating the declared modules and building a representation of the finite state machine (FSM), including the transition relation, the initial states, and the fairness conditions.

- **Encoder:** This module is responsible for converting data types and finite ranges into boolean domains, enabling the application of different encoding policies. Currently, it uses the standard CMU SMV encoding, but there are plans to integrate other approaches, such as those from Word-Level SMV.

- **Semantic Checking:** It implements all the necessary semantic checks on the model, such as ensuring the absence of circular definitions.

- **FSM Compiler:** This module is responsible for implementing and manipulating FSMs using BDDs, supporting both monolithic and partitioned representations. Thanks to its input language independent heuristics, it can be applied across various domains, although it might not be ideal for specialized model checkers like Verification Interacting with Synthesis (VIS). It also provides primitives for computing the image and pre-image of states, regardless of the transition representation method.

- **Fairness:** This module handles fairness constraints, providing the model checking module with information related to these constraints, if they are present.

- **Model Checking:** Responsible for implementing RCTL model checking functionalities, such as reachability, property verification, and quantitative characteristic computation.

- **Explanation Module:** This module provides routines for generating and inspecting counterexamples and proofs with varying levels of detail, presenting them as reusable data structures, independent of the FSM representation method.

- **LTL Module:** This module is responsible for calling an external program to translate LTL formulas into a tableau, which is then integrated into NuSMV, as well as generating a CTL formula to be verified on the synchronous product of the system and the tableau, reusing functionalities from other modules in the NuSMV architecture.

- **Kernel:** The NuSMV kernel provides low-level functionalities such as dynamic memory allocation and data structure manipulation, as well as basic and advanced BDDs primitives. It is integrated through a standardized interface, allowing for future replacement with other BDDs packages, ensuring flexibility and compatibility.

- **Interactive Shell:** The interactive shell provides full access to all system functionalities and is extensible, allowing for the creation of new commands.

- **Graphical User Interface:** Through the interactive shell, the graphical interface is a separate process that communicates with NuSMV, allowing users to inspect and configure environment variables and access all system functionalities (Cimatti, et al., 2000).

3.1.3 Specifications

To verify the FSM, NuSMV uses temporal specifications such as CTL, LTL, and PSL, including real-time quantitative characteristics. These can be placed within modules, with variables renamed for adaptation. NuSMV evaluates the truth of the specifications and, in case of falsity, generates a counterexample demonstrating the property failure.

LTL

In the case of LTL specifications, to specify these formulas, the keyword LTLSPEC must be introduced, and its syntax is as follows:

```
ltl_specification :: LTLSPEC ltl_expr;

                     LTLSPEC NAME name := ltl_expr;
```

Next, we proceed to describe the syntax of the LTL formulas recognized by NuSMV:

```
ltl_expr ::

next_expr -- a next boolean expression

 ( ltl_expr )

 ! ltl_expr -- logical not

 ltl_expr & ltl_expr -- logical and

 ltl_expr | ltl_expr -- logical or

 ltl_expr xor ltl_expr -- logical exclusive or

 ltl_expr xnor ltl_expr -- logical NOT exclusive or

 ltl_expr -> ltl_expr -- logical implies

 ltl_expr <-> ltl_expr -- logical equivalence
```

```
-- FUTURE
 X ltl_expr -- next state
 G ltl_expr -- globally
 G bound ltl_expr -- bounded globally
 F ltl_expr -- finally
 F bound ltl_expr -- bounded finally
 ltl_expr U ltl_expr -- until
 ltl_expr V ltl_expr -- releases
-- PAST
 Y ltl_expr -- previous state
 Z ltl_expr -- not previous state not
 H ltl_expr -- historically
 H bound ltl_expr -- bounded historically
 O ltl_expr -- once
 O bound ltl_expr -- bounded once
 ltl_expr S ltl_expr -- since
 ltl_expr T ltl_expr -- triggered
bound :: [ integer_number , integer_number ]
```

Where:

- **X:** It means that the truth of **p** at time **t** depends on the truth of **p** at the next time, **t + 1**.
- **F:** This means that if **p** is true at a certain time **t**, then at some point in the future, the proposition **p** will be true.
- **F bound:** It means that the proposition **p** will be true at some point within a given time interval.
- **G:** It means that if the proposition **p** is true at time **t**, regardless of the progression of time, **p** will always remain true at all future times. The proposition **p** cannot be false at any point after **t**.
- **G bound:** This formula ensures that, within any specific time interval after time **t**, the proposition **p** must be true throughout that interval.
- **Y:** This condition checks if **p** was true at the time immediately preceding the current time **t**, where: **p** is true at **t** > **t0** if **p** was true at time **t - 1**. **p** is false at **t0** because there is no "previous time" to **t0**.

- **Z:** Similar to the condition **Y**, this condition checks if **p** was true at the time immediately preceding the current time **t**, where: **Y p** is true at **t** > t0 if **p** was true at time **t - 1**, with the difference that **p** is also true at the instant **t0**.
- **H:** This condition means that **p** must be true at all times up to **t**, which implies that **p** must be continuously true, without interruptions, from the beginning until time **t**.
- **H bound:** It means that **p** is true at time **t** when **p** is true at all times within a given time interval preceding **t**. This can be useful to ensure that a condition has been consistently maintained during a specific past time period.
- **O:** This condition means that the proposition **p** will be true at time **t** if it has been true at least once in the past up to time **t**.
- **O bound:** It means that for this proposition to be true at time **t**, **p** must have been true at least once within the time interval up to **t**.
- p **S** q**:** It means that **q** must be true at some point **t0 ≤ t**, and throughout the entire interval between **t0** and **t**, **p** must be true at each time from **t0** up to time **t**. In other words, **p** must remain true until **q** occurs.
- p **T** q**:** It means that **p** must have been true at some point in the past up to time **t**, and **q** must be true at all times between **t0** and **t**, where **t0** is the moment when **p** was true.
- p **U** q: This condition means that **p** must remain true until **q** becomes true at some point **t0 ≥ t**. It is useful for ensuring that a given condition remains valid until another condition occurs in the future.
- p **V** q: It means that **q** must remain true until **p** becomes true (Cavada, et al., 2005).

CTL

In NuSMV, to specify formulas in CTL, it is necessary to use the keyword "CTLSPEC" before the desired formula. Although it is possible to use the deprecated keyword "SPEC", it is not recommended for new projects. Therefore, the syntax should be presented as follows:

```
ctl_specification :: CTLSPEC ctl_expr;

       SPEC ctl_expr;

       CTLSPEC NAME name := ctl_expr;

       SPEC NAME name := ctl_expr;
```

Next, we proceed to describe the syntax of the CTL formulas recognized by NuSMV:

```
ctl_expr ::
simple_expr -- a simple boolean expression
       ( ctl_expr )
       ! ctl_expr -- logical not
       ctl_expr & ctl_expr -- logical and
       ctl_expr | ctl_expr -- logical or
       ctl_expr xor ctl_expr -- logical exclusive or
       ctl_expr xnor ctl_expr -- logical NOT exclusive or
       ctl_expr -> ctl_expr -- logical implies
       ctl_expr <-> ctl_expr -- logical equivalence
       EG ctl_expr -- exists globally
       EX ctl_expr -- exists next state
       EF ctl_expr -- exists finally
       AG ctl_expr -- forall globally
       AX ctl_expr -- forall next state
       AF ctl_expr -- forall finally
       E [ ctl_expr U ctl_expr ] -- exists until
       A [ ctl_expr U ctl_expr ] – for all until
```

Where:

- **EX:** It means that, in the near future, the proposition **p** becomes true in a state accessible from the current state.
- **AX:** It means that there cannot be any transition from **s** to a future state where **p** is not true.
- **EF:** It means that the proposition **p** will be true at some point after the current state, regardless of how many transitions are needed to reach it.
- **AF:** It means that regardless of the transitions the system takes, **p** will inevitably be fulfilled at some point in the future.
- **EG:** This means that the proposition p will always be true along a path that never ends, which means that p remains valid indefinitely throughout all transitions in the system.

- **AG:** This means that no matter how the transitions occur, if the system continues indefinitely, **p** will remain true at all times, that is, in every state along the entire infinite path.
- **E [p U q]:** This means that **p** must remain true until the moment **q** is fulfilled, and this must happen at some point in the future within the sequence of transitions.
- **A [p U q]:** This means that the proposition **p** will be true in all states until the proposition **q** becomes true in some future state.

3.2 UPPAAL

Uppaal, developed by Uppsala University and Aalborg University, is a tool used for the verification of real-time systems modeled as networks of timed automata. Released in 1995, it has proven to be a valuable tool in various applications, such as communication protocols and multimedia systems. Over time, Uppaal has been updated, introducing many features, including efficient data structures, a distributed version, and improvements for UML state diagrams (David, et al., 2018).

It consists of two main components: a graphical user interface implemented in Java and a model-checking engine. Users can model a system using timed automata, which are finite state machines with clocks, simulate its behavior, and verify properties.

The verification engine uses on-the-fly verification and symbolic techniques to efficiently solve constraint systems. It primarily verifies invariants and reachability properties. Some more complex properties can be validated using testing automata or debugging tools within the system. This model-checking process allows for exhaustive exploration of possible system behaviors to ensure they meet the intended design. UPPAAL uses the XTA and XML languages and supports the CTL verification language, enabling advanced and flexible specifications for system properties (UPPAAL, 2009).

Uppaal also introduced Uppaal Statistical Model Checking (SMC) to address gaps in traditional timed automata models, as they are not sufficient for complex cyber-physical systems with continuous, dynamic, or stochastic behaviors. This new branch allows the use of networks of automata with clocks that evolve at different rates, specified through methods such as ordinary differential equations (David, et al., 2018).

Regarding its interface (Figure 3.4), the main window of Uppaal consists of two main parts: the menu and the tabs. The menu provides access to integrated help, which covers syntax and graphical interface details. As for the three tabs, they correspond to the components of Uppaal, which are the editor, simulator, and verifier.

In the editor tab, process models are defined, helping to model systems with similar processes by allowing the use of symbolic variables and constants as parameters. The models can also include local variables and clocks. By default, whenever Uppaal is launched, it creates an initial location, which users can modify by adding new locations and edges to build automata using the available tools.

Additionally, at the bottom of the editor, there is a bar that opens a table where compilation errors will be displayed (UPPAAL, 2009).

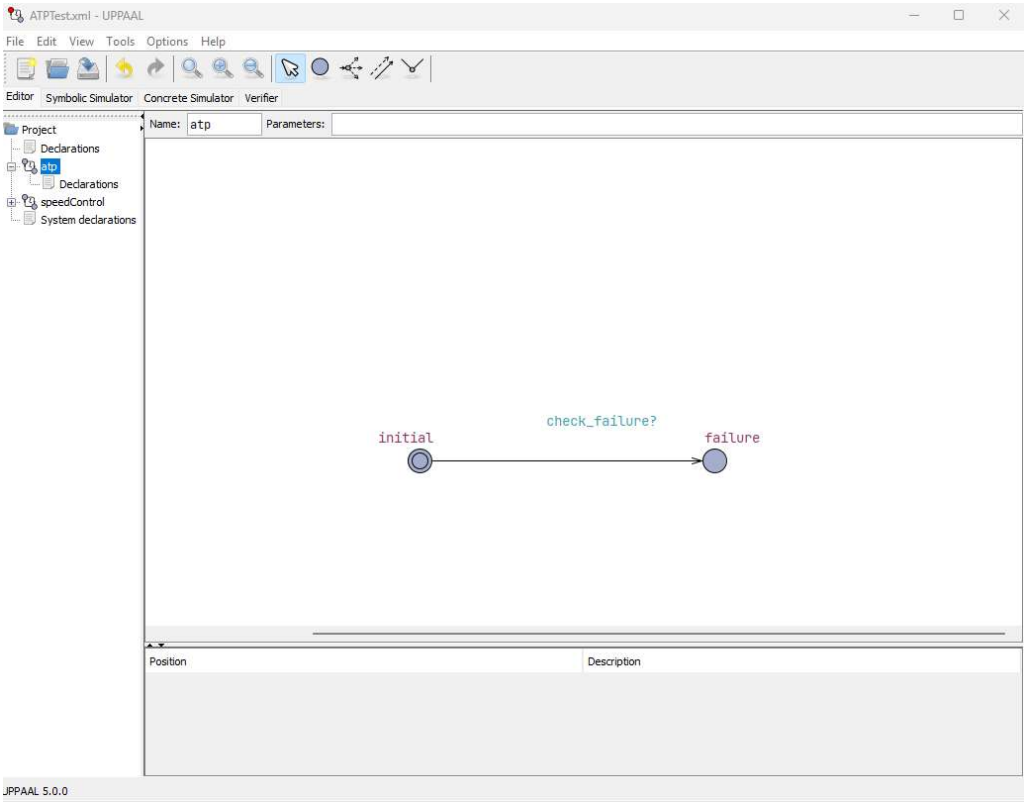


Figure 3.4: Main Window

In Figure 3.5, we can observe the simulator tab. In this tab, the upper-left section displays controls for selecting transitions, while the lower part allows users to work on the existing trace. In the middle section, the declared variables are shown, as well as their updates throughout the simulation. In the right section, the system's processes can be visualized.

To simulate, simply select a state in the upper corner of the left section, and subsequently, in the right section, you will see the state transitions. In the event that no further transitions are possible, the system will enter a deadlock.

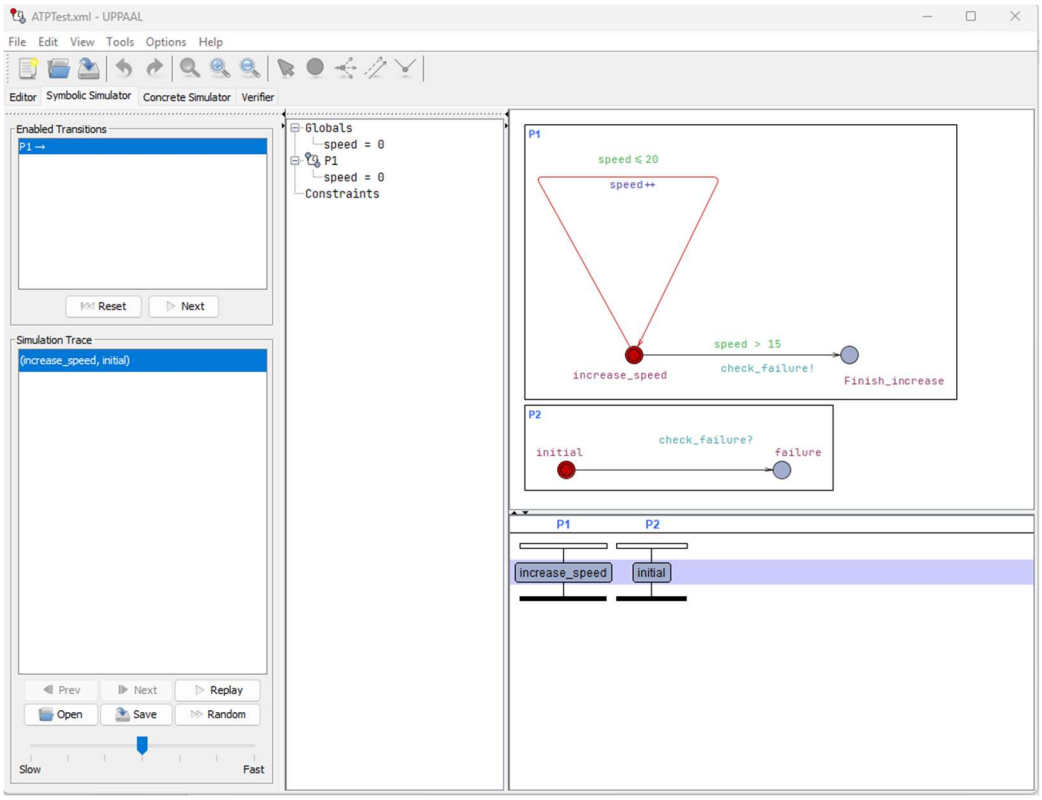


Figure 3.5: Graphical Simulator

In the "Verifier" tab (Figure 3.6), properties to be verified are specified. In the verifier view, the upper section allows users to specify queries, while the lower section logs communication with the model-checking engine.

By clicking “Check,” UPPAAL will verify if the system meets the specification. In the lower tab, under status, you can check whether the property is satisfied. If the property is satisfied, a message saying “Property is Satisfied” will be displayed, along with a green indicator for the respective property in the overview tab. If the property is not satisfied (Figure 3.7), a message saying “Property is not Satisfied” will appear in the lower tab, accompanied by a red indicator in the overview tab. (UPPAAL, 2009).

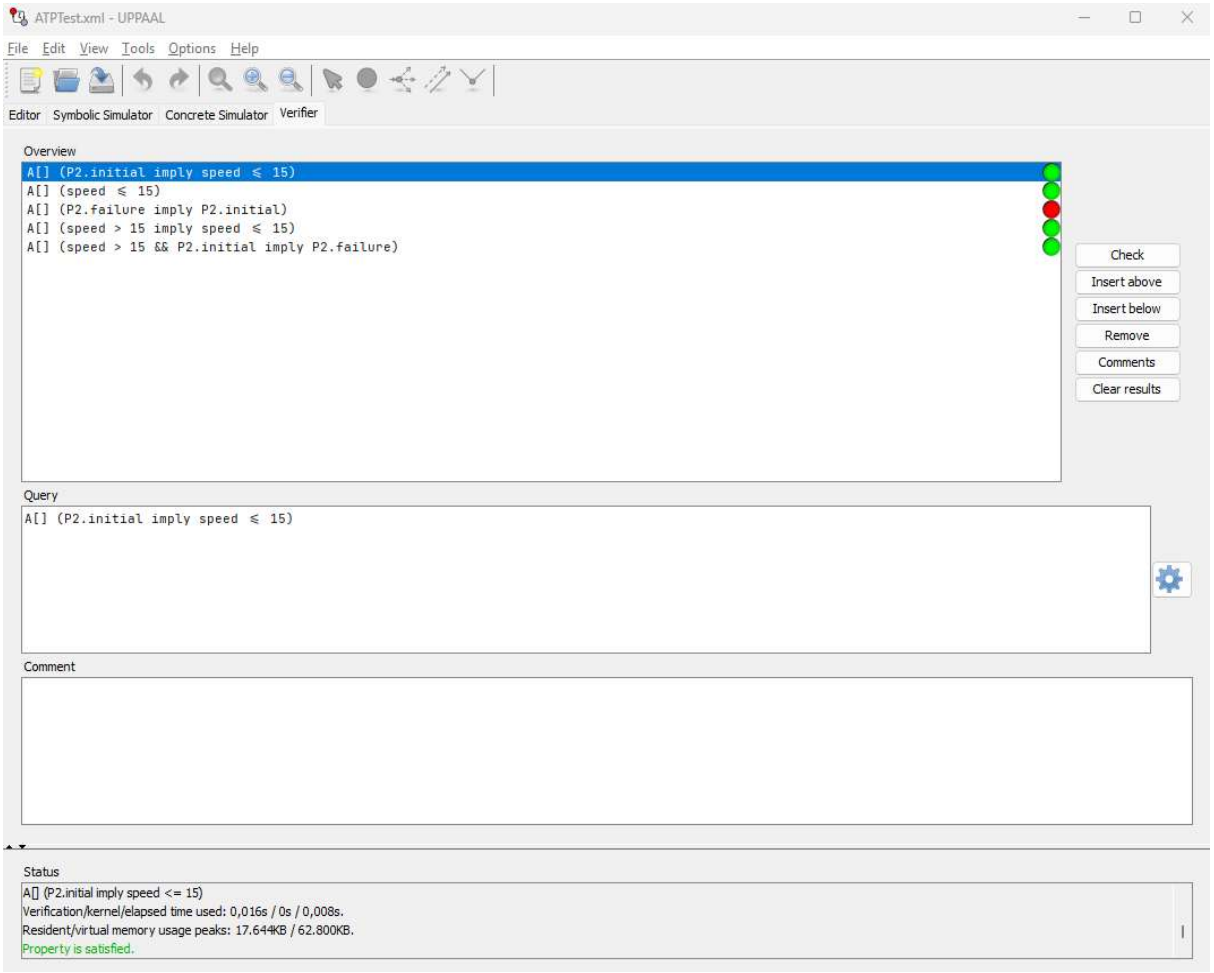


Figure 3.6: Verifier View with property satisfied

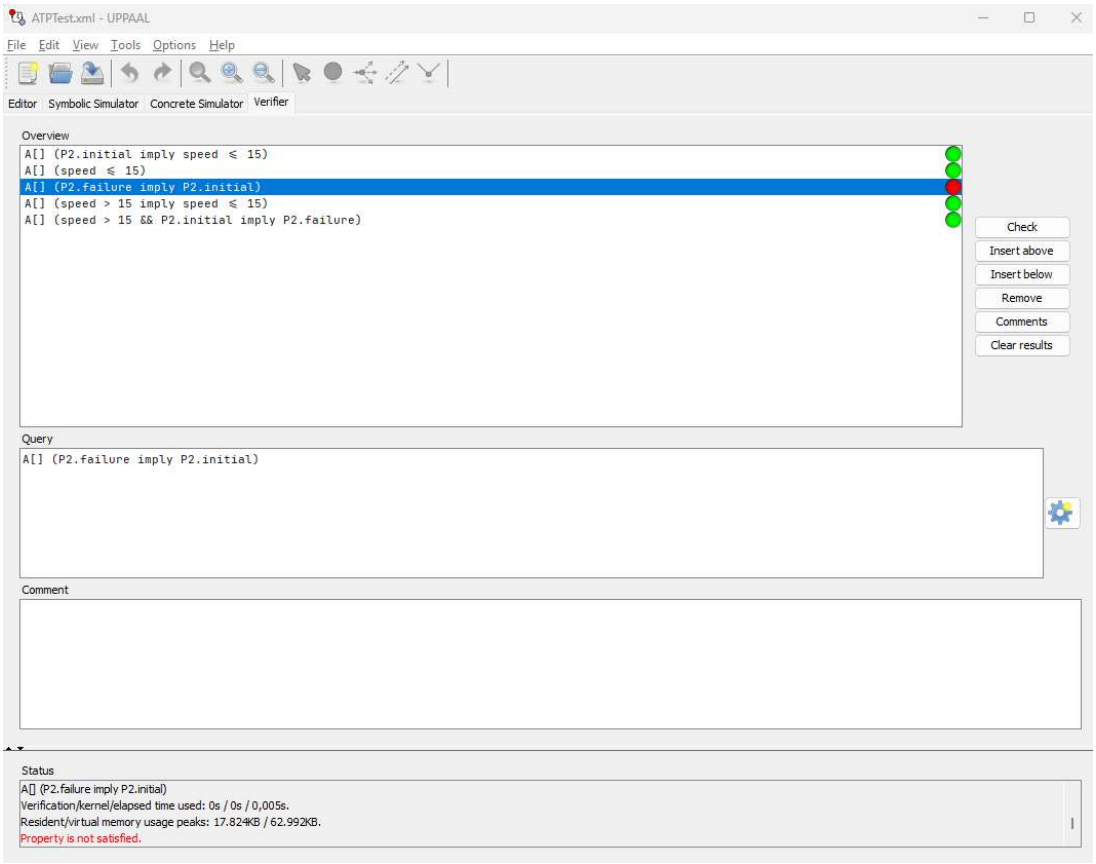


Figure 3.7: Verifier view with property not satisfied

3.2.1 Specifications

The UPPAAL verifier supports various types of queries to verify system properties, such as:

- **E<> p:** It means that the condition **p** will eventually be satisfied in some path within the model.
- **A[] p:** It means that, in all possible paths of the model, the property **p** will always hold true in all future states of those paths, ensuring that the condition will maintain its state continuously and uninterruptedly throughout the system.
- **E[] p:** It means that the property **p** will always be true in all future states in at least one execution path.
- **A<> p:** It means that the property **p** will eventually be true in all possible execution paths of the system.
- **p --> q:** It means that whenever the property **p** is true, then **q** must eventually be true at some point in the future, ensuring that the occurrence of an event **p** guarantees that another event **q** will occur in the future.

3.2.2 What’s about time in UPPAAL?

The concept of time in UPPAAL is based on a continuous time model, implemented symbolically through regions. Instead of working with exact time values in each state, UPPAAL uses time differences to represent the system's temporal properties.

Clocks play a fundamental role in managing time in UPPAAL. They are defined in the global Declarations section and allow the evaluation of time progression. Additionally, synchronization channels, known as "chan," are used to coordinate interactions between system components, such as resetting a clock during a synchronization event.

Through simulation and queries, UPPAAL enables users to analyze how temporal properties behave across different execution paths (UPPAAL, 2009).

4 Automatic Train Protection System

Since we are not going to model a specific system, to illustrate the architecture of a system of this kind, the architecture of the EBICAB 700 system (Figure 4.1), known as Convel in Portugal, will be demonstrated as an example. The functional requirements of the system that must be verified on the ATP system model during the model checking activity are described in this section.

4.1 System Architecture

This system consists particularly of fixed equipment in the infrastructure and equipment in the motor units. The fixed equipment in the infrastructure is used to send information to the train when it passes by the equipment installed on the tracks.

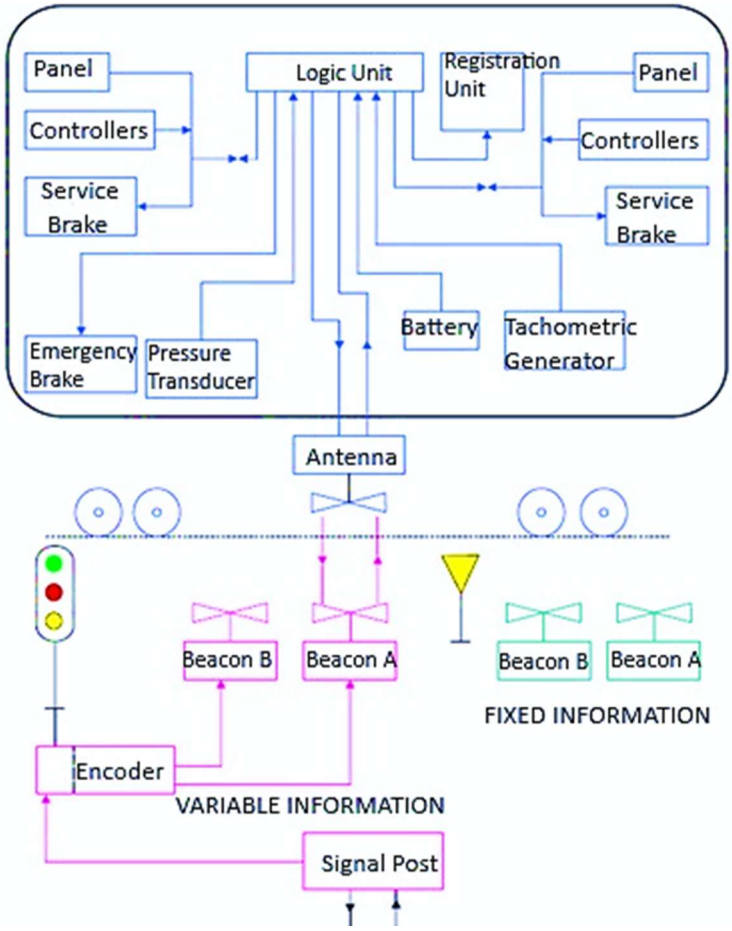


Figure 4.1: Block Diagram for ATP system Architecture
(Transportes, s.d.)

On the tracks, we can find the following equipment:

Beacons: These are installed along the centerline of the track. They contain fixed or variable information that is transmitted to the train's onboard system through an antenna installed on it.

Figure 4.2: Beacon (Pandrol, 2024)

Encoder: This is responsible for encoding all the information received from the signaling station or traffic light signals and transmitting it to the beacons.

Figure 4.3: Encoder (SIEMENS, 2024)

Signaling Station: A command center where all road traffic circulation in a given area is controlled and managed.

Regarding the equipment installed on the train, we can find the following systems:

- **Antenna**: Responsible for energizing the beacons, it is installed on the underside of the train and receives the information stored in the beacons, which is then transmitted to the logic unit.
- **Logic Unit**: The logic unit is responsible for processing all the information received from the beacons, as well as data from the train itself, including information from the dashboard, pressure transducer, the position of the

reversing switch on the motor unit, and the tachometric generator. It then sends the information to the dashboard, speedometer, braking system, and registration unit.

- **Dashboard**: The dashboard is responsible for communication between the system and the train operator, transmitting information calculated by the logic unit. On this panel, the train operator inputs the train's data, which is then transmitted to the logic unit.

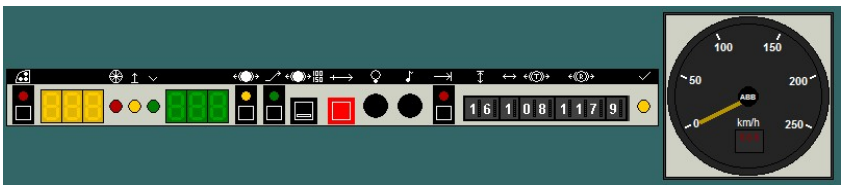


Figure 4.4: Dashboard (Martins, 2014)

Real Speed Alarm and Recording Unit: The real speed alarm and display unit includes an analog/digital speedometer and three alarm indicators that signal brake isolation and the operation of the recording unit. Additionally, it features an audible alarm and a button to test the lights and the audible alarm.

Brake Interface: The system has interfaces between the logic unit and the service and emergency brake systems in the motor units, in order to automatically apply maximum service or emergency braking if the train's actual speed exceeds certain values relative to the allowed speed.

Tachometric Generator: This generator provides the logic unit with relevant information for calculating the actual speed.

Pressure Transducer: This sensor provides information about the pressure in the general braking system pipeline.

Main System Switch: Responsible for turning the system on and off.

Recording Unit: This unit is responsible for recording all relevant events that occur during the train's operation. It is connected to the logic unit, which receives information about the status of the system's components. The following data will be recorded:

- Train initialization parameters.
- Train speed.
- Actions performed by the train operator on any of the dashboard buttons.
- Error alarms indicated by the system.
- Automatic braking actions.

- Pressure variation in the general brake pipe.
- Speeds displayed on the panels.
- Information from the beacons (Instituto da Mobilidade e dos Transportes, s.d.).

4.2 Requirements

Table 4.1 contains both system, subsystem and application requirements for ATP system that are to be verified in both NuSMV and UPPAAL during model checking activity.

Disclaimer:

All the numerical values mentioned in the requirements of the system are arbitrary values and do not represent the actual values in the ATP System.

Table 4.1: Requirements for ATP system

ID	Description	Level
APP-001	**When** the system is not in Shunting mode, **and** the intended direction of travel is not known, **and** the vehicle is not at a standstill, **then** the system **shall** set the roll-away risk indicator to True, **and** activate the driver warning.	Application
APP-002	**When** the vehicle is moving, **and** passing the stop signal is not authorized, **and** the current stop signal status is *Red*, **and** the previous stop signal status was not *Red*, **and** the first brake response has reached a *complete stop*, **then** the system **shall** activate the *emergency brake*.	Application
APP-003	**When** a temporary speed limit has been initiated, and the vehicle has traveled more than 100 meters beyond the point where the limitation became active, and signal monitoring is not active, **then** the system shall enable the *speed limit reset control*.	Application

APP-004	**When** the system is operating in *Standard mode* or has entered a *beacon error condition*, **and** the sum of the desired speed **and** the additional safety margin exceeds or equals the currently applied *emergency brake deceleration curve*, **then** the system **shall** update the *emergency brake deceleration curve* to match the sum of the desired speed and the additional margin.	Application
SUBSYS-001	**When** the current stop signal status is *Red*, and the train's speed is greater than or equal to the applicable speed limit, and the vehicle is not at a standstill, **then** the system shall activate the *emergency brake*.	Subsystem
SYS-001	**When** the vehicle has been operating in *shunting mode* for a distance greater than or equal to 500 meters, **then** the system shall indicate that *shunting mode* must be terminated.	System

5 Models, Verification, and Results

The validation of system requirements is an essential process to ensure compliance with established specifications. In this chapter, we will discuss the analysis of formal verification using specialized tools such as NuSMV and UPPAAL, which enable the automatic verification of system properties in order to identify potential requirement failures. This ensures that the system operates in a predictable and safe manner. The obtained results will be presented and analyzed to assess the system's adherence to the specified requirements, providing insights into possible improvements and necessary adjustments.

5.1 Model of ATP System in NuSMV

Requirement APP-001:

This requirement instructs the system to consider the vehicle to be in a rollway state if it is not in shunting mode, the travel direction is unknown, and the vehicle is not in standstill. To alert the operator, a driver warning is activated, ensuring that appropriate actions are taken regarding the vehicle.

Figure 5.1 illustrates the expected modeling of our requirement in NuSMV.

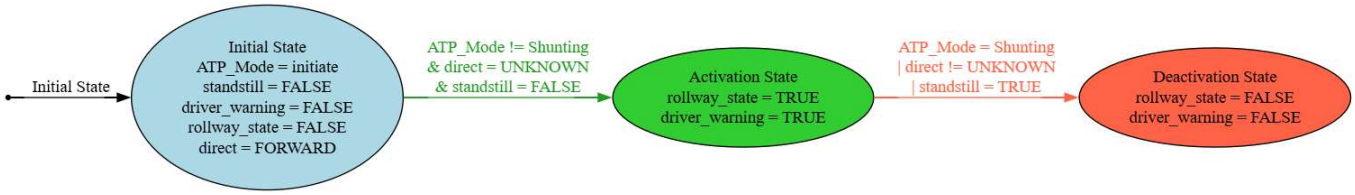


Figure 5.1: Block Diagram for requirement APP-001

To model this requirement, it was developed in a ".smv" file using "Notepad++."

This model aims to ensure that the vehicle does not move uncontrollably and that it alerts the driver. The model defines variables to represent the vehicle's operating mode, called "ATP_Mode"; the vehicle's direction, "direct"; the stop status, "standstill"; as well as the driver alert states, "driver_warning" and uncontrolled movement state, "rollway_state".

The system's main logic can be summarized as follows: if the vehicle is not in maneuver mode, called “Shunting” the vehicle’s direction is unknown, “UNKNOWN,” and the train is not at standstill, then the system activates both the rollway state as true and triggers the driver warning. Otherwise, both remain deactivated.

Temporal properties, such as LTL, were specified to ensure that the described behavior is consistently upheld.

```
MODULE main

--variable declaration
VAR
--possible states
ATP_Mode: {initiate, Shunting, against, red_light_crossing};
--Possible states for the travel direction.
direct: {UNKNOWN, FORWARD, REVERSE};
--standstill status
standstill: boolean;
--Driver warning status
driver_warning: boolean;
--rollway status
rollway_state: boolean;

ASSIGN
	--initialization
	init(ATP_Mode) := initiate;
	init(standstill) := FALSE;
	init(driver_warning) := FALSE;
	init(direct) := FORWARD;
	init(rollway_state) := FALSE;

next(rollway_state) := case
	ATP_Mode != Shunting & direct = UNKNOWN & standstill = FALSE : TRUE;
	ATP_Mode = Shunting | direct != UNKNOWN | standstill = TRUE : FALSE;
	TRUE : rollway_state;
	esac;

next (driver_warning) := case
	ATP_Mode != Shunting & direct = UNKNOWN & standstill = FALSE : TRUE;
	ATP_Mode = Shunting | direct != UNKNOWN | standstill = TRUE : FALSE;
	TRUE : driver_warning;
	esac;
-----------------------------Properties-----------------------------------
```

LTLSPEC

```
G ((ATP_Mode != Shunting & direct = UNKNOWN & !standstill) -> (next(rollway_state)
= TRUE & next(driver_warning) = TRUE));
```

This property guarantees that the system activates the driver’s warning and sets the rollway state to true whenever the defined conditions are met.

```
LTLSPEC

G ((ATP_Mode = Shunting | direct != UNKNOWN | standstill) -> (next(rollway_state)
= FALSE & next(driver_warning) = FALSE));
```

Ensure that no changes occur in the rollway state or driver warning status if any of the required conditions are not fulfilled.

```
LTLSPEC

G F TRUE;
```

Finally, the last property verifies that the model does not enter a deadlock state, ensuring continuous and reliable operation.

Figure 5.2 presents the results of the verification of the specified properties for the modeled requirement. As shown in the figure, the model does not exhibit deadlocks. Furthermore, whenever the conditions are met, the system transitions to the desired states. If any condition is not fulfilled, the states remain unchanged, ensuring the model's consistency.

```
-- specification  G ( F TRUE)  is true
-- specification  G (((ATP_Mode != Shunting & direct = UNKNOWN) & !standstill) ->
 (next(rollway_state) = TRUE & next(driver_warning) = TRUE))  is true
-- specification  G (((ATP_Mode = Shunting | direct != UNKNOWN) | standstill) ->
(next(rollway_state) = FALSE & next(driver_warning) = FALSE))  is true
```

Figure 5.2: Requirement verification results for APP-001

Requirement APP-002:

This requirement states that the system must activate the emergency brake in situations where the vehicle is in motion, is not authorized to pass a stop signal, the signal is red and has changed from a previous state, and it has been detected that the vehicle has already come to a complete stop. Despite the immobilization, the activation of the emergency brake is required as a safety measure, ensuring that the stop has been effectively controlled in the face of a risk situation or signal violation.

To better understand how to model this requirement, a state diagram was created to represent the system's behavior (Figure 5.3).

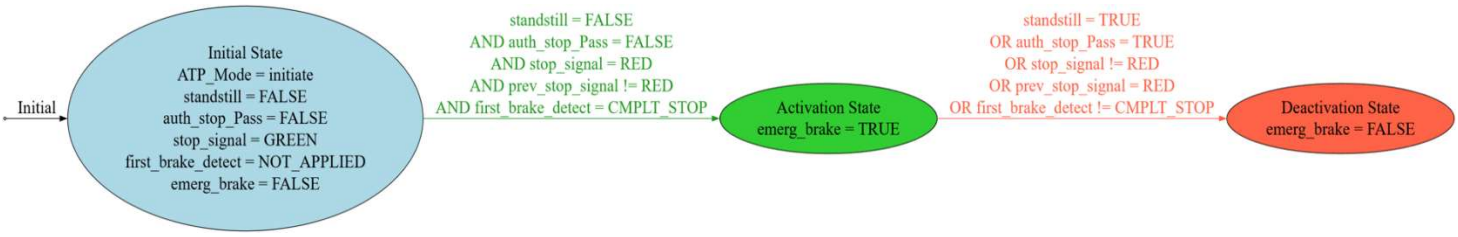


Figure 5.3: Block Diagram for Requirement APP-002

As with the previous requirement, an ".smv" file was developed in Notepad++ to model this requirement.

After interpreting the requirement, a model was created to represent the behavior of a system that determines when to activate the emergency brake based on specific conditions. In this model, variables are first defined to describe the system's state, including the current and previous status of the stop signal, a variable indicating whether the vehicle is stationary, authorization to pass the signal, the status of the first braking, and whether the emergency brake is activated.

In the initialization phase, it was established that the vehicle is in motion, does not have authorization to pass the signal, the signal is green, no braking has occurred, and the emergency brake is deactivated. The emergency brake transition logic only occurs if all conditions are met, namely: the vehicle is in motion, there is no authorization to pass the signal, the signal is red, the previous signal state was not red, and a complete braking has been detected. If any of these conditions fail, the emergency brake remains deactivated. If neither of the first two rules is satisfied, the system maintains the current state of the brake.

Additionally, properties in LTL were specified for verification.

```
MODULE main

--Varibale declarations
VAR
--possible states for the previous signal stop
prev_stop_signal: {RED, YELLOW, GREEN};

--Authorized Stop Passage status
uth_stop_Pass: boolean;

--standstill status
standstill: boolean;

--possible states for stop signal
stop_signal: {RED, YELLOW, GREEN};
```

```
        --possible states for the first brake detection
        first_brake_detect: {CMPLT_STOP, PARC_STOP, NOT_APPLIED};

        --emergency brake status
        emerg_brake: boolean;

        ASSIGN
                --initializations
                init(standstill) := FALSE;
                init(auth_stop_Pass) := FALSE;
                init(stop_signal) := GREEN;
                init(first_brake_detect) := NOT_APPLIED ;
                init(emerg_brake) := FALSE;
                init(prev_stop_signal) := GREEN;


        next(emerg_brake) := case
                (standstill = FALSE & auth_stop_Pass = FALSE & stop_signal = RED &
                prev_stop_signal != RED & first_brake_detect = CMPLT_STOP) : TRUE;

                (standstill = TRUE | auth_stop_Pass = TRUE | stop_signal != RED |
                prev_stop_signal = RED | first_brake_detect != CMPLT_STOP) : FALSE;
                TRUE : emerg_brake;
                esac;
```

```
-------------------------------- Properties---------------------------------
```

LTLSPEC

```
G ((!standstill & !auth_stop_Pass & stop_signal = RED & prev_stop_signal != RED &
first_brake_detect = CMPLT_STOP) -> (next(emerg_brake) = TRUE));
```

The LTL property verifies whether the system correctly activates the emergency brake as soon as all critical conditions are met. This property states that if the vehicle is not stopped, does not have authorization to pass the red signal, the current signal is red, the signal has changed state and the first detected braking results in a complete stop, then in the next state, the emergency brake must be activated. This verification ensures that the system responds safely and automatically to a possible signal violation, making it a safety verification.

LTLSPEC

```
G (((standstill = TRUE) | (auth_stop_Pass = TRUE) | (stop_signal != RED) |
(prev_stop_signal = RED) | (first_brake_detect != CMPLT_STOP)) ->
(next(emerg_brake) = FALSE));
```

This property ensures that the emergency brake will only be activated when all conditions are fully met. Otherwise, it will remain deactivated, guaranteeing that it will

not be triggered in unexpected situations. In this way, the system maintains safety and prevents unintended activations.

```
LTLSPEC
G F TRUE;
```

Finally, the last property verifies that the model does not enter a deadlock state, ensuring continuous and reliable operation.

As shown in figure 5.4, the model meets expectations, ensuring that the system operates in accordance with the specified requirements. Additionally, the analysis confirms that the model does not exhibit deadlocks, guaranteeing the continuous existence of state transitions and ensuring smooth system functionality.

```
-- specification  G ( F TRUE)  is true
-- specification  G (((((!standstill & !auth_stop_Pass) & stop_signal = RED) & prev_stop_signal != RED) & first
_brake_detect = CMPLT_STOP) -> next(emerg_brake) = TRUE)  is true
-- specification  G (((((standstill = TRUE | auth_stop_Pass = TRUE) | stop_signal != RED) | prev_stop_signal =
RED) | first_brake_detect != CMPLT_STOP) -> next(emerg_brake) = FALSE)  is true
```

Figure 5.4: Requirement verification results for APP-002

Requirement APP-003:

This requirement states that the system must activate the speed restriction reset button whenever the speed restriction is active, the distance traveled minus the location where the restriction was activated exceeds 100 meters, and at the moment, there is no supervision of a semaphore. If any of these conditions are not met, the reset button must not be activated.

This expected behavior can be verified in the figure 5.5.

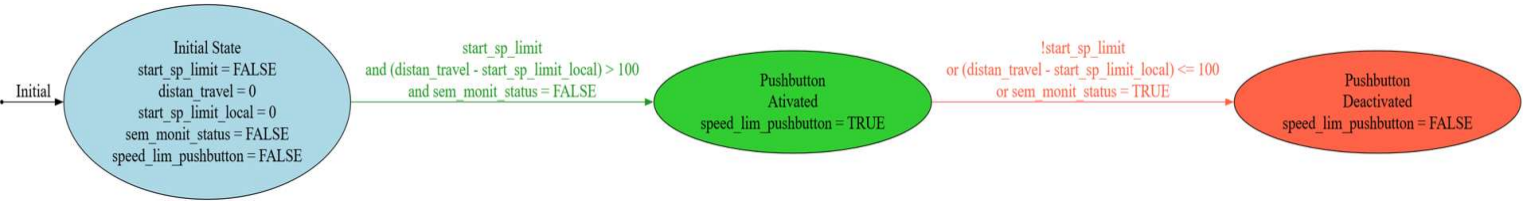


Figure 5.5: Block Diagram for APP-003

After analyzing the requirement, a model was developed to represent the behavior of a system responsible for determining when the speed restriction reset button should be activated.

This model defines variables that describe the system state, such as the speed limit status at the start of the vehicle's operation (start_sp_limit), the distance traveled by the vehicle (distan_travel), the location where the restriction was activated (start_sp_limit_local), the semaphore monitoring status (sem_monit_status) and the state of the speed limit reset button (speed_lim_pushbutton).

During the initialization phase, the system assumes that the speed restriction is inactive, the distance traveled is zero, the activation location is also zero, there is no semaphore to be monitored, and the reset button is deactivated.

The transition logic for activating the reset button only occurs if all conditions are met, where the speed limit status at the start of the vehicle's operation is active, the difference between the distance traveled and the location where the restriction was activated is greater than 100 meters, and there is no primary semaphore to be monitored. When these conditions are simultaneously true, the system activates the button. Otherwise, the button maintains its current state without being automatically deactivated.

In this case, since the calculation of the difference between the distance travelled and the location where the restriction was activated is straightforward and easily interpreted by the model, it is performed directly within the condition evaluation. However, for more complex computations, the use of auxiliary variables in combination with the “DEFINE” keyword would be recommended. This approach not only enhances clarity and maintainability but also aligns with best practices in formal modeling by promoting modular and readable specifications.

To validate the model's behavior, properties were specified in LTL. The main property states that whenever the mentioned conditions are met, the reset button will be activated in the next transition. Another property was specified to ensure that the button is not activated if any of the conditions are not met. Additionally, a property was included to ensure that the system does not enter a deadlock, meaning it is always possible to reach some future state.

This model ensures that button activation only occurs in safe scenarios, in accordance with the expected behavior defined in the requirement.

```
--speed limit reset status pushbutton
MODULE main
--variable declarations
VAR
--start speed limit status
start_sp_limit: boolean;
--distance traveled (m)
distan_travel : 0..300;
--locationn where "start speed limit" was activated (m)
start_sp_limit_local : 0..1;
--signal monitoring status
sem_monit_status : boolean;
--speed limit reset pushbutton status
speed_lim_pushbutton : boolean;

       ASSIGN
       --initialization
       init(start_sp_limit) := FALSE;
       init(distan_travel) := 0;
       init(start_sp_limit_local) := 0;
       init(sem_monit_status) := FALSE;
       init(speed_lim_pushbutton) := FALSE;

       --distance traveled simulation
       next(distan_travel) := case
                  distan_travel < 300 : distan_travel + 1;
                  TRUE : distan_travel;
                  esac;


       --pushbutton state transition
       next(speed_lim_pushbutton) := case
              (start_sp_limit = TRUE & (distan_travel -
              start_sp_limit_local) > 100 & sem_monit_status = FALSE) :
              TRUE;
              (start_sp_limit = FALSE | (distan_travel -
              start_sp_limit_local) <= 100 | sem_monit_status = TRUE) :
              FALSE;
       TRUE : speed_lim_pushbutton;
       esac;

---------------------------------Properties-----------------------------------

LTLSPEC

G((start_sp_limit = TRUE & (distan_travel - start_sp_limit_local) > 100 &
sem_monit_status = FALSE)->(next(speed_lim_pushbutton) = TRUE));
```

This property ensures that throughout the system's execution, whenever the conditions are true, the speed limit reset button will be activated in the next state transition.

The keyword “G” indicates that this verification is globally valid, meaning it applies at all possible moments during the system's execution. The use of the next function

determines that the button activation must occur in the state transition immediately following the moment when the conditions are met.

This property formalizes the expected behavior according to the modeled requirement. It ensures that the implemented model is correct and that the button will be activated exclusively in scenarios considered safe.

```
LTLSPEC

G((start_sp_limit = FALSE | (distan_travel - start_sp_limit_local) <= 100 |
sem_monit_status = TRUE)->(next(speed_lim_pushbutton) = FALSE));
```

Unlike the previous properties, this property ensures that whenever any condition is not met, the reset button will remain inactive.

Similarly, the keyword G, which represents global verification, ensures that this rule remains valid at all times during system operation. The use of the next function indicates that the button activation variable's value will be evaluated in the state transition immediately following the moment when any of these conditions is true.

```
LTLSPEC

G F TRUE;
```

As in previous models, this property ensures that the system never reaches a deadlock state, guaranteeing that execution can always progress and the system remains responsive and reliable.

As illustrated in figure 5.6, the model performs as expected, accurately implementing the activation logic based on the distance traveled, the initial status of the speed restriction, and the presence of semaphore monitoring. The formal analysis confirms that all specified properties are satisfied, ensuring the proper activation and non-activation of the pushbutton under valid and invalid conditions, respectively.

Additionally, the verification guarantees that the model remains free from deadlocks, ensuring the continuous progression of state transitions and the reliable operation of the system.

```
-- specification  G ( F TRUE)  is true
-- specification  G (((start_sp_limit = TRUE & distan_travel - start_sp_limit_local > 100) & sem_monit_status =
FALSE) -> next(speed_lim_pushbutton) = TRUE)  is true
-- specification  G (((start_sp_limit = FALSE | distan_travel - start_sp_limit_local <= 100) | sem_monit_status
= TRUE) -> next(speed_lim_pushbutton) = FALSE)  is true
```

Figure 5.6: Requirement verification results for APP-003

Requirement APP-004:

This requirement refers to the speed supervision logic of the ATP system, specifically the monitoring of the speed of the emergency braking speed curve. It defines that whenever the ATP system mode is set to 'Standard Mode' or 'Beacon Error State,' when the sum of the desired speed and additional speed margin is greater than or equal to the speed of the emergency braking speed curve, then the system must set the value of the emergency braking speed curve to the sum of the desired speed and additional speed margin.

A formula was developed to simulate an emergency braking speed curve. The reference curve for comparison is calculated using this formula:

$$ebc = \left(2 * brake_{capacity} * \sqrt{\frac{4 * des_{speed}{}^{2} + target_{distance} * brake_{capacity}}{brake_{capacity}}} \right) * 3.6$$

The block diagram, in the figure 5.7, visually represents the expected behavior of the requirements in question.

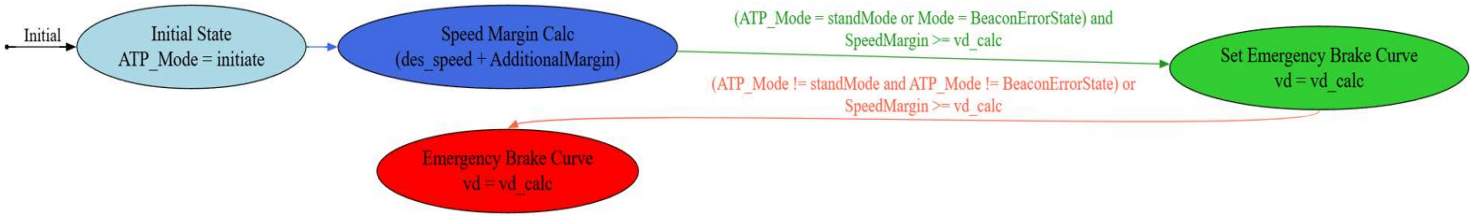


Figure 5.7: Block Diagram for APP-004

After analyzing the requirement, a formal model was developed in NuSMV to represent the behavior of the module responsible for defining the emergency braking speed curve to be monitored by the system. This behavior is conditioned by the system's operating mode and the comparison between a speed margin and the calculated emergency braking speed curve.

The model defines variables that represent both inputs and outputs of the system, such as the current ATP mode (ATP_Mode), braking capacity (brake_capacity), distance to the target (Target_Distance), desired speed (des_speed), additional speed margin (AdditionalMargin), and the output value corresponding to the emergency speed curve (ebc). During the initialization phase, the system assumes the 'initiate' mode, and the output variable 'ebc' is initialized with a symbolic value.

For the calculation of the emergency braking speed curve, the equation was interpreted and adapted to the format accepted by NuSMV, with discretization of values and the use of integer approximation of the square root. Since the NuSMV language does not support floating-point arithmetic or mathematical functions like the square root natively, it was necessary to use the technique of value range tabulation to approximate sqrt, making it possible to obtain an approximate value for the calculation of the emergency braking speed curve.

Initially, with the aim of verifying all possible values of the square root (delta) within the range defined by the input variables, a Python script was developed to generate a table with these values. However, this approach proved to be ineffective, as the combination of inputs was excessively large, leading NuSMV to suffer a state explosion.

Additionally, the “DEFINE” keyword was used to compute auxiliary expressions, such as the speed margin (SpeedMargin) and the intermediate value “delta”, improving the readability and modularity of the model.

As mentioned, NuSMV does not support floating-point numbers, so all calculations were executed in multiples of five to obtain an approximate integer value. Thus, for example, if the final calculated speed is 250 km/h, given that we are working with multiples of five, the actual speed is not 250 km/h but rather 50 km/h.

The value of the 'ebc' variable is only adjusted to the calculated speed margin (SpeedMargin) when the system mode is in “standMode” or “BeaconErrorState”, and when the speed margin is greater than or equal to the calculated emergency braking speed curve (ebc_calc). If these conditions are not met, the “ebc” value is adjusted to the exact value of the braking curve.

To validate the model's behavior, formal properties were specified in CTL.

This model ensures that the logic for assigning the emergency braking speed curve meets the expected behavior described in the requirement, guaranteeing that value adjustments are only made when the system is in safe and appropriate conditions for such an operation.

```
MODULE main
VAR
        -- ATP states
        ATP_Mode: {initiate, Shunting, BeaconErrorState, standMode};

        -- Input: brake capacity (m/s²)
        brake_capacity : 1..10;

        -- Input: Target Distance (m)
        Target_Distance : 1..200;

        -- Input: Desired Speed (km/h)
```

```
        des_speed : 1..100;

        -- Additional speed margin (km/h)
        AdditionalMargin : 0..10;

        -- Output: Calculated emergency brake speed curve (in tenths of 5
        km/h)
        ebc : 0..40000;


DEFINE
-- Calculated tolerance on a scale of 5 to obtain whole numbers
SpeedMargin := (des_speed + AdditionalMargin) * 5;

-- Delta calculation, multiplied by 5 to maintain the scale
delta := (4 * 5 * des_speed * des_speed + Target_Distance * brake_capacity
* 5) / brake_capacity;


 -- Integer approximation of the square root of delta
 sqrt_delta :=
   case
      delta < 1 : 0;
      delta < 4 : 1;
      delta < 9 : 2;
      delta < 16 : 3;
      delta < 25 : 4;
      delta < 36 : 5;
      delta < 49 : 6;
      delta < 64 : 7;
      delta < 81 : 8;
      delta < 100 : 9;
      delta < 121 : 10;
      delta < 144 : 11;
      delta < 169 : 12;
      delta < 196 : 13;
      delta < 225 : 14;
      delta < 256 : 15;
      delta < 289 : 16;
      delta < 324 : 17;
      delta < 361 : 18;
      delta < 400 : 19;
      delta < 441 : 20;
      delta < 484 : 21;
      delta < 529 : 22;
      delta < 576 : 23;
      delta < 625 : 24;
      delta < 676 : 25;
      delta < 729 : 26;
      delta < 784 : 27;
      delta < 841 : 28;
      delta < 900 : 29;
      delta < 961 : 30;
      delta < 1024 : 31;
      TRUE : 32; -- for larger values
```

```
        esac;


    -- Final calculation of the braking curve, scale x5
    ebc_calc := 2 * brake_capacity * (sqrt_delta) * 18; -- 3.6 * 5 = 18

    ASSIGN
            -- Initializations
            init(ebc) := 1;
            init(ATP_Mode) := standMode;

    next(ebc) := case
            (ATP_Mode = standMode | ATP_Mode = BeaconErrorState) & SpeedMargin
            >= ebc_calc : SpeedMargin;
            (ATP_Mode != standMode & ATP_Mode != BeaconErrorState) | SpeedMargin
            < ebc_calc : ebc_calc;
            TRUE : ebc;
            esac;
```

```
--------------------------------Properties-----------------------------------
```

SPEC

AG(((ATP_Mode = standMode | ATP_Mode = BeaconErrorState) & SpeedMargin >= ebc_calc) -> (ebc = SpeedMargin));

This property ensures that, in all possible paths and states of the system, whenever the ATP is in “standMode” or “BeaconErrorState” and the speed margin is greater than or equal to the emergency braking speed curve, the value of emergency brake speed curve will be adjusted to be equal to the margin. This guarantees that the ATP system behaves correctly in safe scenarios, adjusting the emergency braking speed curve as defined by the requirement.

SPEC

EF(((ATP_Mode != standMode & ATP_Mode != BeaconErrorState) | SpeedMargin < ebc_calc) -> (ebc = ebc_calc));

The property ensures that there is at least one path in which, if the system does not meet the requirement conditions, the value of the emergency brake speed curve will be correctly adjusted to the emergency braking curve.

SPEC

AG EX TRUE;

This property in CTL verifies the absence of deadlock in the model, ensuring that in all possible states along all execution paths of the system, or AG, there is always at least

one possible transition to some next state (EX TRUE). In other words, there is no state from which the system cannot continue its execution.

This verification is important because it guarantees that the model will never be stuck in any state, which could indicate modeling errors, logical failures, or undesirable situations in the execution of the real system.

The results of the verification of the described CTL specifications can be observed in figure 5.8, which demonstrates that the model complies with the requirement. All properties have been satisfied, indicating that the system behaves as expected in the different defined scenarios.

```
-- specification AG (EX TRUE)  is true
-- specification AG (((ATP_Mode = standMode | ATP_Mode = BeaconErrorState) & SpeedMargin >= ebc_calc) -> ebc =
SpeedMargin)  is true
-- specification EF (((ATP_Mode != standMode & ATP_Mode != BeaconErrorState) | SpeedMargin < ebc_calc) -> ebc =
 ebc_calc)  is true
```

Figure 5.8: Requirements verification results in NuSMV for APP-004

Requirement SUBSYS-001:

This subsystem requirement describes a critical safety situation within the ATP system, in which the emergency brake must be automatically activated. The requirement logic establishes that if the signal is red, the train's speed is equal to or greater than the speed limit defined for the zone, and if the train is still in motion, then the system must consider this situation as dangerous.

As a response, the system must immediately activate the emergency brake by setting the corresponding variable to true. This measure aims to prevent the train from passing a red signal at high speed, which could result in a collision or another serious accident.

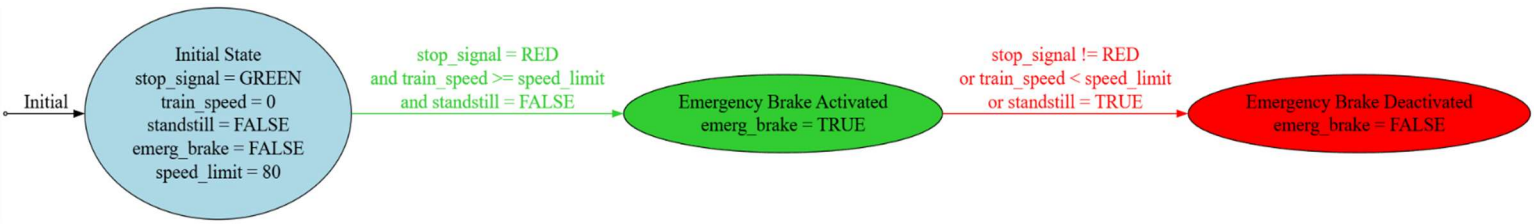


Figure 5.9: Block diagram for subsystem requirement SUBSYS-001

To better understand and prepare for the formal modeling of this behavior, a block diagram was previously created (Figure 5.9), representing the key components involved: signal state, train speed, stopping condition, and emergency brake activation. This diagram serves as the foundation for the modelling that will follow. The goal is to ensure, through formal verification, that this safety behavior is correctly implemented and validated in the system.

Based on the analyzed requirement, a formal model was developed in NuSMV to represent the behavior of the module responsible for activating the emergency brake. The model simulates the decision logic that determines when the system should automatically trigger the emergency brake based on the traffic signal status, the train's speed, the permitted speed limit of the track and the train's stop condition.

The model defines variables representing both system inputs and outputs. The inputs include the traffic signal status (*stop_signal*), which can take the values RED, YELLOW, or GREEN; the immobilization status of the train (*standstill*), indicating whether the train is stopped or moving, the current speed of the train (*train_speed*) and the permitted speed limit of the track (*speed_limit*). The main output modeled is the emergency brake status (*emerg_brake*), which can be activated (TRUE) or deactivated (FALSE).

During the model initialization phase, it is assumed that the train is not stopped (*standstill* = *FALSE*), the signal is green (*stop_signal* = *GREEN*), the initial speed is zero, and the emergency brake is deactivated. The evolution of the train's speed is simulated with a simple logic of incremental increase up to a maximum of 120 km/h.

The activation of the emergency brake is conditioned by a logical expression that directly reflects the system requirement: if the traffic signal is red, the train speed is greater than or equal to the speed limit and the train is not at standstill, then the emergency brake must be activated. This rule is implemented in the transition definition of the "*emerg_brake"* variable through the *case* structure. In any other situation, the emergency brake is not activated.

To validate the model's behavior, formal properties were specified using LTL logic.

This model allows for the formal verification of the expected safety behavior in cases of passing a red signal, ensuring that the system only automatically reacts by applying the emergency brake in well-defined critical situations. Such an approach contributes to increasing confidence in the safe behavior of the automatic train control system.

```
MODULE main
--declarations
VAR
--standstill status
standstill: boolean;

--possible states for Stop signal
stop_signal: {RED, YELLOW, GREEN};

--train speed
train_speed: 0..120;

--speed limit
speed_limit: 0..120;

--emergency brake status
emerg_brake: boolean;


ASSIGN
--initializations
init(standstill) := FALSE;
init(stop_signal) := GREEN;
init(train_speed) := 0 ;
init(emerg_brake) := FALSE;
init(speed_limit) := 80; --in km/h

next(train_speed) := case
       train_speed < 120 : train_speed + 1;
       TRUE : train_speed;
       esac;

next(emerg_brake) := case
       (stop_signal = RED & train_speed >= speed_limit & standstill =
       FALSE) : TRUE;
       (stop_signal != RED | train_speed < speed_limit & standstill !=
       FALSE) : FALSE;
       TRUE : emerg_brake;
       esac;


------------------------------ Properties----------------------------------

LTLSPEC

G ((stop_signal = RED & train_speed >= speed_limit & standstill = FALSE) ->
(next(emerg_brake) = TRUE));
```

This is an LTL property used to verify behaviors over time. The operator "G" it means that the described condition must hold at all moments during the system's execution. It ensures that whenever the traffic signal is red, the train is in motion and its speed is equal to or greater than the speed limit, the system must necessarily activate the

emergency brake in the next execution cycle. This logic is designed to ensure the safe behavior of the modeled system.

```
LTLSPEC

G ((stop_signal != RED | train_speed < speed_limit & standstill = TRUE) ->
(next(emerg_brake) = FALSE));
```

This property ensures that whenever a condition imposed by the requirement is not met, the system should not activate the emergency brake in the next execution cycle.

This rule helps prevent false positives in brake activation, ensuring that it is only used in truly critical conditions.

```
LTLSPEC

G F TRUE;
```

This property verifies that at all times during execution, it will always be possible to advance to some future state, meaning the model does not enter a deadlock.

If it is satisfied, it means the system will always have a possible transition and will continue evolving over time.

The fact that all properties are satisfied confirms that the system correctly meets the requirement SUBSYS-001. Figure 5.10, which represents the results obtained in NuSMV, reinforces the findings verified in the table.

```
-- specification  G ( F TRUE)  is true
-- specification  G (((stop_signal = RED & train_speed >= speed_limit) & standstill = FALSE) -> next(emerg_brake) = TRUE)
 is true
-- specification  G ((stop_signal != RED | (train_speed < speed_limit & standstill = TRUE)) -> next(emerg_brake) = FALSE)
 is true
```

Figure 5.10: Requirement verification results for requirement SUBSYS-001

Requirement SYS-001:

This system requirement describes a safety operational condition within the ATP system, related to the allowed duration of the shunting mode. The goal is to prevent the train from remaining in this mode for an excessive distance, which could compromise operational safety.

The logic of the requirement establishes that if the vehicle is operating in shunting mode and the distance traveled with this mode active is greater than or equal to 500 meters, the ATP system must indicate that the shunting mode should be deactivated. This means the system will recognize that the prolonged use of this mode has exceeded the permitted limit.

Thus, we can ensure that the shunting mode, which is generally restricted to low-speed operations and specific areas, is not used in inappropriate situations, such as long-distance routes.

To aid understanding and future formal modeling of this behavior, a state diagram (Figure 5.11) has been previously created, representing the key elements involved.

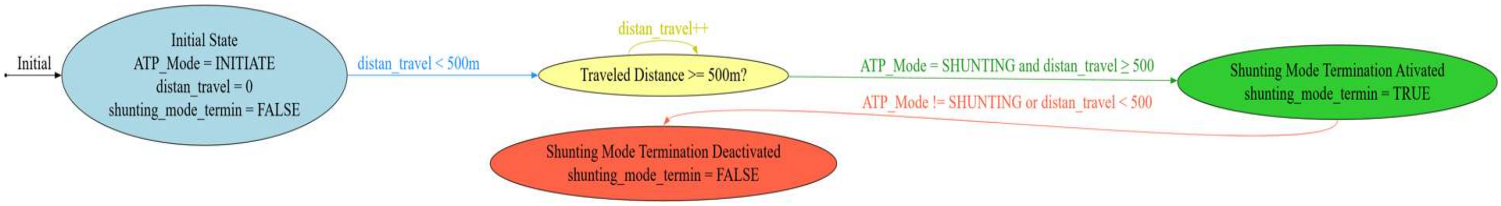


Figure 5.11: Block diagram for system requirement SYS-001

Based on the analyzed requirement, a formal model in NuSMV was developed to represent the expected behavior for the deactivation of the shunting mode.

This model defines three main variables: the ATP operating mode, the distance traveled by train and the indication for ending the shunting mode. The “ATP_Mode” variable represents the current state of the operating mode and can take the values “INITIATE”, “SHUNTING”, “FORWARD” and “REVERSE”. The “distan_travel” variable represents the total distance traveled during operation, ranging from 0 to 550 meters. The boolean variable “shunting_mode_termin” represents the logical output that signals whether the system recognizes that the distance limit in shunting mode has been reached, thus requiring its termination.

During the model initialization phase, the system is assumed to start in the “INITIATE” mode, with the distance traveled set to zero and the shunting mode termination indication deactivated. The distance progression is simulated through a simple logic

that increases the value of the “distan_travel” variable at each cycle, up to a limit of 510 meters, representing the train's movement over time.

The activation of the shunting mode termination variable is based on a logical expression that directly reflects the system requirement. If the system is operating in shunting mode and the distance traveled is greater than or equal to 500 meters, then the “shunting_mode_termin” variable is activated in the next state. In any other situation, when the ATP mode is different from “SHUNTING” or the distance is less than 500 meters, the variable remains unchanged or is reset to false.

To validate the model's behavior, formal properties were specified in LTL.

This model enables the formal verification of the system's expected behavior in relation to shunting mode control. It ensures that the state transition indicating the end of shunting mode occurs exclusively when the distance condition is met, preventing prolonged operation in a mode that should be restricted to short maneuvers.

```
MODULE main
--variables declaration

VAR
--possible states for ATP
ATP_Mode: {INITIATE, SHUNTING, FORWARD, REVERSE};

--distance travelled (m)
distan_travel : 0..550;

--Status of the variable that activates the end of shunting mode
indication.
shunting_mode_termin : boolean;

ASSIGN
--Initializations
init(shunting_mode_termin) := FALSE;
init(distan_travel) := 0;
init(ATP_Mode) := INITIATE;

--Distance travelled increase
next(distan_travel) := case
       distan_travel < 510 : distan_travel + 1;
       TRUE : distan_travel;
       esac;

--Transition of the state of the indication variable to end the shunting
mode.
next(shunting_mode_termin) := case
       (ATP_Mode = SHUNTING & distan_travel >= 500) : TRUE;
       (ATP_Mode != SHUNTING | distan_travel < 500) : FALSE;
       TRUE : shunting_mode_termin;
       esac;
```

```
-------------------------------Properties----------------------------------
```

LTLSPEC

G((ATP_Mode = SHUNTING & distan_travel >= 500)->(next(shunting_mode_termin) = TRUE));

With the temporal operator G, which means "always," the specified condition must be valid in all possible states of the system at any point in time.

This property ensures that in all possible states, if the train travels 500 meters or more while in shunting mode, the system must mandatorily trigger the end of the shunting mode.

This verification is essential to ensure that the model meets the requirement limiting the allowed distance in shunting mode.

LTLSPEC

G((ATP_Mode != SHUNTING | distan_travel < 500)->(next(shunting_mode_termin) = FALSE));

Unlike the previous property, this one ensures that in all possible states, the indication for ending the shunting mode will not be activated if any of the conditions are not met. This prevents false positives, ensuring that the activation of the end of SHUNTING mode only occurs when all conditions are simultaneously true. It prevents the system from incorrectly triggering the end of the shunting mode prematurely, reinforcing the precision and reliability of the control logic required by the safety requirement.

LTLSPEC

G F TRUE;

This property, serves as a general robustness test for the model to ensure that the modelled system will always continue and never be blocked in a dead-end state.

The fact that all three properties are satisfied (evaluated as TRUE) confirms that the system behavior aligns correctly with the requirement SYS-001.

These results, reinforced by the analysis shown in figure 5.12, provide strong evidence that the formal model accurately and reliably implements the logic described in the requirement.

```
-- specification  G ( F TRUE)  is true
-- specification  G ((ATP_Mode = SHUNTING & distan_travel >= 500) -> next(shunting_mode_termin) = TRUE)  is true
-- specification  G ((ATP_Mode != SHUNTING | distan_travel < 500) -> next(shunting_mode_termin) = FALSE)  is true
```

Figure 5.12: Requirement verification results for requirement SYS-001

5.2 Model of ATP System in UPPAAL

Requirement APP-001:

The modeling of the requirement in UPPAAL differs slightly from that in NuSMV, as UPPAAL uses timed automatons and is structured through states and transitions that represent the model. Timed automaton in UPPAAL can communicate by using channels and shared data structures.

During the modeling of this requirement, two different approaches were adopted. In one case, channels were used to enable communication between templates. In the other approach, a single template was employed, capable of verifying conditions and triggering the desired states.

This demonstrates that the same requirement can be modeled in multiple ways. There is no strictly correct or incorrect modeling approach, what truly matters is achieving the intended final result.

Figure 5.13 presents the project tree in which three templates were created, using channels to communicate with each other. The "ATP" template, shown in figure 5.14, is responsible for verifying whether the conditions are met and synchronizing the "active" channel.

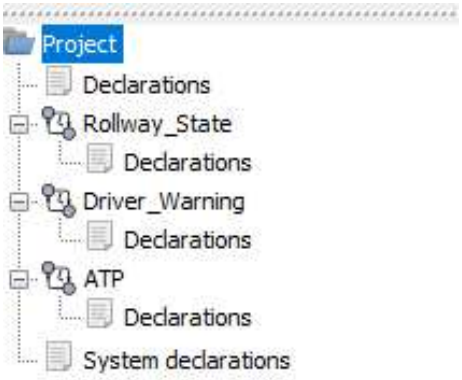


Figure 5.13: UPPAAL navigation tree for APP-001

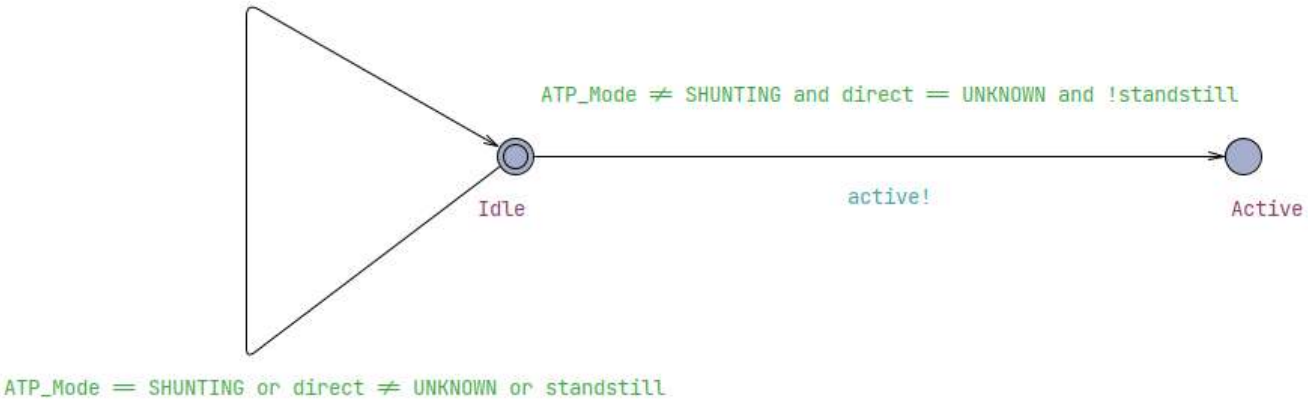


Figure 5.14: Template for ATP model

Once the conditions are met, the "active" channel is synchronized, triggering the transition of the states defined in the "Rollway_State" template (Figure 5.15) and "Driver_Warning" template (Figure 5.16).

However, whenever a condition is met, the model remains in the "Idle" state.

Figure 5.15: Template for Rollway_State

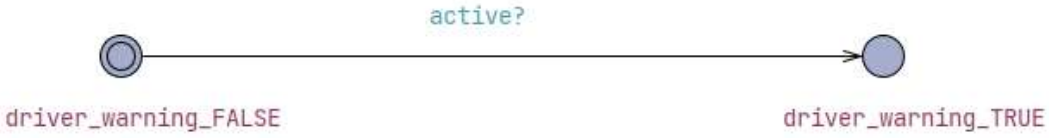


Figure 5.16: Template for driver_warning

Just like in NuSMV, in UPPAAL, all variables used in the model must be declared in the "declarations" section under the “project” folder. This ensures that they can be accessed by all created templates.

```
// Possible modes for ATP MODE
const int INITIATE = 0;
const int SHUNTING = 1;
const int AGAINST = 2;
const int RED_LIGH_CROSSING = 3;

// Possible states for travel direction
const int UNKNOWN = 4;
const int FORWARD = 5;
const int REVERSE = 6;

// standstill status
bool standstill;

// rollway status
bool Rollway_state = false;

// driver_warning status
bool Driver_warning = false;

// channel
chan active;

// initializations
int ATP_Mode = INITIATE;
int direct = FORWARD;
```

To use the "channel" functionality in UPPAAL, it is necessary to declare a variable of type "chan", giving it a name as desired.

Next, we provide a description of the properties verified in UPPAAL for this model:

A[] (ATP_Mode != SHUNTING && direct == UNKNOWN && !standstill imply P2.driver_warning_TRUE and P3.rollway_state_TRUE)

The above property verifies that if the ATP system is not in shunting mode, the desired travel direction is unknown, and the train is not in standstill, then a warning will be triggered to alert the driver and the rollway state shall be set to true. The result turned against this query by UPPAAL verifier is “Property is satisfied”.

A[] not deadlock

The property ensures that the system model is free from deadlocks. If no deadlock is detected, the UPPAAL verifier confirms the property as 'Property is satisfied.'

A[] (ATP_Mode == SHUNTING or direct != UNKNOWN or standstill imply P1.Idle and P2.driver_warning_FALSE and P3.rollway_state_FALSE)

This above property ensures that, in any state of the system, if the ATP system is in shunting mode, the train's direction is different from UNKNOWN or it is in standstill, then the system will remain inactive, no warning will be triggered for the driver and the rollway state will not be activated. The result returned by the UPPAAL verifier is: “Property is satisfied.”

Figure 5.17 provides the CTL-based description of the properties verified in UPPAAL. The green dots indicate that all properties have been satisfied. Figure 5.18 shows the final verification of the results.

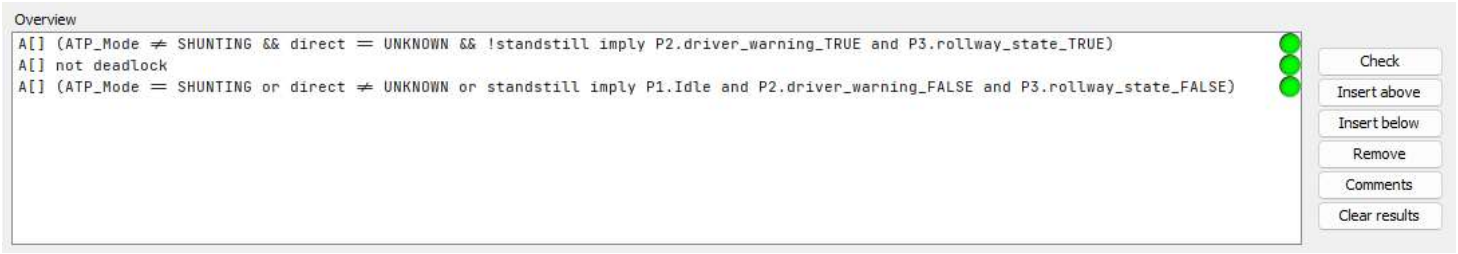


Figure 5.17: Properties in CTL of UPPAAL for APP-001

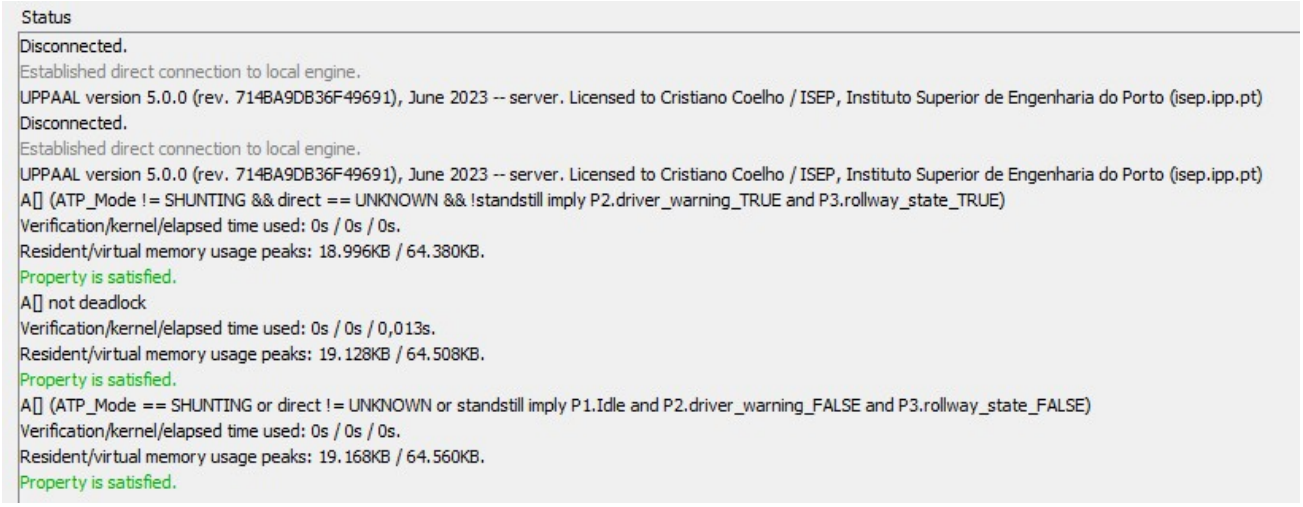


Figure 5.18: Final verification results in UPPAAL for APP-001

Regarding the second approach, as previously mentioned, only a template was used. Similarly to the first approach described above, it was also necessary to declare all the variables used in the model in the "declarations" section, located under the "project" folder.

```
// Possible modes for ATP MODE
const int INITIATE = 0;
const int SHUNTING = 1;
const int AGAINST = 2;
const int RED_LIGH_CROSSING = 3;

// Possible states for travel direction
const int UNKNOWN = 4;
const int FORWARD = 5;
const int REVERSE = 6;

// standstill status
bool standstill;

// rollway status
bool Rollway_state = false;

// driver_warning status
bool Driver_warning = false;

// initializations
int ATP_Mode = INITIATE;
int direct = FORWARD;
```

The main difference between the two approaches is that, in the previous version, the automaton states explicitly represented the transition of the actions expected by the model. In this more simplified version (Figure 5.20), however, the actions do not directly depend on state transitions but are dynamically updated in the "update" section as soon as the predefined conditions are met (Figure 5.19).

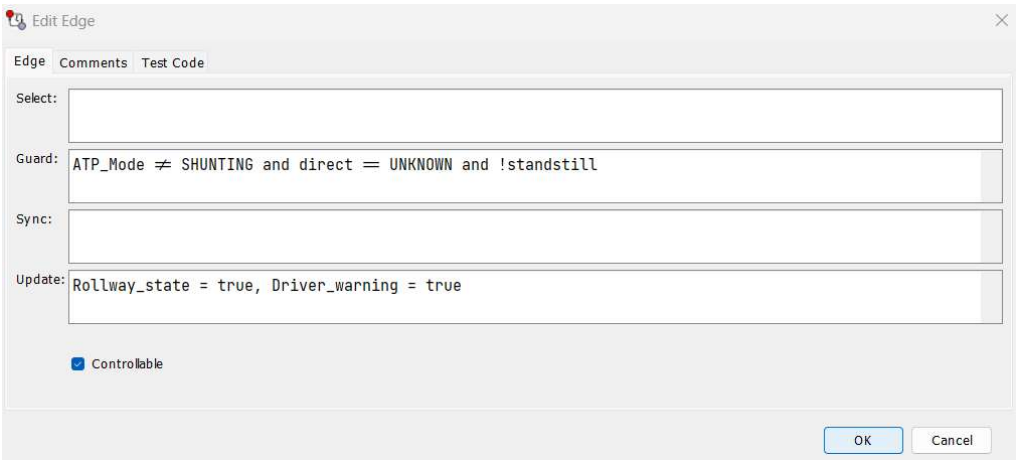


Figure 5.19: “Update” section in “Edit Edge”

This approach makes the model more compact and efficient, eliminating the need for multiple states to represent status changes. Additionally, it allows for more direct control of variables within the system's logic.

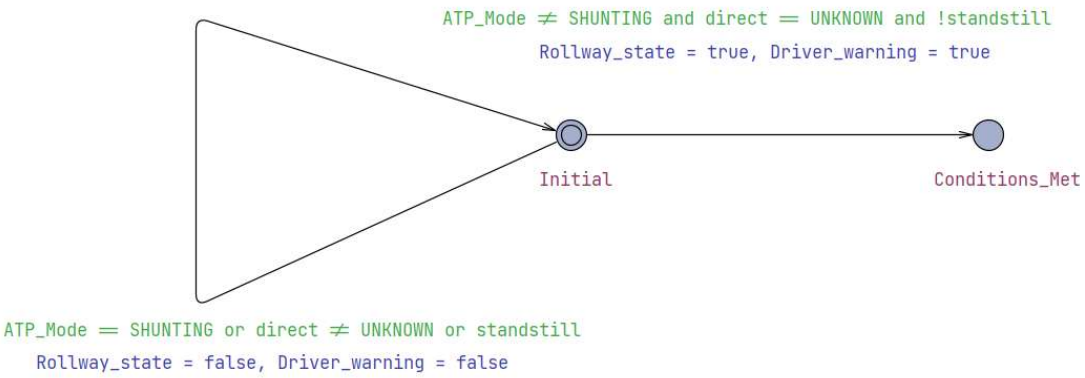


Figure 5.20: Second approach for ATP model

Regarding the properties verified in this model, we now present their description:

```
A[] (ATP_Mode == SHUNTING || direct != UNKNOWN || standstill imply Rollway_state ==
false && Driver_warning == false )
```

Similarly to what happened in the previous approach, this property ensures that in any state of the system, if the ATP system is in shunting mode, the train's direction is different from UNKNOWN or it is in standstill, then the system will remain inactive, no warning will be triggered for the driver, and the rollway state will not be activated. The result returned by the UPPAAL verifier is: “Property is satisfied.”

```
A[] (ATP_Mode != SHUNTING && direct == UNKNOWN && !standstill imply Rollway_state
== true && Driver_warning == true )
```

Similarly to the previous approach, this query verifies that if the ATP system is not in shunting mode, the desired travel direction is unknown, and the train is not in standstill, then a warning will be triggered to alert the driver and the rollway state shall be set to true. The result returned by the UPPAAL verifier is: “Property is satisfied.”

```
A[] not deadlock
```

The query ensures that the system model is free from deadlocks. The UPPAAL verifier confirms the property as 'Property is satisfied'.

Figure 5.21 provides the CTL-based description of the properties verified in UPPAAL for the second approach. All properties have been satisfied. Figure 5.22 shows the final verification of the results.

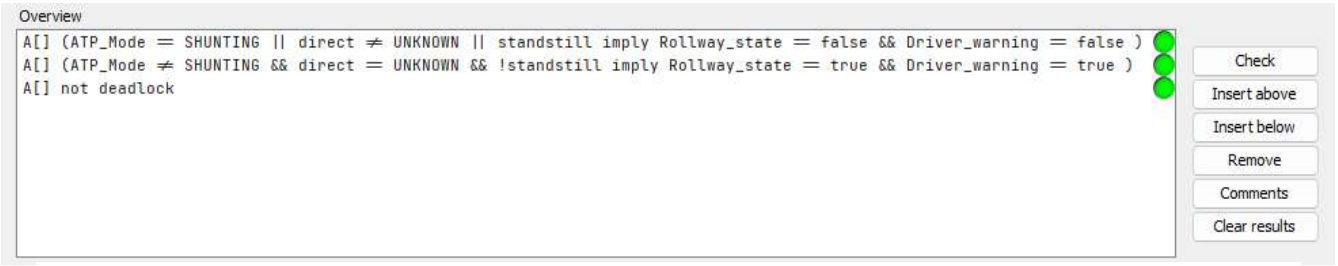


Figure 5.21: Properties in CTL of UPPAAL for second approach of APP-001

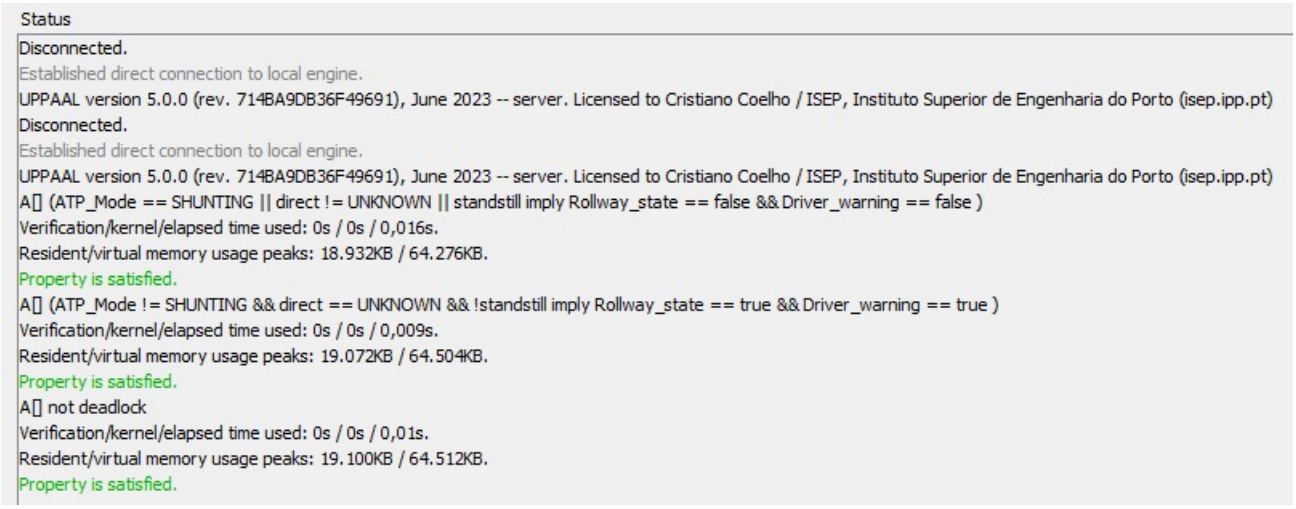


Figure 5.22: Final verification results in UPPAAL for second approach of APP-001

Requirement APP-002:

Figure 5.23 presents the model developed for this requirement, consisting of two main states: 'Initial' and 'Conditions_met'. The 'Conditions_met' state represents the situation in which all necessary conditions have been met, including: the vehicle is not in standstill, it does not have authorization to proceed through a red light, the previous signal was not red and the vehicle has already made a stop. Whenever these conditions are met, the emergency brake (emerg_brake) will be activated, assuming the value 'true'.

In this example, we adopted the approach in which the state transition is updated through the 'update' action, accessible in the 'Edit edge' tab (Figure 5.24). This strategy ensures that the state change occurs accurately and in alignment with the defined criteria.

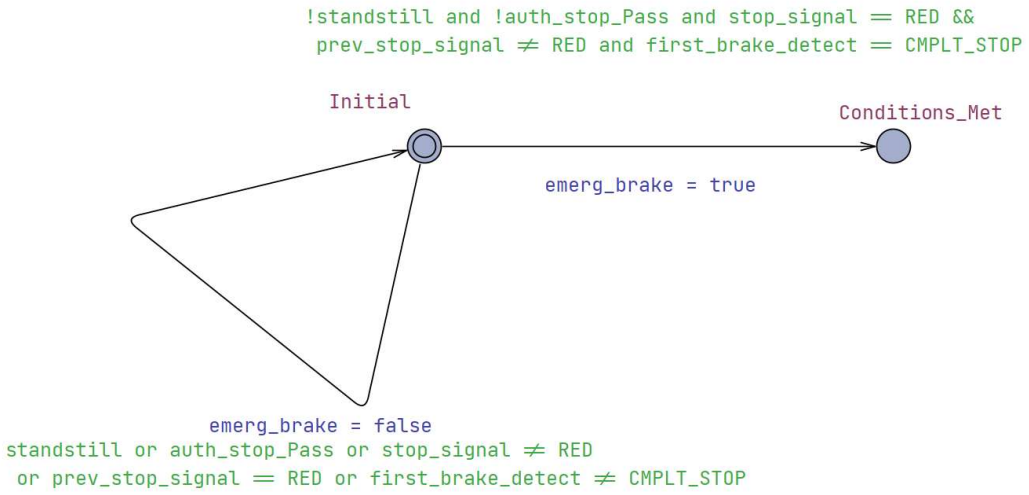


Figure 5.23: Automaton for APP-002

Figure 5.24: “emerg_brake” in update section

Whenever any of the conditions are not met, the emergency brake will be deactivated, assuming the value false and, consequently, will not be triggered.

Below it is possible to visualize all the variables used in the model, declared as *global declarations*.

```
//prev_stop_signal and stop_signal states
const int RED = 0;
const int YELLOW = 1;
const int GREEN = 2;

//first brake detection states
const int CMPLT_STOP = 3;
const int PARC_STOP = 4;
const int NOT_APPLIED = 5;

//Authorized Stop Passage Status
bool auth_stop_Pass;

//standstill status
bool standstill;

//emergency brake status
bool emerg_brake;

int prev_stop_signal = GREEN;

int stop_signal = GREEN;

int first_brake_detect = NOT_APPLIED;
```

To ensure that the model functions as expected, some properties have been specified:

```
A[] (!standstill and !auth_stop_Pass and stop_signal == RED and prev_stop_signal != RED and first_brake_detect == CMPLT_STOP imply emerg_brake == true)
```

Similarly to NuSMV, this CTL specification defines a safety rule that must always be valid in the system. It establishes that if the vehicle is not in standstill, does not have authorization to proceed past a stop signal, the signal is red, the previous signal was not red (indicating a recent change), and the initial braking has already brought the vehicle to a complete stop, then the system must activate the emergency brake.

This rule is formally expressed as an implication that must hold true in all possible states of the system, ensuring a safe response in the event of a potential signal violation.

A[] (standstill or auth_stop_Pass or stop_signal != RED or prev_stop_signal == RED or first_brake_detect != CMPLT_STOP imply emerg_brake == false)

The specification states that, at all times in the system, if the vehicle is stationary, or if there is authorization to proceed past a stop signal, or if the signal is not red, or if the signal was already red previously, or if the initial braking has not yet resulted in a complete stop, then the emergency brake must remain deactivated.

In other words, this formula defines scenarios where there is no critical risk and, therefore, activation of the emergency brake is not necessary. It complements the previous specification by ensuring that the system only reacts with an emergency response when all risk conditions are present.

A[] not deadlock

The specification states that the system must never enter a deadlock state, meaning there must always be a possible transition from any given state. This ensures that the system will not become blocked or frozen, remaining continuously capable of reacting and evolving throughout its execution.

As illustrated in figure 5.25 we can verify that these properties have been satisfied.

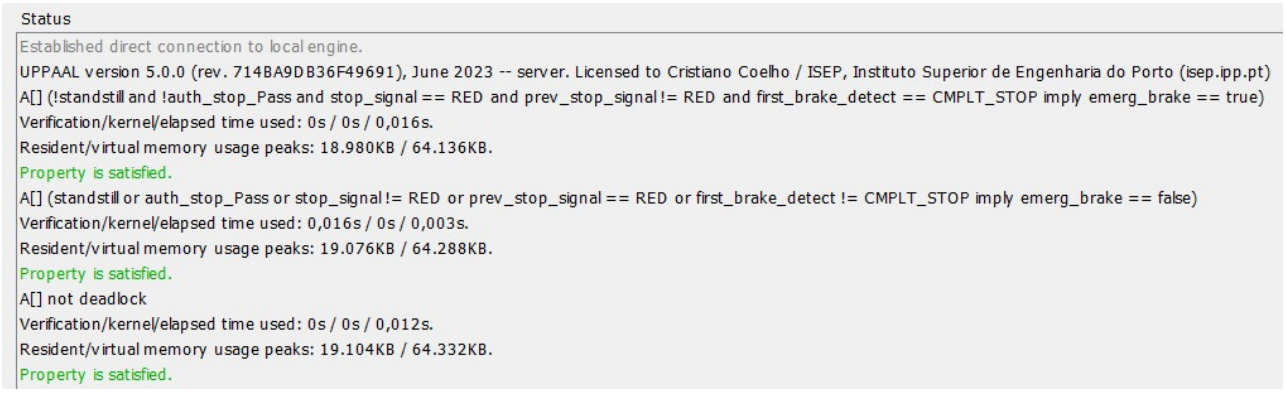


Figure 5.25: Final verification results for APP-002

Requirement APP-003:

The presented model was developed with the objective of formally verifying the system's behavior in relation to the requirement APP-003. This requirement defines the conditions for activating or maintaining the deactivation of the speed limit reset button (speed limit pushbutton).

The defined conditional logic considers three main factors: the current state of the speed limit, the distance traveled from the point where the restriction was activated, which must not exceed 100 meters and, finally, the status of the traffic light monitoring system.

To represent this behavior, the model consists of two main templates: “atp” (Figure 5.26), responsible for modeling the speed limit button states (active and inactive), and “distanceControl” (Figure 5.27), which simulates the progressive increase in the traveled distance and calculates the difference between the distance traveled and the point where the restriction was activated. In the latter, two states have been defined: “increase_distance”, which represents the state in which the speed is incremented, and “Finish_increase”, which marks the end of the speed increment process.

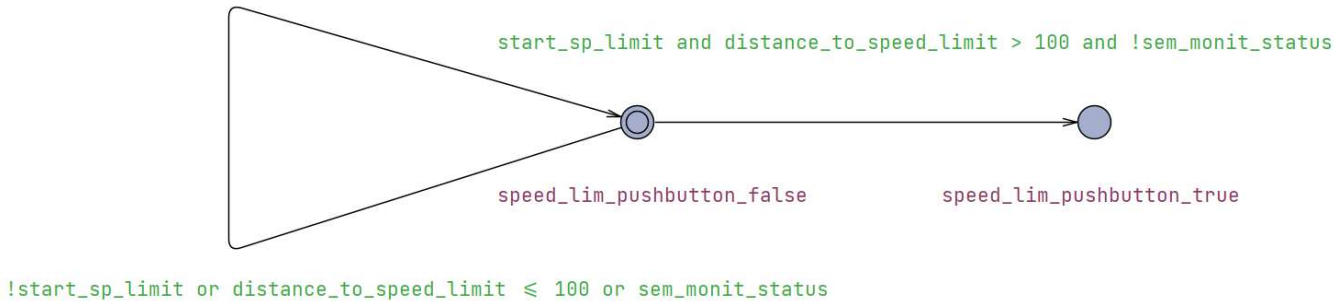


Figure 5.26: “atp” template for button status control

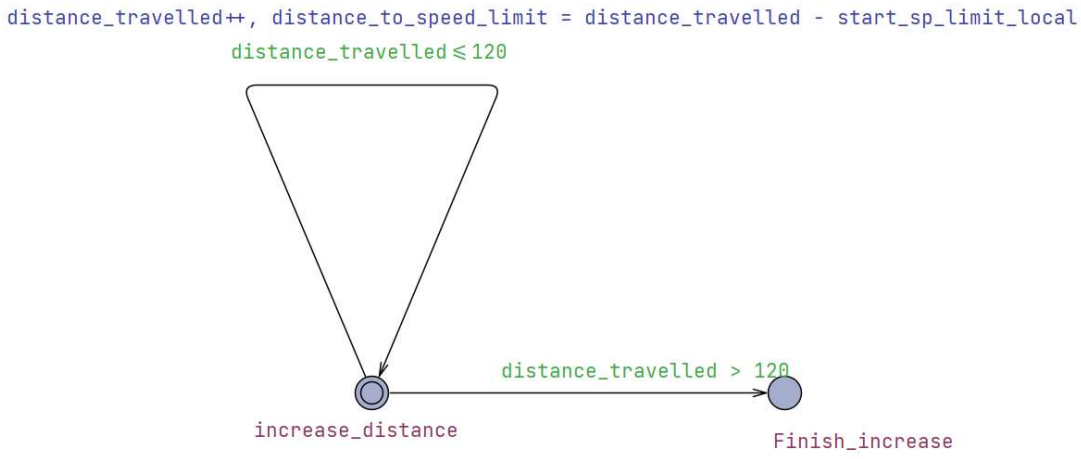


Figure 5.27: “distanceControl” template for distance increase

The transition between button states is regulated by “guards” (Figure 5.28) that directly follow the conditions established by the requirement. The button will be activated only if the restriction is active, the difference between the traveled distance and the activation point exceeds 100 meters and there is no traffic light under monitoring.

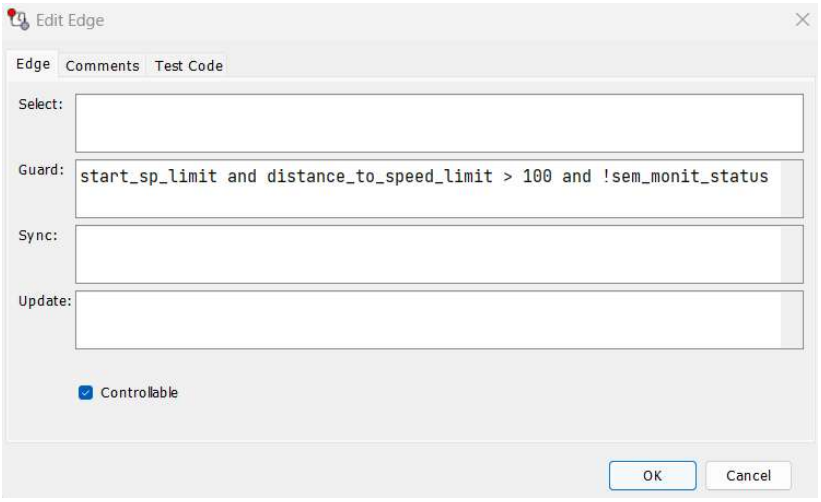


Figure 5.28: “Guard” for button states transition regulation

Additionally, the model includes global variables as we can see below, that represent the restriction state, the traveled distance, the presence or absence of semaphore monitoring, the initial location of the restriction activation and the button's own state.

```
//represents the distance travelled.
int distance_travelled;

//start speed limit status
bool start_sp_limit;

//Semaphore montioring status
bool sem_monit_status;

//location of when the "start speed limit" was active (m)
int start_sp_limit_local = 0;

//auxiliary calc variable
int distance_to_speed_limit;
```

Formal verification is performed through three main queries: one that ensures the correct activation of the button under valid conditions, another that guarantees the button is not activated when any of the conditions are invalid, and a third that verifies the absence of deadlock states in the system.

```
A[] (start_sp_limit && distance_to_speed_limit > 100 && !sem_monit_status imply P2.speed_lim_pushbutton_true)
```

This property verifies whether, in all possible system states, whenever the initial speed limit is active, the difference between the traveled distance and the point where the restriction was activated exceeds 100 meters and the traffic light monitoring is disabled, then the speed limit reset button will be activated. This precisely corresponds to the system's functional requirement and formally ensures that, under these conditions, the expected behavior will be fulfilled.

```
A[] (!start_sp_limit || distance_to_speed_limit <= 100 || sem_monit_status imply P2.speed_lim_pushbutton_false)
```

This property ensures that, in all possible system states, if at least one of the conditions established by the requirements is not met, then the system must not activate the speed limit reset button, meaning it should remain in the “speed_lim_pushbutton_false” state. This confirms that the button will only be activated when all necessary conditions are simultaneously true, preventing unintended activations.

```
A[] not deadlock
```

This property verifies whether the system never enters a deadlock state during its execution. This means that, in all possible system paths, there will always be a possible transition to be executed, ensuring that the system never becomes stuck without available actions. Successfully verifying this property indicates that the model is free from situations where processes could become trapped and unable to proceed, guaranteeing that the system's behavior is always progressive and executable.

All properties have been successfully satisfied, confirming that the requirement was correctly modeled and validated within the system, ensuring the expected behavior in all anticipated scenarios. The obtained results (Figure 5.29) confirm that the model meets the expected behavior defined by the requirement, ensuring both the safe activation and the prevention of unintended activations of the button. Additionally, it guarantees the continuous progression of the system states, maintaining its functionality as expected.

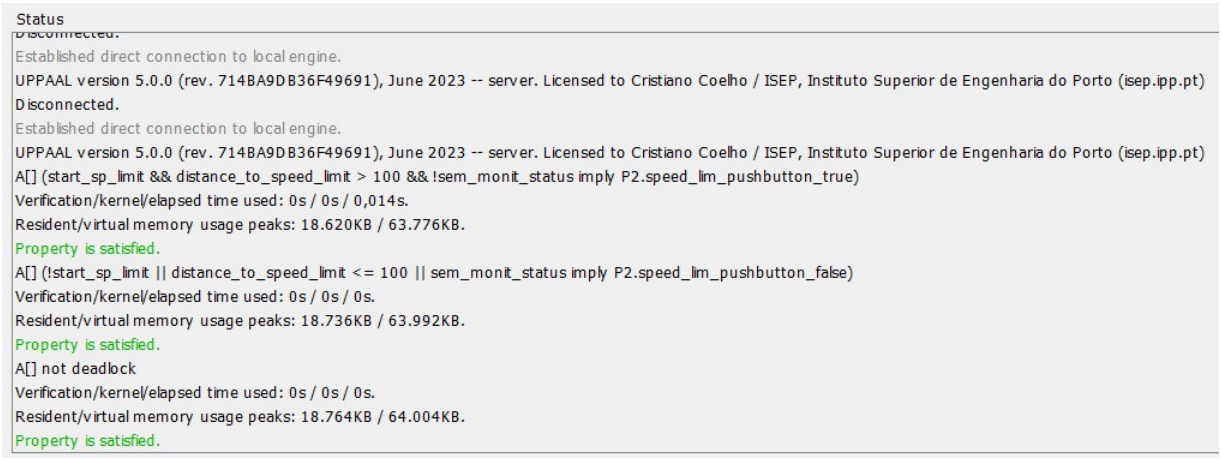


Figure 5.29: Final verification results for APP-003

The expected behavior of the model can be analyzed in the 'Symbolic Simulator,' allowing real-time observation of the speed increase when selecting the 'random' option in “simulation trace” tab (Figure 5.30), while simultaneously tracking state transitions as conditions are met (Figure 5.31).

Additionally, it is possible to visualize a graph of state transitions, making it easier to interpret when they occurred.

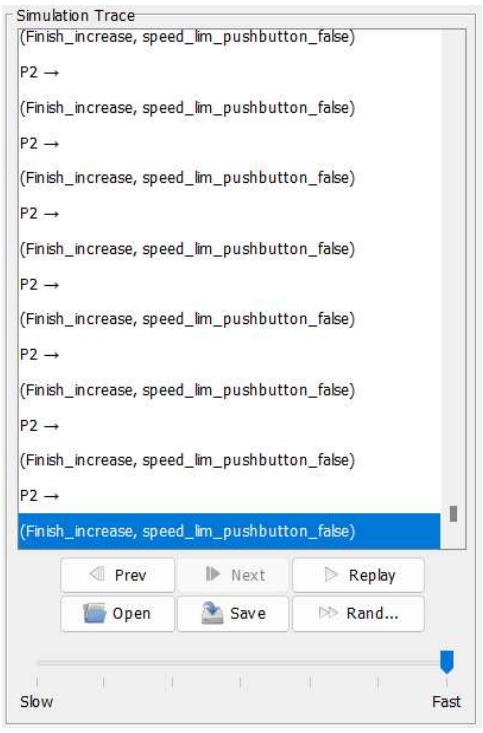


Figure 5.30: Simulation Trace with random option

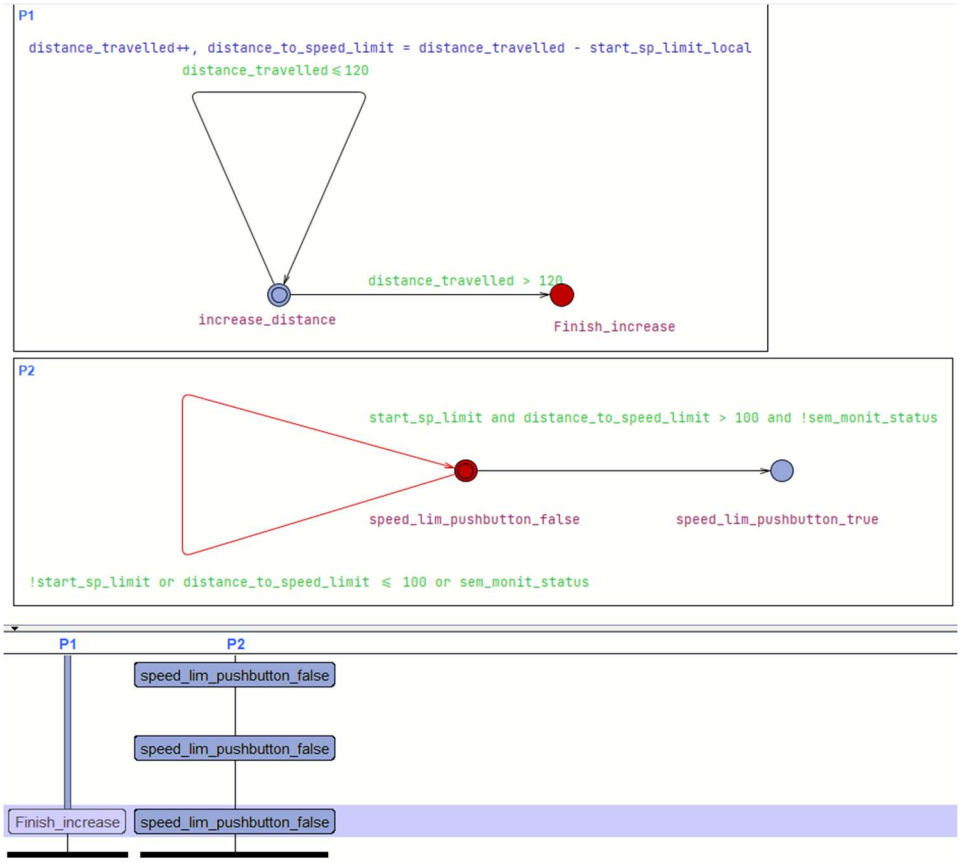


Figure 5.31: State Transition

Requirement APP-004:

The following model presented in UPPAAL describes the behavior already explained in NuSMV, which aims to verify the conditions for updating the emergency brake speed curve.

The main task of the model is to calculate the emergency brake speed curve (calculate_eb_speed) and compare it with the safety margin to determine whether the system should adjust the speed or maintain the calculated value.

At the beginning of the model, some global declarations are made to represent the system parameters, such as the ATP modes (which include INITIATE, SHUNTING, BEACON_ERROR_STATE, and STAND_MODE) and input variables, such as the desired speed, additional safety margin, target distance for braking, and the system's braking capacity.

The model also contains important calculation functions, such as the function “sqrt_to_int”, which approximates the square root of a number by converting the value

into an integer, since UPPAAL does not support the 'double' data type during property verification.

```
int sqrt_to_int(int x) {
    int res = 0;
    while (res * res <= x) {
        res++;
    }
    return res - 1;
}
```

The function “calculate_eb_speed” computes the emergency brake speed curve based on the formula provided in the requirement, considering braking capacity, target distance, and desired speed.

```
int calculate_eb_speed (int brake_capacity, int target_distance, int
desired_speed)
{
        int delta = (4 * 5 * desired_speed * desired_speed + target_distance
        * brake_capacity * 5) / brake_capacity;
        return 2 * brake_capacity * sqrt_to_int(delta)) * 18;//multiply by
        18 to convert speed to km/h (3.6 * 5)
}
```

Meanwhile, the function “calculate_margin” is responsible for calculating the safety margin based on the desired speed and additional margin.

```
int calculate_margin (int desired_speed, int add_speed_margin)
{
    int calc_margin = (desired_speed + add_speed_margin) * 5;
    return calc_margin;
}
```

As was done in NuSMV, all calculations involving decimal values were adapted to work with multiples of five to avoid the use of decimal numbers.

The model consists of a template called “atp” (Figure 5.32) which describes the states and transitions of the ATP system.

When the conditions are verified, the system transitions to the “conditions_met” state, where the variable “eb_speed_result” is updated with the calculated speed margin value. Otherwise, the system returns to the initial state and maintains the previous value of the “eb_speed_result” variable.

To ensure the model functions correctly, three CTL specifications are defined.

This UPPAAL model validates the expected behavior of the system, particularly regarding the verification of conditions for updating the emergency brake speed curve, ensuring that safety conditions are met and that the system does not become blocked during execution.

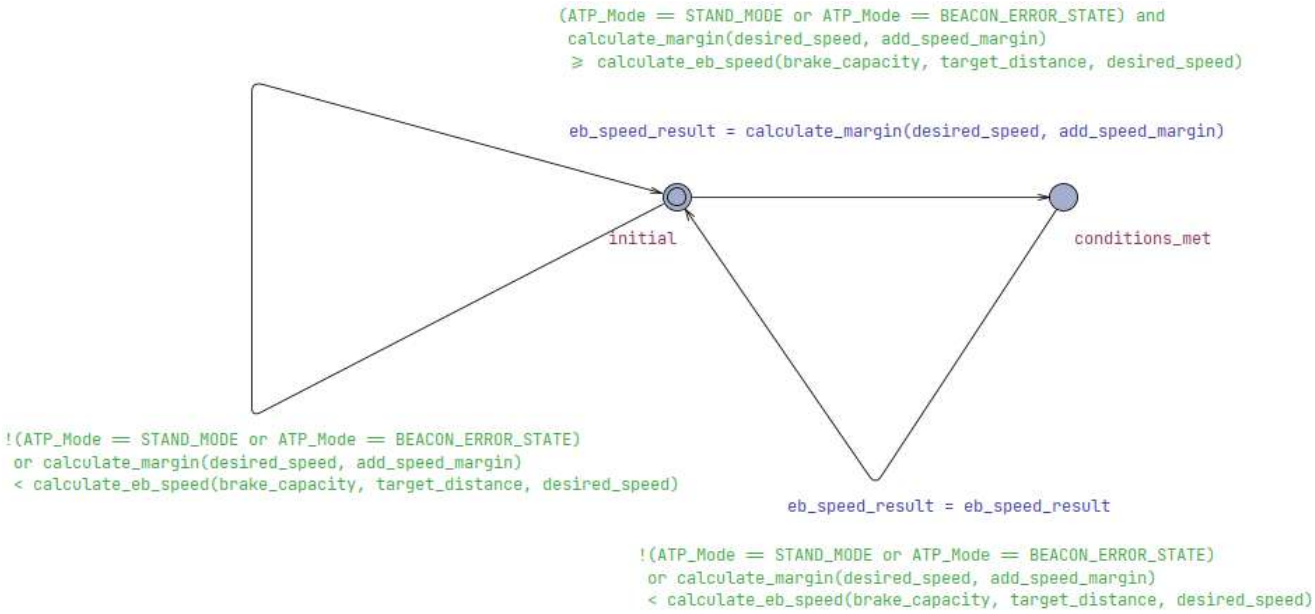


Figure 5.32: ATP template for Requirement APP-004

```
A[] ((ATP_Mode == STAND_MODE || ATP_Mode == BEACON_ERROR_STATE) &&
calculate_margin(desired_speed, add_speed_margin) >=
calculate_eb_speed(brake_capacity, target_distance, desired_speed)) imply
calculate_eb_speed(brake_capacity, target_distance, desired_speed) ==
calculate_margin(desired_speed, add_speed_margin)
```

This property verifies that, in all possible execution paths of the system and at all points in time (A[]), if the ATP mode is “STAND_MODE” or “BEACON_ERROR_STATE” and the calculated speed margin (calculate_margin) is greater than or equal to the emergency brake speed curve (calculate_eb_speed), then the emergency brake speed curve must be equal to the calculated speed margin.

This property ensures that the system correctly responds to the requirement specifications.

```
A[] (!(ATP_Mode == STAND_MODE || ATP_Mode == BEACON_ERROR_STATE) ||
calculate_margin(desired_speed, add_speed_margin) <
calculate_eb_speed(brake_capacity, target_distance, desired_speed)) Imply
P1.initial
```

This property ensures that the system remains in the initial state whenever the conditions are not met.

It states that, at all times and for all possible executions (A[]), if the ATP mode is neither STAND_MODE nor BEACON_ERROR_STATE, or if the calculated speed margin (calculate_margin) is less than the emergency brake speed curve (calculate_eb_speed), then the system must remain in the initial state (P1.initial).

In practical terms, this property guarantees that the system does not change the value of the emergency brake speed curve if it is not in a valid mode or if the margin is insufficient.

`A[] not deadlock`

The property verifies that the system never enters a deadlock state, meaning there is always at least one possible transition from any reachable state.

All specified properties were successfully verified (Figure 5.33), confirming that the requirement was correctly modeled and validated within the UPPAAL system. The results demonstrate that the model behaves as expected under all relevant conditions defined by the requirement, ensuring the correct assignment of the emergency brake speed curve.

Moreover, the model prevents incorrect assignments when these conditions are not met and guarantees that the system never enters a deadlock state. This confirms not only the correctness of the brake curve logic implementation but also the robustness and continuous operability of the modeled system.

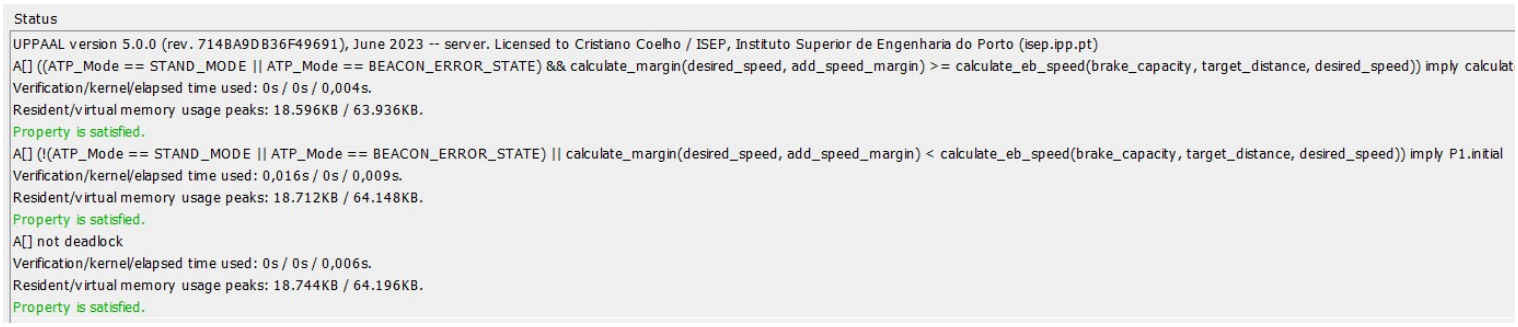


Figure 5.33: Verifications results in UPPAAL for APP-004

Requirement SUBSYS-001:

O The presented UPPAAL model verifies that the emergency brake is activated whenever the traffic signal is red, the train's speed is greater than or equal to the speed limit, and the train is not in standstill.

This model consists of two templates: atp and speedControl.

The atp template (Figure 5.34) represents the ATP system, responsible for triggering the emergency brake. This template consists of two main states: “emerg_brake_false”, which represents the situation where the brake is not activated and “emerg_brake_true”, which represents the state where the brake has been activated.

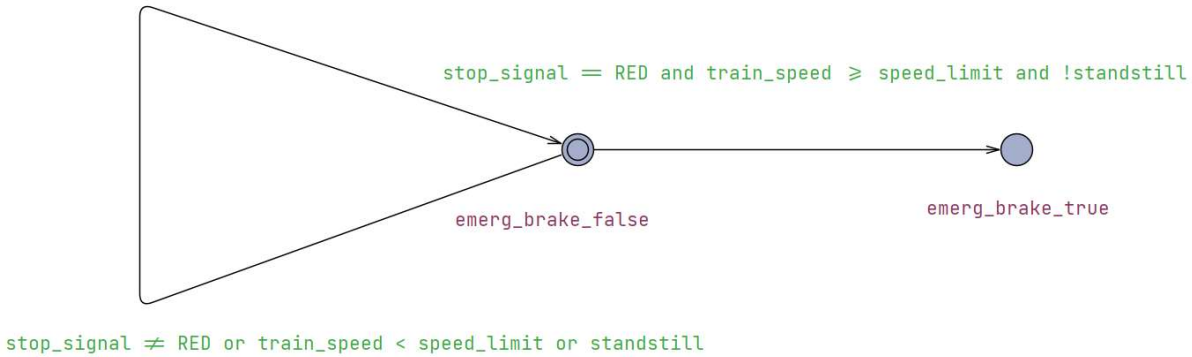


Figure 5.34: “ATP” template in UPPAAL model for Requirement SUBSYS-001

The transition between these states occurs whenever the traffic signal is red, the speed is greater than or equal to the limit and the train is not in standstill, changing its state from “emerg_brake_false” to “emerg_brake_true”, indicating that the emergency brake has been activated.

If any of these conditions are not met, the state “emerg_brake_false” should be activated or maintained.

The “speedControl” template (Figure 5.35) simulates the train's behavior in terms of speed. The train starts at zero speed and while it remains below 120 km/h, it increases its speed by one unit. Once it reaches 120 km/h, the system transitions to a state where speed increment stops, simulating a realistic operational scenario.

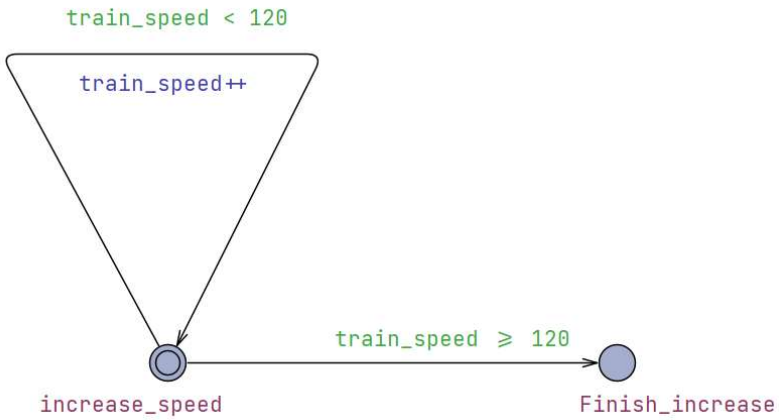


Figure 5.35: “speedControl” template in UPPAAL model for Requirement SUBSYS-001

In terms of declarations, the global declarations section defines all the main variables that will be used by all system processes.

```
//stop_signal states
const int RED = 0;
const int YELLOW = 1;
const int GREEN = 2;

//train speed
int train_speed = 0;

// speed limit equal to 80 km/h
int speed_limit = 80;

//stop signal equal to green
bool stop_signal = GREEN;

//standstill status
bool standstill;
```

Among them are the constants representing the traffic signal states (RED, YELLOW, GREEN), the variable “train_speed”, which stores the train's current speed and “speed_limit”, which sets the speed limit at 80 km/h.

Additionally, the boolean variable “stop_signal” is declared, indicating the current signal state, initially set to green and standstill, which informs whether the train is stopped or moving.

The model also includes three formal verifications, or queries, that check whether the system's behavior is correct. Just like in the NuSMV model, the first two verify the functional aspect, while the last one checks robustness.

A[] (stop_signal == RED and train_speed >= speed_limit and !standstill imply P2.emerg_brake_true)

This specification formalizes requirement SUBSYS-001, which mandates that the system must activate the emergency brake whenever all risk conditions are met. It ensures that the system responds as expected in critical situations, automatically engaging the brake to prevent accidents.

A[] (stop_signal != RED or train_speed < speed_limit or standstill imply P2.emerg_brake_false)

Unlike the previous specification, this one ensures that the emergency brake will not be activated when at least one of the risk conditions is not present, preventing unexpected emergency brake engagements.

A[] not deadlock

Just like in NuSMV, this specification verifies that, in any situation throughout the execution of the model, it will always be possible to continue the execution, advancing to some subsequent state.

As a result, all verifications were successfully confirmed, as we can see in figure 5.36, proving that the model fully meets the functional requirement. It ensures the correct

activation of the brake in emergency situations, the non-activation in safe conditions, and the continuous execution of the system without deadlocks.

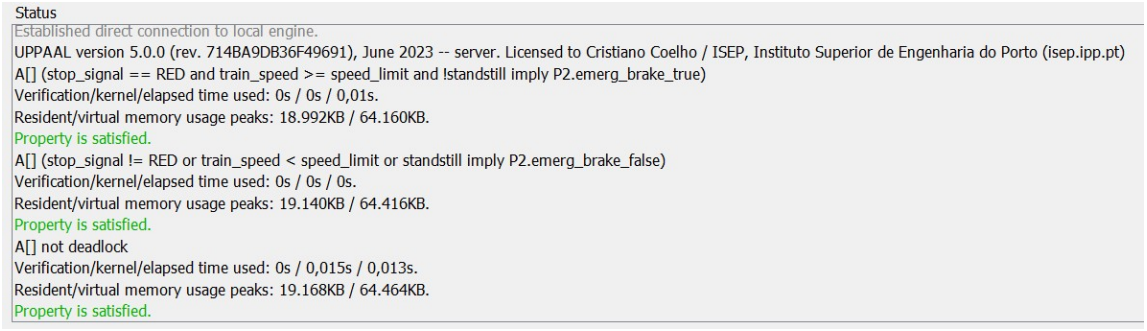


Figure 5.36: Verification results in UPAAL for SUBSYS-001

Requirement SYS-001:

This model developed in UPPAAL represents the behavior specified by the requirement for triggering the end of the shunting mode. It consists of two main state templates: “atp”, which defines the logic for activating the end of shunting indication and “distanceControl,” responsible for simulating the progressive increase in the distance traveled by the vehicle.

As in the previously explained models, this model also declares global variables representing the possible states of ATP, a variable to control the distance, a boolean variable “shunting_mode” indicating whether the system is in shunting mode, and a variable “distanceTraveled,” which receives the current value of “distance,” incremented in the “distanceControl” template, and is used as input for the decision-making process.

```
//ATP MODE states
const int INITIATE = 0;
const int SHUNTING = 1;
const int FORWARD = 2;
const int REVERSE = 3;

// auxiliar varibale to increment distance
int distance;

//Status of the variable that activates the indication for ending the
shunting mode
bool shunting_mode;

// distance travelled
int distanceTraveled;
```

The “atp” template (Figure 5.37) has two states: “shunting_mode_termin_false,” which represents the situation where the activation for ending the shunting mode is not yet active, and “shunting_mode_termin_true,” which represents the activation for deactivating the shunting mode. The transition between these states is conditioned by the logic of the requirement, meaning the system must remain in “shunting_mode_termin_false” as long as the shunting mode is not active or the traveled distance is less than 500 meters. The transition to the

“shunting_mode_termin_true” state occurs when the shunting mode is active and the traveled distance reaches or exceeds 500 meters.

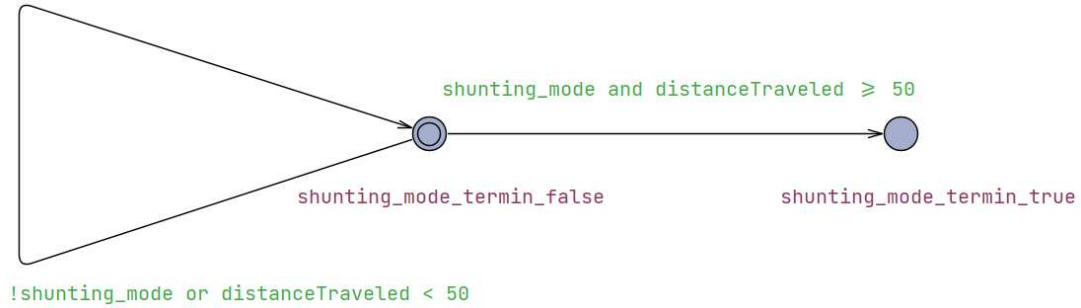


Figure 5.37: “atp” template for shunting mode deactivation

The “distanceControl” template (Figure 5.38) simulates the train's movement process. In the “increase_distance” state, the “distance” variable is continuously incremented until it reaches the value of 500m, being copied to the “distanceTraveled” variable at each cycle. When this minimum value is reached, the system transitions to the “Finish_increase” state, which represents the end of the movement simulation process.

To ensure the reliability of the model, three formal properties are verified.

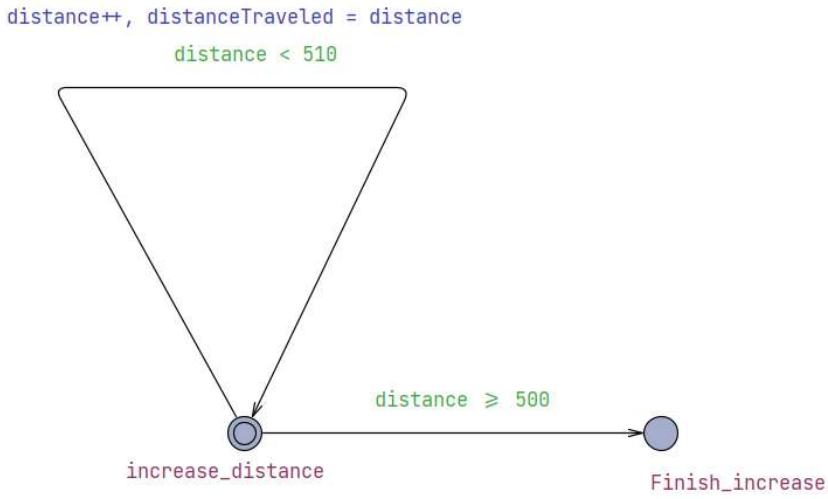


Figure 5.38: “distanceControl” template for UPPAAL model

A[] (shunting_mode && distance >= 500 imply P2.shunting_mode_termin_true)

This property ensures that, in all possible execution paths of the system and at all points in time, whenever the conditions are met, process P2, which represents the ATP decision logic, must transition to the state “shunting_mode_termin_true.” This indicates that the system has correctly triggered the end of the shunting mode.

A[] (!shunting_mode || distance < 500 imply P2.shunting_mode_termin_false)

This property verifies that in all possible executions of the system and at all points in time, if any of the conditions imposed by the requirement are not met, then the system must be in the state “shunting_mode_termin_false” or transition to it. This ensures that the shunting mode is not deactivated in unexpected situations.

A[] not deadlock

Just like in the previously explained models, this property ensures that the system is free from deadlock situations, meaning there are always possible state transitions.

All specified properties were successfully verified (Figure 5.39), confirming that the requirement related to the indication of the end of the shunting mode was correctly modeled and validated in the UPPAAL environment. The results demonstrate that the model behaves as expected under all relevant conditions defined by the requirement, ensuring that the indication is correctly activated only when the specified conditions are met.

Additionally, the model prevents incorrect activations of this indication when those conditions are not fulfilled and guarantees that the system never enters a deadlock state.

```
A[] (shunting_mode && distance >= 500 imply P2.shunting_mode_termin_true)
Verification/kernel/elapsed time used: 0s / 0s / 0,01s.
Resident/virtual memory usage peaks: 19.012KB / 64.236KB.
Property is satisfied.
A[] (!shunting_mode || distance < 500 imply P2.shunting_mode_termin_false)
Verification/kernel/elapsed time used: 0s / 0s / 0,014s.
Resident/virtual memory usage peaks: 19.144KB / 64.488KB.
Property is satisfied.
A[] not deadlock
Verification/kernel/elapsed time used: 0s / 0s / 0s.
Resident/virtual memory usage peaks: 19.164KB / 64.524KB.
Property is satisfied.
```

Figure 5.39: Verification results in UPPAAL for Requirement SYS-001

5.3 Results

In this section, we present the results of the formal verification of requirements, conducted using the UPPAAL and NuSMV tools. The tables compile the identifiers of application, subsystem, and system requirements defined throughout this work, providing a structured overview of the analysis performed for each tool. Additionally, we detail the verification status of each requirement and the time recorded by both tools.

In total, four application requirements, one subsystem requirement and one system requirement were tested, all successfully validated in both environments used. These results confirm the compliance of the implementations with the established requirements, ensuring both the expected functionality and the system's resilience against potential failures.

Tables 5.7 and 5.8 summarize the verification status and execution time for each requirement in NuSMV and UPPAAL, respectively.

Table 5.1: Requirements verification results of NUSMV

Requirement ID	Status	Time (s)
APP-001	Verified	0,025
APP-002	Verified	0,025
APP-003	Verified	0,046
APP-004	Verified	28
SUBSYS-001	Verified	0,039
SYS-001	Verified	0,075

Table 5.2: Requirements verification results of UPPAAL

Requirement ID	Status	Time (s)
APP-001	Verified	0,017
APP-002	Verified	0,019
APP-003	Verified	0,018
APP-004	Verified	0,023
SUBSYS-001	Verified	0,017
SYS-001	Verified	0,043

Among all the requirements, APP-004 was the most challenging one to model and verify in both UPPAAL and NuSMV, requiring meticulous modeling of the emergency brake speed curve calculation and safety margin comparison, where several attempts were made until the expected results were achieved.

Due to its complexity, the UPPAAL model incorporated integer approximations of square root functions and detailed system parameters to simulate real-time speed curve calculations, adapting decimal computations into integer multiples of five to conform to tool limitations. Similarly, the NuSMV model required precise symbolic representations to correctly handle these calculations. In this specific case, the solution involved simulating the behavior of the square root function.

The model verifies critical safety properties, such as ensuring that the emergency brake speed curve is updated only when safety margins are met and preventing incorrect updates. Both tools successfully verified these properties, confirming the correctness of the brake curve logic and system safety.

This requirement exemplifies the delicate balance between model fidelity and verification performance, demonstrating how both UPPAAL and NUSMV can be effectively used to verify complex real-time safety-critical functions.

Analysis of verification Times

Figure 5.40 illustrates a comparison of verification times for each requirement, where both tools, UPPAAL and NUSMV, were used. The chart clearly shows that UPPAAL is more effective in verification for almost all requirements. The exception is APP-004, where NuSMV takes significantly longer, 28 seconds, but the verification times for the remaining models are comparable, with a significant difference compared to UPPAAL. This leads us to conclude that NuSMV presents an optimized model formulation and efficient processing.

This comparison highlights UPPAAL's efficiency in handling timed automata-based models, while NuSMV demonstrates robust performance for most simpler cases but may experience slower processing in complex scenarios, such as performance degradation in more computationally intensive cases, as seen in APP-004. Both tools, however, maintain correctness and successfully validate all requirements.

All requirements were formally verified without errors or deadlocks, proving the correctness and robustness of the developed models.

The efficient performance of both tools confirms that they both provide a robust solution for verifying safety-critical railway control systems.

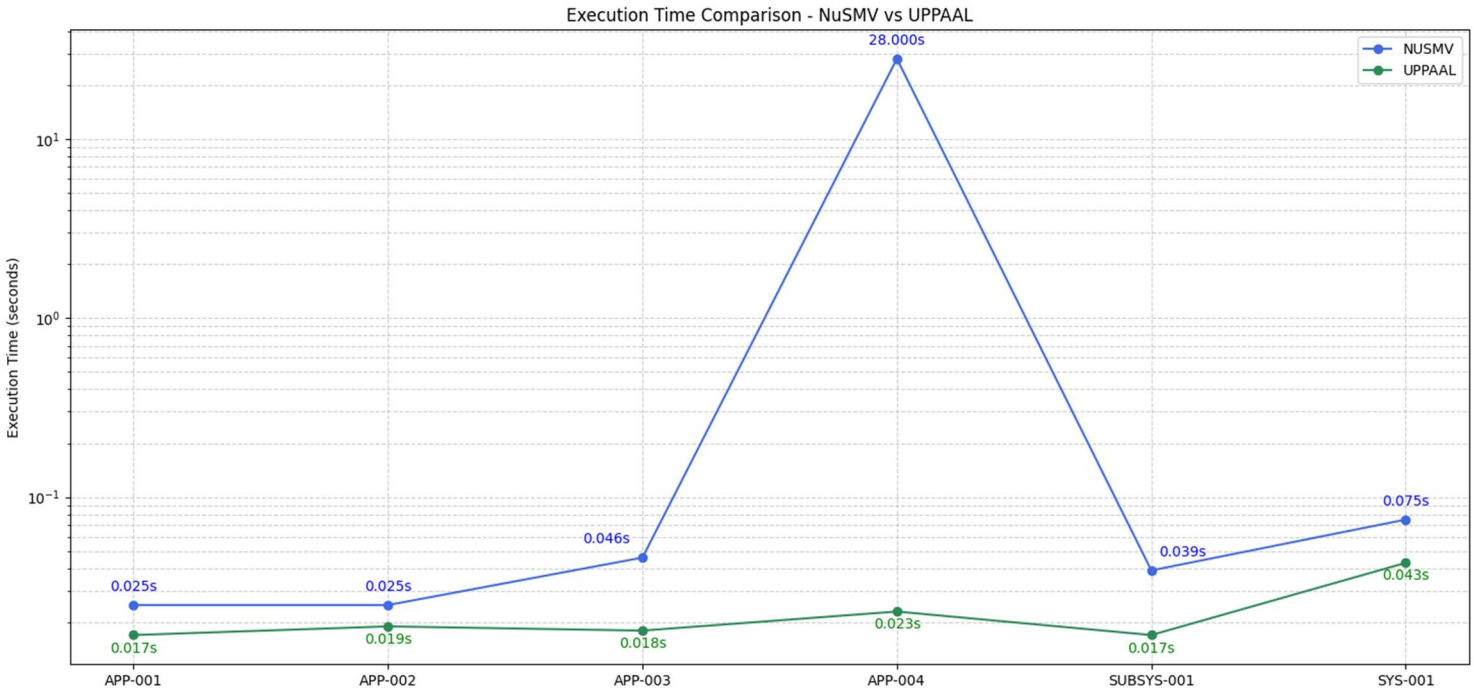


Figure 5.40: Execution time comparison between NuSMV and UPPAAL

6 Conclusions and Future Work

This thesis aimed to present a comprehensive formal verification of critical safety requirements for a railway control system, using two widely recognized formal verification tools: UPPAAL and NuSMV. The models developed in both tools accurately represent application, subsystem, and system-level requirements, focusing on safety-critical properties such as emergency brake activation and emergency brake speed curve calculation.

The verification results demonstrate that both tools effectively validate the functional correctness and safety constraints of the system. UPPAAL showed superior performance in handling timed automata and real-time behaviors, while NuSMV provided robust symbolic model-checking capabilities for discrete and simpler models. It is important to highlight that all requirements were fully verified without deadlocks or errors, confirming the correctness, safety, and reliability of the system in all tested scenarios.

A notable challenge was the modeling of the emergency brake speed curve, specifically for application requirement APP-004, which required careful adaptation of mathematical functions and precise handling within the limitations of both tools. The successful verification of this requirement highlights the capability of formal methods to address complex real-time safety-critical functions in railway systems.

Looking ahead, to ensure the safety of these systems, a relevant extension of this work would be the integration of formal techniques for requirement elicitation, formalization, and understanding, such as those provided by NASA's FRET (Formal Requirements Elicitation Tool) (PROGRAM, s.d.). FRET facilitates the transformation of natural language requirements into formal temporal logic specifications, enabling seamless integration with model verification tools like NuSMV.

The implementation of FRET can optimize the verification process and enhance accuracy in requirement validation, reducing potential errors in translating informal requirements into formal specifications. Additionally, exploring automated methods for generating verification queries and systematic coverage analysis through FRET could significantly improve productivity and reliability in verification results.

Overall, combining formal verification with advanced requirement engineering tools like FRET can strengthen the precision and usability of formal methods, contributing to a safer and more efficient development of safety-critical railway systems.

All the objectives proposed in this thesis were successfully achieved. Initially, the goal was to model and verify the system exclusively using NuSMV, however, the scope of the work was expanded to also include modeling with UPPAAL. This extension allowed for a comparative analysis between the two tools, providing a deeper understanding of their respective strengths and limitations when applied to the verification of safety-critical railway systems.

Bibliography

Admin, 2022. *Railway Signalling Concepts*. [Online]
Available at: https://www.railwaysignallingconcepts.in/2-3-4-aspect-signalling-headway-calculation/
[Accessed Dec 2024].

Almakhour, M., Sliman, L., Samhat, A. E. & Mellouk, A., 2020. Pervasive and Mobile Computing. *Verification of smart contracts: A survey*, p. 8.

Argenia, n.d. *Platform for Railway Signaling*. [Online]
Available at: https://argenia-railway-technology.myshopify.com/products/frauscher-axle-counters
[Accessed 23 Dec 2024].

Arzaghi, S., 2021. School of Mathematics, Statistics and Computer Science. *An Introduction To Linear Temporal Logic*, Feb, pp. 1-2.

Badshah, D. A., 2024. *Formal Specification in Software Engineering*. [Online]
Available at: https://afzalbadshah.medium.com/formal-specification-in-software-engineering-b66852d96fc6
[Accessed Jan 2025].

Baier, C. & Katoen, J.-P., 2008. *Principles of Model Checking*. Cambridge, Massachusetts: The MIT Press.

Basit-Ur-Rahim, M. A., Arif, F. & Ahmad, J., 2014. Formal Verification of Sequence Diagram using. *2014 World Congress on Computer Applications and Information Systems, WCCAIS 2014,* pp. 1-6.

Bertino, E., 2012. Chapter 23 - Policies, Access Control, and Formal Methods. In: S. K. Das, K. Kant & N. Zhang, eds. *Handbook on Securing Cyber-Physical Critical Infrastructure*. Oxford: Morgan Kaufmann, pp. 573-594.

Beyer, D. & Lemberger, T., 2017. Software Verification: Testing vs. Model Checking. *A Comparative Evaluation of the State of the Art*.

Busard, S. et al., n.d. Universit´e catholique de Louvain, Louvain-La-Neuve, Belgium. *Verification of railway interlocking systems*.

Cavada, R. et al., 2005. *NuSMV 2.6 User Manual*. Povo (Trento) – Italy: FBK-irst.

Cimatti, A., Clarke, E., Giunchiglia, F. & Roveri, M., 2000. NUSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer,* 2(4), pp. 410-425.

Cimatti, A., Pistore, M. & Roveri, M., 1999. *MCT'99: A Hands-on Tutorial on Model Checking: NuSMV*. s.l., s.n.

Clarke, E. M., 2007. School of Computer Science Carnegie Mellon University Pittsburgh. *The Birth of Model Checking*, Feb, p. 3.

Clarke, E. M., 2007. School of Computer Science Carnegie Mellon University Pittsburgh. *The Birth of Model Checking*, Feb.

Clarke, E. M., 2007. School of Computer Science Carnegie Mellon University Pittsburgh. *The Birth of Model Checking*, Feb.

Clark, S., 2024. *HISTORY OF RAILWAY SIGNALLING (from the Bobby to the Balise)*, p. 8.

Commission, E., n.d. *ETCS Levels and Modes*. [Online]
Available at: https://transport.ec.europa.eu/transport-modes/rail/ertms/what-ertms-and-how-does-it-work/etcs-levels-and-modes_en
[Accessed Dec 2024].

Commission, E., n.d. *History of ERTMS*. [Online]
Available at: https://transport.ec.europa.eu/transport-modes/rail/ertms/history-ertms_en
[Accessed Dec 2024].

Connor, D. P. & Schmid, P. F., 2023. *The Railway Technical Website*. [Online]
Available at: http://www.railway-technical.com/signalling/train-protection.html
[Accessed Dec 2024].

Connor, D. P. & Schmid, P. F., 2023. *The Railway Technical Website, Train Protection*. [Online]
Available at: http://www.railway-technical.com/signalling/train-protection.html
[Accessed Dec 2024].

David, A. et al., 2018. *Uppaal SMC Tutorial*, Jan.

Department of Computer Science, U. o. O., n.d. *prismmodelchecker*. [Online]
Available at: https://www.prismmodelchecker.org/
[Accessed Jan 2025].

DOULOS, 2025. *The Designer's Guide to PSL*. [Online]
Available at: https://www.doulos.com/knowhow/psl/
[Accessed 14 march 2025].

Evans, A. W., 1996. The economics of Automatic Train Protection in Britain. *Transport Policy*, p. 106.

Ferroviário, M. N., 2021. *Sinalética, Comunicação e Segurança\Sinais e Avisos*. [Online]
Available at:
https://colecao.fmnf.pt/ficha.aspx?id=186&ns=216000&filtro=2430341101180630181842470150980281820331081951 28&modo=album
[Accessed Dec 2024].

Flammini, F., 2013. Industrial Engineering & Management. *Automatic Train Protection Systems*, p. 1.

Ganguly, S. S., 2001. History of Railway Signalling.

Group, S. M., n.d. *Model of Gregory's semaphore signal*. [Online]
Available at:
https://collection.sciencemuseumgroup.org.uk/objects/co27021/model-of-gregorys-semaphore-signal
[Accessed 30 Dec 2024].

Group, S. M., n.d. *Telegraph block instrument*. [Online]
Available at:
https://collection.sciencemuseumgroup.org.uk/objects/co209536/telegraph-block-instrument
[Accessed Dec 2024].

Gunter, E. & Peled, D., 2005. Formal Aspects of Computing. *Model Checking, Testing and Verification Working Together*, 01 Aug, pp. 201-221.

H.Song, Liu, H. & E.Schnieder, 2018. A train-centric communication-based new movement authority. *IEEE Transactions on Intelligent Transportation Systems*, pp. 2328-2338.

Holzmann, G. J., 1997. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. *The Model Checker SPIN*, pp. 279-295.

II, S. M. W., n.d. *What is model checking?*. [Online]
Available at: https://klu.ai/glossary/model-checking
[Accessed Jan 2025].

Instituto da Mobilidade e dos Transportes, I., n.d. *REGULAMENTO GERAL DE SEGURANÇA*. [Online]
Available at: https://www.imt-ip.pt/sites/IMTT/Portugues/TransportesFerroviarios/CaminhodeFerro/RegulamentacaoTecnicaSeguranca/Documents/Regulamenta%C3%A7%C3%A3o%20Tecni

ca%20De%20Seguran%C3%A7a/RGS%20V%20Sistemas%20Complementares%20de%20Seguran%C3%A7a%20V1.0%20-%20DRAF

Johnson, C. G., 2007. Genetic Programming. *Genetic Programming with Fitness based on Model Checking*, April.

Lukács, G. & Bartha, T., 2024. Periodica Polytechnica Transportation Engineering. *Verification of Railway Control Systems Using Model Checking and CTL, Explained Through a Case Study*, pp. 402-411.

Martins, J., 2014. *Segurança Ferroviária*. [Online]
Available at:
https://www.trainlogistic.com/Imagens/PDF/Artigos/PDF_SegurancaFerroviaria.pdf

Martins, J., 2017. *Convel - Ficha Técnica*. [Online]
Available at:
https://www.trainlogistic.com/pt/Comboios/Gabinete/fich_convel.htm
[Accessed 15 Dec 2024].

Mir, A. A., Balakrishnan, S. & Tahar, S., 2000. Modeling and verification of embedded systems using cadence SMV. *Canadian Conference on Electrical and Computer Engineering,* Volume 1, pp. 179-183.

NUSMV, 2014-2023. *Overview*. [Online]
Available at: https://nusmv.fbk.eu/overview.html
[Accessed March 2025].

Palumbo, M., 2013. Railway Signalling since the birth to ERTMS. Nov, pp. 1-2.

Pandrol, 2024. *Barre sur traverse*. [Online]
Available at: https://www.pandrol.com/fr/product/barre-sur-traverse/
[Accessed 30 Dec 2024].

Pereira, D., Proença, J., Nandi, G. & Tovar, E., 2024. *Introduction to Model Checking,* Porto: s.n.

PROGRAM, N. T. T., n.d. *aeronautics, FRET : Formal Requirements Elicitation Tool*. [Online]
Available at: https://software.nasa.gov/software/ARC-18066-1
[Accessed 17 May 2025].

Rail, I., 2024. *ATC System Evolution & RFID's Safety Role in Rail Industry*. [Online]
Available at: https://www.intertechrail.com/articles/evolution-of-atc-system-and-rfid-in-rail-safety
[Accessed Dec 2024].

Railways, E. U. A. f., n.d. *European Rail Traffic Management System (ERTMS)*. [Online]
Available at: https://www.era.europa.eu/domains/infrastructure/european-rail-traffic-management-system-ertms_en
[Accessed Dec 2024].

Railways, E. U. A. F., n.d. *European Rail Traffic Management System (ERTMS)*. [Online]
Available at: https://www.era.europa.eu/domains/infrastructure/european-rail-traffic-management-system-ertms_en
[Accessed Dec 2024].

railwaysignalling.eu, 2015. *Principles of Railway interlocking*. [Online]
Available at: https://www.railwaysignalling.eu/railway-interlocking-principles-railwaysignalling
[Accessed 26 Dec 2024].

Railworks, 2024. *Automatic Train Supervision (ATS)*. [Online]
Available at: https://www.railworks.com/projects/automatic-train-supervision-ats
[Accessed Dec 2024].

SCIENCE, D. O. C., n.d. *FDR*. [Online]
Available at: https://www.cs.ox.ac.uk/activities/concurrency/tools/fdr/
[Accessed Jan 2025].

Shedd, T. C., Vance, J. E., Allen, G. F. & Britannica, T. E. o. E., 2024. Encyclopedia Britannica. *railroad*, 15 Dec..

SIEMENS, 2024. *Interlocking Systems*. [Online]
Available at: https://www.mobility.siemens.com/global/en/portfolio/rail-infrastructure/mainline/interlocking-systems.html

Simulink, n.d. *Simulink Design Verifier*. [Online]
Available at: https://www.mathworks.com/products/simulink-design-verifier.html
[Accessed 20 Mar 2025].

Site, T. R. T. W., 2023. *Signalling*. [Online]
Available at: http://www.railway-technical.com/signalling/
[Accessed Dec 2024].

SlideServe, 2014. *Temporal Logics*. [Online]
Available at: https://www.slideserve.com/collin/temporal-logics
[Accessed Jan 2025].

Software, C., 2024. *Automatic Train Control*. [Online]
Available at:
https://www.criticalsoftware.com/multimedia/critical/en/Y1cMA1tGf-CSW-Railway-WhitePaper-AutomaticTrainControl_2.pdf
[Accessed Dec 2024].

T&N, 2023. *Sistema Convel com sucessor anunciado*. [Online]
Available at: https://www.transportesenegocios.pt/sistema-convel-com-sucessor-anunciado/
[Accessed Dec 2024].

Trackopedia, n.d. *European Train Control System (ETCS)*. [Online]
Available at:
https://www.trackopedia.com/en/encyclopedia/infrastructure/european-train-control-system-etcs#top
[Accessed Dec 2024].

Transportes, I. d. M. e. d. T., n.d. *REGULAMENTO GERAL DE SEGURANÇA*. [Online]
Available at: https://www.imt-ip.pt/sites/IMTT/Portugues/TransportesFerroviarios/CaminhodeFerro/RegulamentacaoTecnicaSeguranca/Documents/Regulamenta%C3%A7%C3%A3o%20Tecnica%20De%20Seguran%C3%A7a/RGS%20V%20Sistemas%20Complementares%20de%20Seguran%C3%A7a%20V1.0%20-%20DRAF
[Accessed Dec 2024].

UPPAAL, 2009. *UPPAAL*. [Online]
Available at: https://uppaal.org/
[Accessed 18 March 2025].

UPPAAL, n.d. *Uppaal 5 released!*. [Online]
Available at: https://uppaal.org/
[Accessed Jan 2025].

Van Lamsweerde, A., 2000. Formal specification: A roadmap. *Proceedings of the Conference on the Future of Software Engineering, ICSE 2000,* pp. 147-159.

Website, T. R. T., 2023. *Route Signalling*. [Online]
Available at: http://www.railway-technical.com/signalling/route-signalling.html
[Accessed Dec 2024].

Wien, F. T., n.d. *Model checking - A short informal Introduction*. [Online]
Available at: https://embsys.technikum-wien.at/projects/decs/verification/formalmethods.html
[Accessed Jan 2024].

Wikipedia, 2024. *Railway signalling*. [Online]
Available at: https://en.wikipedia.org/wiki/Railway_signalling#cite_note-:0-1
[Accessed Dec 2024].

Xionga, X., Liua, J., Zhangb, M. & Dingc, Z., 2010. 34th Annual IEEE Computer Software and Applications Conference Workshops. *Modeling and Verification of an Automatic Train Protection System*, pp. 226-231.

Yuan, Z. et al., 2021. IEEE Transactions on Intelligent Transportation Systems. *Virtual parameter learning-based adaptive control*, 22(12), pp. 7943-7954.