Università degli Studi di Firenze

Scuola di Ingegneria - Dipartimento di Ingegneria dell'Informazione

---

Tesi di Laurea Magistrale in Ingegneria Informatica

Implementation of a parser for railway interlockings: from XML to UMC

Candidato
Alessio Venturi

Relatori
Prof. Alessandro Fantechi
Prof. Laura Carnevali

Correlatore
Ing. Gloria Gori

Anno Accademico 2021/2022

Acknowledgements

I would like to give my acknowledge to my supervisor, Professor Alessandro Fantechi, and engineer Gloria Gori. Their guidance, expertise were essential in helping me to achieve my goals, and I am deeply grateful for their assistance.

A special thank to my mother, Krystyna and my brother Andrea, fundamental in my life. Without you, I would not have been the person I am today.

Another special thanks to my mates Giuseppe, Alessio, Giorgio, Fatjon, Andrea, Leonardo, Martin and Kevin. In Italy we use to say : "Friends are the family you choose", and you guys are that family.

To my university colleagues, in particular Geme and Giunta for accompanying me on this journey sharing moments and the same passions. Despite everything, I know that I can always count on my *monkeys*.

Last but not least, to Eva, for coming into my life without knocking and making each day unique. Without you, it would not be the same.

Abstract

A crucial mechanism called railway interlocking regulates the way trains move along tracks and switches, ensuring the safety of railway operations.

Interlocking systems have evolved from mechanical and electromechanical systems to computer-based systems due to improvements in technology. However, the dependence of interlocking systems on software has created additional difficulties in maintaining their reliability.

Reliability includes the ability of the system to provide the desired functionality in a timely and accurate manner despite challenging circumstances. Software design, development, testing, and maintenance are just some of the factors that must be considered to achieve software reliability in interlocked systems.

This thesis implements and distributes a parser to convert XML files related to railway stations into UMC code and their approximate graphics representations, through a graphical interface that offers users an easy way to interact with the tool.

It also discusses the role of UMC, the proposed algorithm, and software reliability assurance in interlocking systems.

Software reliability is essential to ensure the safe and reliable operation of interlocking systems.

Contents

Chapter 1

Introduction

An interlocking is a safety-critical system that regulates train movement in a station and between nearby stations in the context of railway signaling. Since the 1990s, interlocking systems —which were initially electro-mechanical systems built for almost a century— have been gradually being replaced by modern computerized systems. This is an ongoing process and in many geographical areas this replacement hasn't been done yet. However, railways infrastructure companies are still interested in finding more affordable systems that can guarantee the same or higher level of safety.

The interlocking keeps track of the condition of the railway's yard (such as points, switches, and track circuits) and approves or denies train routing in line with the general railway safety and operating laws that apply in the area or nation where the interlocking is situated.

The interlocking is responsible for managing routes in such way as to avoid derails and collision. Each train has a destination and a route assigned from the interlocking that is also responsible for managing the relative signals, which determine the velocity and the obligation to stop trains.

Traditionally, the idea of route has served as the foundation for the design

of a railway interlocking system. A route is a path that a train can take to get from an entry track to an exit or stop track. It is made up of a series of track circuits and points. Railway engineers determine a set of feasible routes based on established guidelines, norms, and operational opportunities given a station's track layout. Permission to move a train is actually given by setting a route in this set for it.
Setting a route typically means:

- Verifying that the route is free[1] of other trains, by track circuits or other presence sensors.
- Command the points to their correct position.
- Check that the points have actually reached the commanded position.
- Set the signals to give the driver permission to move.

A data structure known as a control table, which is unique to the station where the system is located, is typically used to define the instantiation of these general rules on a station topology (made up of the track layout and the set of routes). The interlocking system's development is guided by the control table. We therefore intend a system that simply accepts requests for reservations and grants or denies reservations based on safety regulations, up until the reservation has been fully utilized (i.e. the track is once again free) or has been safely canceled. Centralised interlockings are difficult to design and expensive to test against safety standards. The need to confirm all potentially conflicting combinations of various routes through the station is what causes the complexity.

[1]Free route means that all track circuits and the points forming it are available, in addition to any other possible extra conditions, such as those dictated by sidewall protection or overlap sections.

Distributed interlocking, on the other hand, may be advantageous for the possibility of easier deployment and maintenance, for copper-free communication if wireless links are adopted, for a facilitation of modular formal verification, leading to a simpler certification process, and for other reasons.

To address the challenges of designing and testing distributed interlocking, it is essential to test these systems thoroughly to ensure they meet safety standards. To facilitate this process, a parser has been created to convert railway XML files into UMC models, allowing for direct testing of configurations. The XML files describes each element and each route of the railway network, while UMC is a verification framework for analysis and model checking of system designs. This parser builds on the architecture proposed in [5] and provides a solution for improving railway safety while reducing costs and complexity. It also implements a function to draw an approximate visual representation of the railway network, since geographic coordinates were not available.

Thesis structure

The content of this thesis is organised as follow: In Chapter 2, the general concepts and terminology of software dependability, model checking and interlocking are shown. In Chapter 3, we describe how using UMC model, we have implemented and verified these systems for some key properties. In Chapter 4, we report on a configuration tool developed to parse railway XML file into UMC models to quickly test every route of a given family of interlocking systems. Finally, in Chapter 5 we show some of the results obtained, while also giving also an overview of the full workflow of the tool.

Chapter 2

General concepts and terminology

Nowadays, software systems are everywhere and have a huge impact on how we live. We rely on software to perform essential tasks in many areas ranging from transportation systems to healthcare, banking, and telecommunications. However, as software systems become more intricate and interconnected, they also become more prone to malfunctions, which can have serious repercussions for stability in terms of safety, security and economy.

The history of software dependability begins with the use of software systems to perform calculations in science and math. Because the earliest software systems were relatively straightforward, dependability was not a major concern. However, as software systems became more complex, so did the need for reliable software. New concepts such as software dependability, interlocking system or model checking emerged and which are crucial to understand the work presented in this thesis. **Software dependability** is the capacity of a software system to perform its intended functions precisely, consistently and dependably.

Railway **interlocking systems** play a critical role in ensuring the safety and efficient operation of rail networks. Given the safety-critical nature of these systems, ensuring their dependability is critically important. The necessary formal verification can be applied to it to analyze and enhance the dependability of railway systems.

Having said that, **model checking** is a critical approach used in software dependability to verify that systems work as design while minimizing the potential of mistakes or failures.

2.1 Software dependability

CENELEC EN 50128 [1], the European standard that specifies the process and technical requirements for the development of software for use in railway control and protection applications, acknowledges the importance of software dependability in railway systems and requires that software developers take appropriate measures to ensure the safety and reliability of railway control and protection systems.

Availability, reliability, maintainability, and occasionally other qualities like durability, safety, and security are all measured by a system's dependability. Systems have a "criticality" property that refers to their capacity to manage "critical" tasks in a way that encourages users to "trust" the system and thus be able to use it without special apprehension: a reliable system is one that consistently performs as expected.

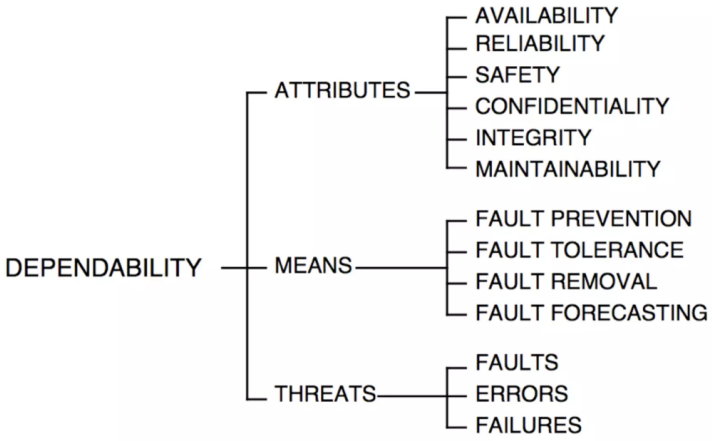

Figure 2.1: Dependability tree.

Dependability can be decomposed into three main elements (Fig 2.1):

- Attributes: a way to evaluate a system's dependability.

- Means: techniques to improve the dependability of a system.
- Threats: the factors that can compromise a system's dependability.

2.1.1 Attributes

Attributes are a way to assess the dependability of a system and can be seen as qualities the system. They can assume a qualitative or quantitative measure.

Reliability

Probability that the system will run smoothly at time t given that it ran smoothly between t_0 and t.

In general, *Reliability* R at given time t can be defined with an exponential form of the type:

$$R(t) = e^{-\lambda t}$$

where $\lambda = \frac{1}{MTTF}$, λ is the failure rate and $MTTF$ is the **Mean Time To Failure**. On the other hand, situations that do not fall under reliability will indicate the unreliability Q of a system . Thus:

$$Q(t) = 1 - R(t)$$

Maintainability

It is the ease with which a system or software component can be modified to change or add functionality, correct defects or anomalies, improve performance or other attributes, or adapt to a changed environment. It can be calculated as probability that the system will run smoothly at time t given that it didn't run smoothly between t_0 and t.

It is defined:

$$M(t) = 1 - e^{-\mu t}$$

where μ is the repair rate, the inverse of **Mean Time To Repair** (MTTR). In Fig. 2.2 is shown how the Time To Failure (TTF) and Time To Repair (TTR) are measured.

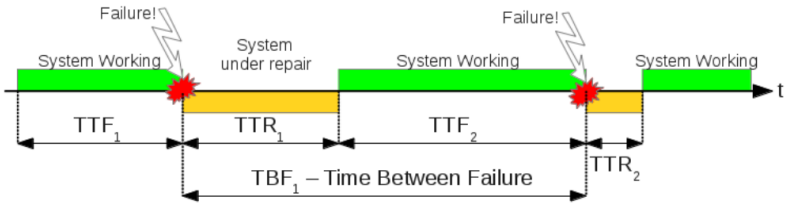

Figure 2.2: Temporal line of MTTF and MTTR [6].

Availability

The probability that a system will perform properly when utilized under specified conditions in an ideal support environment at a specific moment in time. Ratio of the **Mean Time To Failure** and the sum of **Mean Time To Repair** and **Mean Time To Failure**:

$$A(t) = \frac{MTTF}{MTTF + MTTR} \approx \frac{MTTF}{MTBF}$$

Since the repair time should be much less than the operating time, making this approximation:

$$MTBF \approx MTTF \iff MTTF >> MTBF$$

then, the *Availability* value should be as close to 1 as possible.

Safety

Probability that the system is "safe" at time t, having been in the interval $[t_0, t)$. If a system is functioning normally or has failed in a benign way, it is said to be in a "safe" state (fail-safe). *Reliability* and the idea of coverage can be used to define the safety of a system:

$$S(t) = R(t) + C(1 - R(t)) = R(t) + Q(t)$$

where the C factor represents the coverage provided by the safety mechanisms. In order for a system to be safe, it must have components that, whenever possible, prevent failures and in other cases, mitigate their effects. This ability is represented by the C coefficient over the set of "covered" failures. The most crucial safety mechanisms are those that operate at the system level and can perform a Safety Assessment, i.e. thorough evaluation of hazards.

Security

The difference between *Safety* and *Security* is that the latter one is different from Safety in that it refers to data protection. Since it contains what are known as "intentional failures," or errors that come from active malicious sources, security does not properly fall under the category of dependability. Assuming that we are working with closed systems, we will treat security separately while keeping in mind that all other disciplines may be dependent on it.

2.1.2 Means

Fault Prevention

The goal of *Fault Prevention* is to keep systems from developing faults. Use of effective implementation strategies and development methodologies can achieve this and respect the first role: never enter failures, fix any errors that still remain.

Fault Removal

Fault Removal is classified into two types: removal during development and removal during use. Before a system is put into production, removal during development must be verified in order to find and fix any errors. A system is required to track failures after systems are put into production and to eliminate them through routine maintenance.

Fault Tolerance

A system's ability to maintain proper operation in the event that one or more of its components fail due to a fault is known as fault tolerance.
Even a minor failure can result in total breakdown, so if its operating quality does decrease at all, the decrease is proportional to the severity of the failure. When a component of a system fails, a fault-tolerant design enables the system to continue operating as intended, possibly at a reduced level, rather than ceasing to function altogether. It can be understood in two ways:

- Software (Fault Tolerance): *Fault Tolerance* through software. It is a broad field where software that can handle errors from both hardware and other software is known as fault-tolerance algorithms.

- (Software Fault) Tolerance: Managing software-related errors. It makes an effort to emulate the standard hardware fault-tolerance mechanisms using redundancy and diversity (i.e. in case of failure)

Fault Forecasting

Fault Forecasting predicts likely faults so that they can be removed or their effects can be circumvented. It targets the errors which are systematic and do not follow a probability distribution. In other words of *Fault Forecasting*: probability that the program finds no bugs at time t. It is very challenging to quantify this parameter because it depends on the structure of the program under consideration as well as a specific, complex configuration of the input state. If we have a lot of components, we have to test them all, which reduces overall system reliability while increasing costs. Two or three years of testing for a critical system may be acceptable; beyond that, the costs become prohibitively expensive.

2.1.3 Threats

In software systems, the configuration of all the system's bits constitutes the system's state. Knowing that system are susceptible to flaws and can break down is important to recognize and classify them. A fault causes an error, which is an incorrect state of the system. An error becomes a failure if it starts spreading beyond the system's interface and becomes observable as you can see in Fig 2.3

It is important to keep in mind that an error can cause other errors even before it manifests itself by aggravating the system's erroneous state [6]. Finally, an error may also not manifest itself.

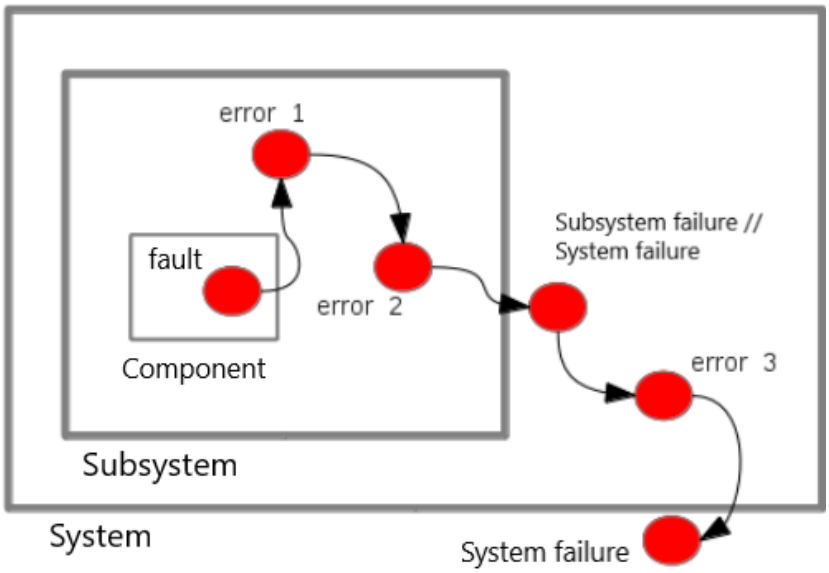

Figure 2.3: Workflow of error propagation.

There are essentially two types of faults:

- Mechanical Defects (or random)
- Systematic failures

While mechanical defects typically occur at the hardware level and only manifest when a bug in the code is executed, the project failures are systematic and often the result of a poor design. It is essential to point out that these errors are always present in the system.

On the other hand, failures can be divided into two groups:

- Benign failures: they result in a loss of value (in the economy) that is equal to or less than the benefit the system provided.
- Catastrophic failures: result in a value loss that is greater than the system's contribution to the benefit.

2.2 Interlocking System

2.2.1 Physical Domain

Interlocking system's primary goal is to make sure that two trains never find themselves on the same track, which could be extremely dangerous [5]. This circumstance is avoided in route-based interlocking systems by requesting that trains reserve their route before occupying it.

All track elements that make up the route must give their consent before the train can reserve it. In reality, a rail network is made up of numerous track elements of various types. As it was already mentioned, we also introduced a new class called Signal in this work to simulate the behavior of a railway marker board in a simple/complicated setting.

From now on, we will assume that the direction from right to left is *up* and left to right is down. Fig. 2.4 shows a piece of railway network and in order to provide a better understanding of the concepts, we will give some explanations. The linear **545** has **A965** as *up* neighbor and **14M** as *down* neighbor. It has also two signals, **jy545m** in *down* direction and **j545m** in *up* direction, which are used according to the train direction. The point **14M** has three neighbors : **A911** in *stem*, **545** in *plus* and **547** in *minus*.

Having said that, let's list them with their rules:

- Linear section:
  - has no more than two neighbors, one in the *up* exit and the other in the *down* exit, which could either be linears or points.
  - has no more than two signals, one direction each at most
  - exhibits bidirectional but not simultaneous behavior
- Point section:

  - has three neighbors called *stem*, *plus* and *minus*, which could either be linears or points
  - can be switched in two position, *plus* and *minus*, connecting it to the *stem* and forbidding the traffic from *plus* to *minus* and vice versa
  - exhibits bidirectional but not simultaneous behavior
  - forms the straight path with the *plus*
  - forms the branching path with the *minus*

- Signal:
  - is connected to only one linear
  - is by default in red state

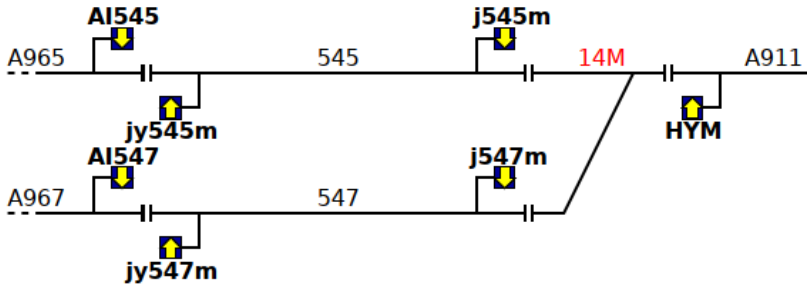

Figure 2.4: Example network composed by five linears (A965, 545, A911, A967 and 547), a point (14M) and seven signals ,where HYM, jy547m and jy545m are in up direction (right to left) and AI545, AI547, j547m and j545m are in down direction (left to right).

2.2.2 Interlocking system with local controllers

The core of the suggested interlocking logic is to install a controller in each element of the layout, that is, in each point segment and in each linear section.

In this way, the track components are compared to network communication nodes that can be queried for reservations.

Each controller reads the local sensor and directs local actuators and surrounding nodes when a path is booked. The physical topology of the distributed system can then follow the track layout, since each node only needs to talk to neighboring nodes and it is expected that trains can enter the station from one of its ends. Trains must get a clear route from the distributed interlocking system in order to enter the station.

The approaching train itself demands a path and waits for approval, therefore modeling a train covers both; participating in the reservation system as an initiator and modeling movement across the network.

2.2.3 Distributed systems

Distributed systems can be viewed as a collection of interconnected nodes which must converge to a shared knowledge, about the data and the state of the network (e.g. detect a node failure). To achieve this result, we need to check the following conditions:

- Validity: if the value is proposed from one of the nodes of the system, the choice of this value can be validated
- Agreement: every node will agree to the same value creating concordance
- Termination: each node decides a value sooner or later

In a distributed system it is usually needed to introduce *checkpoints*, with the aim of guaranteeing the system consistency i.e. every node has correct and shared knowledge. In order for the system to resist faults, it is necessary

to ensure that all the actions that take place between two checkpoints are considered atomic To be considered atomic, the actions have to follow the ACID rules:

- Atomicity: the action is unbreakable during its execution.
- Consistency: it does not breach the system.
- Isolation: it is independent and does not affect other atomic action.
- Durability: if executed, its effect are registered permanently.

Domino-effect incidents have the most disastrous consequences in the context of software dependability, which is why they are one of the most important concerns in distributed algorithms.
In 1996, Delvosalle defined domino effect as a primary accident in a primary installation (this event might not be a major accident), inducing one (or more) secondary accident(s), concerning secondary installation(s). This (these) secondary accident(s) must be a major one(s) and must extend the damage caused by the primary accident. Therefore, the domino effects act in a chain, involving a number of installations [4].

A possible solution to this problem is the technique of recovery lines. These lines are drawn following two rules:

- They have to pass through checkpoints.
- They must not intersect the communication lines.

By establishing that all the action occurred between two *recovery lines* are considered atomic, the domino effect will stop on the *recovery lines*, restarting from the last commit[1].

[1]Commit: the event that finishes the atomic action

2.2.4 The Two Phase Commit Protocol (2PC)

As previously stated, in order for a train to proceed, it must first request a route and then wait for the route to be granted. While the train is performing the route, elements that are no longer committed by the train can be released. Route reservation is now a global state concept in distributed systems that is dependent on the states of individual elements that are local to each node. To establish a path, a distributed consensus algorithm must reach consensus on this global state.

Route Reservation

The algorithm used for route reservation is the standardized protocol called two-phase commit (2PC) [12], an atomic commitment protocol for distributed systems that guarantees the atomicity, consistency, isolation, and durability (ACID) of a transaction. A sequence of database interactions on several independent data sources can be merged together and committed or rolled back as a single transaction using 2PC protocol.

This is a centralized protocol with a single coordinator (train) and the railway´s nodes acting as slaves.

As shown in Fig. 2.5 the two phases are commonly called: *Voting phase* and *Decision phase*, respectively. During the *Voting phase*:

- First, the coordinator C_i establishes his status and sends a Prepare T message to the network of nodes where the transaction T must be completed.
- When a slave node receives the Prepare T message, it determines whether to commit or cancel its component (or part) of T. If the

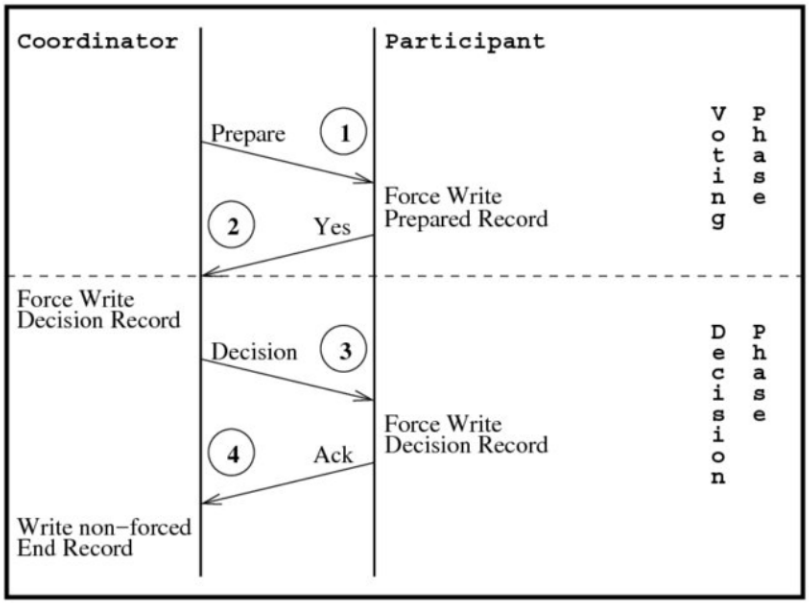

Figure 2.5: Two Phase Commit Protocol [3].

component has not yet completed its duty, the slave node may delay, but it must finally give a response.

- If the node does not want to commit, it sends an Abort T message to C_i to prevent the realisation of the route reservation.
- If the node wants to commit, it sends a Ready T message to C_i. Once the Ready T message has been sent to C_i, nothing can prevent it from committing its part of the transaction T except Coordinator C_i.

The *Decision phase* begins when the coordinator C_i receives the response Abort T or commit T from all of the nodes that are cooperatively executing the transition T. The node will be treated as if it had issued an Abort T if, for some reason, it fails to react after a reasonable timeout period has passed. During the *Decision phase*:

- If the coordinator receives Ready T from all the participating nodes of

T, then it decides to Commit T. Then, the coordinator sends a message Commit T to all the nodes involved in T.

- When a node receives a Commit T message, the component of T at that node is committed.
- If a node receives the message Abort T, it aborts T. The nodes that have already committed the transaction will attempt a roll back to the state in which they have to be able to receive a new request.
- If the coordinator receives an Abort T messages from one or more nodes, it sends an Abort T messages to all nodes involved in transaction T.

In this project, the database corresponds to the nodes of the railway interlocking with sequentially configuration corresponding a given route.
The interactions are the *request* and *response* messages propagated from the first node to the last itinerary and vice versa to reserve a route. The communication starts from the train (Fig. 2.6).

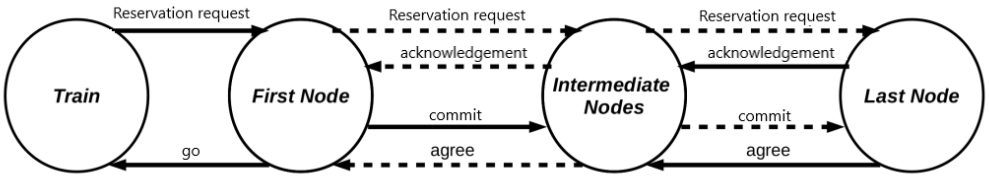

Figure 2.6: Two-Phase Commit sequential architecture; The train node starts the route reservation process sending a request to the first node, which will propagate the message until the last node. The last node will send back an Ack message to the first node, which will propagate the commit to the last node. The last node will send back an Agree message, that will reach the first node that will sent a *go* message to the train.

Each node receiving a commit will prepare to enforce the transaction (e.g. switching color for signals and switching the point's position to create another route). When the last node of the route has gotten the commit, it will send back an Agree message and once the coordinator receives it, it will communicate a *go* message to the train.

Here, the knowledge of the track layout is given to the train, which means that the nodes don't need to know the details except for neighbors and signals. Said that, the train is the element that starts all the conversation, knowing the list of nodes involved in the route and, because it is a sequential architecture, each node will only consider its neighbors for that route.

Sequential release

A sequential release mechanism can be implemented to increase the usage and utility of the given railway track system. The goal of such a mechanism is to make reserved nodes available for other trains to reserve and use as soon as the train that reserved them first no longer needs them.
When an individual track section reserved for a specific route detects the presence and then absence of a train, it returns to a state where it is available for new reservations. In railways, where a full free route in front of a train must be granted to allow safe train movement, not using sequential release reduces the availability of track elements, resulting in poorer network usage.

2.3 Model Checking

This formal verification approach assesses the correctness of a given system by thoroughly inspecting its state space and determining if the system satisfies a set of defined properties or requirements. These qualities are often

stated using temporal logic, allowing accurate description of the system's behavior over time.

The system's behavior and the possible states are created using a mathematical model that is cross-referenced with the desired properties to determine if the system meets the established requirements. If the model checker finds any violations, it provides a counter example to help developers identify the source of the problem and make the necessary changes.

The goal of model checking is (given an interpretation structure M and a formula ϕ) to check whether M $\models \phi$ (M satisfies ϕ).

It is important to introduce the concept of *Logic*. *Logic* refers to a formal method for reasoning about correctness, reliability and safety of a software system providing a framework to express, analyze and verify properties that a system must ensure. Logic establishes well-defined relationship between the elements and their behaviour over time. The different logics can be list as:

- Propositional logic or Propositional calculus: it deals with propositions (which can be true or false) and relations between propositions (foundation of First-order logic)
- Predicate logic or First-order logic: it uses quantified variables over non-logical objects, and allows the use of sentences that contain variables
- Linear tree logic (LTL): it is a modal temporal logic with modalities referring to time. It is possible to encode formulae about the future of paths
- Computational Tree logic (CTL): it is a branching-time logic, which means that its representation of time is a structure resembling a tree

in which the future is unknowable.

- High-order logic: It is similar to First-order logic but it has additional quantifiers and, sometimes, stronger semantics.

The *Logic* used from the framework UMC is the event-based and state-based logic Unified Computational Tree Logic (UCTL or CTL*) that combines features of both linear-time (LTL) and branching-time (CTL) temporal logics.

In CTL, starting from a state machine, it is possible to unfold it by obtaining an infinite tree of possible computations. The operators and quantifiers in CTL are listed below:

- temporal operator:
  - X: next state
  - F: future
  - G: globally
  - U: until
- path quantifier:
  - A: all paths
  - E: exists a path

As interpretation structure, Kripke [11] is used:

$$M :=< S, S_0, \rightarrow, AP, \mathbb{L} : S \rightarrow 2^{AP} >$$

where M is a tuple, S is a finite space of state, $S_0 \subset S$ is the finite space of initial state, $\rightarrow \subset S \text{x} S$ is the transition state function, AP is the space of atomic proposition and $L : S \rightarrow 2^{AP}$ is the state labeling function.

As a rule, CTL is more expressive than LTL, but it lacks the ability to express linear-time properties that are valid along a single execution path.

For this reason, UCTL overcomes this limitation by introducing additional operators that allows for the representation of both linear-time and branching-time properties within a signle logic. It may include new path quantifiers like "P" (i.e. for "some path" or "proportion of paths") and operators the combined the both properties.

This extended expressiveness makes UCTL (or CTL*) a more versatile tool for reasoning about complex system behaviors and relationships.

The *state-space explosion* issue in the context of model checking occurs when the possible configurations result in such a large number of states that it is difficult to process them. In fact, the previous CTL method is part of the explicit model checking techniques, as it generates the entire space of states.

To contrast the generation of the complete space of states before verifying the property as the traditional approach, on-the-fly model checking tackles this issue by interleaving the exploration of the state space with the verification process, thereby improving the efficiency and scalability of the model checking procedure.

On the fly model checking

It is a formal method verification methodology used to validate system against temporal logic specifications.
The on-the-fly approach offers several advantage:

- Early detection of property violation: if a violation is detected, the system can send an alarm immediately.

- Partial state space exploration: it is not necessary to explore the entire state space to verify a property but just to focus on the relevant parts.
- Memory efficiency: since not the entire state space system is built, this approach can be more memory-efficient and suitable for very large system.
- Incremental verification: the changes in the system or proprieties can be checked without regenerating the entire space.

Despite these benefits, on-the-fly model checking is not always the best solution for every system or property because it relies heavily on optimizations and heuristics to minimize the number of states that need to be explored. It can also introduce additional complexity and may not always guarantee the desired level of accuracy or efficiency.

In this work the properties are expressed in terms of abstractions as the Double Labelled Transition System (L2TS) that is elaborated in UMC and it shows only abstract labels (i.e. state and transition) defined by the abstractions.

In the next chapter we are going to explain what is UMC framework as well as its semantics and properties with the aim to introduce the model used to recreate the environment and the abstraction required to grant a correct behaviour.

Chapter 3

UMC Framework

How could we explore and exploit the advantage given by the on-the-fly approach for model checking and construction? How could we assess the suitability of the Unified Modeling Language (UML) methodology for the specification and verification of a system's dynamic behavior?
These questions brought the FMT (Formal Methods and Tools) Laboratory of ISTI-CNR to develop UMC [13], a verification framework for the definition, exploration, analysis and model checking of system designs represented as a set of state machines (UML). The model in UMC is composed by a textual description of a set of UML state chart diagrams (class), by a set of instantiated objects and by a set of abstraction rules (Fig. 3.1).

```
Class classname_1 is
  ...
end classname_1 ;
  ...
Class classname_n is
  ...
end classname_n ;

Objects
  objname_1: classname_1 ... ;
  objname_n: classname_n ... ;
  ....

Abstractions {
  Action ... -> ...
  State  ... -> ...
  ...
}
```

Figure 3.1: Structure of an UMC model.

The class declarations represent a template for the set of active or non-active objects of the system. If an object is active, the dynamic behavior of the object is described using the states and transitions associated to the class, while the non-active object only plays the role of an interface that can only receives signals. In the following sections, we will explain the parts of

the model and how they were adapted for the purpose of this work.

3.1 UMC class

An UMC class describes structure and behaviour of the object(Fig. 3.2).

```
Class classname is

 Signals
  ...                  -- list of asychronous events accepted by the class objects

 Operations
  ...                  -- list of synchronous operation calls accepted

 Vars
  ...                  -- list of local, private, attributes of the class objects

 State  ... = ...
 State  ... = ...      -- the structure of statechart nodes and subnodes
 State  ... = ...

  ... -> ... {...}
  ... -> ... {...}     -- the definition of the edges of the statechart
  ... -> ... {...}

end  classname;
```

Figure 3.2: Structure of an UMC class.

In UMC, there are two types of inactive predefined classes and objects. The *OUT* object can be used to send signals outside the system, while the *ERR* object, can be used to send the notification or error signals outside the system.

3.1.1 Signals and Operations

These are the list of asynchronous (Fig. 3.3) and synchronous (Fig. 3.4) events which trigger the transitions of the object.

```
-- signal with no parameters
signal_name;

-- signal with n parameters
signal_name (arg_1, ... , arg_n: type_n);
```

Figure 3.3: UMC signals.

```
-- operation with no parameters and no result value
op_name;

-- operation with n parameters and result type.
op_name (arg_1, ... , arg_n: type_n): result_type;
```

Figure 3.4: UMC operation.

The typing of the signal parameter or operation result is not strictly required from the compiler, but it is recommended to allow the static checking and to enable a meaningful display of the run time value.

3.1.2 Local variables

The member variables are defined under the keyword $Vars$.

By default, all the variables are private and local, which means that they can not be accessible by other objects and can not be shared among all the objects. When the keyword *Priority* is declare, the object will respect a

```
-- untyped, initialized by default, multiple local variable
varname1; varname2; varname3;

-- expicitly typed local variable
varname: int;

-- explicilty typed, expliciltly initialized, local array
varname1: int[] := [];
```

Figure 3.5: Syntax of UMC variables.

hierarchy for the order in which messages are sent, If it is declared in all the active classes as first variable, then the possible system evolution takes in account only objects with highest priority value.

3.1.3 States

The set of nodes that defines the state chart is described by a set of rules (*State*).

```
State parentstate = substate_1, substate_2, ... , substate_n
```

Figure 3.6: Syntax of UMC state.

A composite sequential state acts as a container for a nested state chart diagram. As shown in Fig. 3.6, the top-level state includes all the sub-states in which the object can be found. Additionally, a composite includes also an implicit internal region that includes all the required sub-state but, in UMC, this region is never explicitly shown. In this composite, the first sub-state becomes the initial state unless the keyword *initial* is explicitly provided in one of the sub-state.

It can happen that there is the necessity to postpone certain events(matching signals or operation) when the object is in certain sub-state. The keyword $Defers$ is used for this purpose with only one requirement: the event declared must match exactly the event name (Fig. 3.7).

```
State statename Defers event_1 , ... , event_n(arg1,...,argn)
```

Figure 3.7: Syntax of state with deferred events.

3.1.4 Transitions

The edges of the state chart are defined by a set of rules or *Transition*. To transit from the *source* state to the *target* state, firstly an event has to be triggered. Secondly, the *guard* condition has to be verified and finally, the *action* code , a sequence of basic actions, is executed when the transition is triggered and verified.

source **->** *target* **{** *trigger* **[** *guard* **]** **/** *actions* **}**

Figure 3.8: Syntax of UMC transition.

```
s1 -> s2      -- a very simple completion transition with no guards nor effects

s1 -> s2
   { myopcall(x,y)[_caller=obj1 and x>0] /
     v1:=x;     -- a full transition with trigger, guards and actions
   }
```

Figure 3.9: Example of UMC transitions.

Fig. 3.8 and Fig. 3.9 show the syntax of a transition and two simple examples, respectively. The *source* and *target* are states or list of states, since there is the possibility of executing a *join* transition in case of multiple sources or a *fork* transition in case of multiple targets. It is possible that the same *transition* is triggered from different event, which can be seen as an overload of the transition.

The *trigger* is constituted by an event name and possibly its sequence of formal parameters declared in Signals and Operations sections. Using the "-" symbol, the transition triggering event is declared absent and the object will try to transit from *source* to *target* at every step (creating a new edge of the state chart). The *guard* is a boolean expression involving trigger variables and it is optional. Lastly, *actions* can be either empty or actions such as

variable assignment, signal sending, finite loops and operation calls.

Assignment

```
varname := right_side;
varname[index] := right_side;
```

Figure 3.10: Syntax of UMC assignment.

In Fig. 3.10, the *right_side* can be an expression or a call of operation, while the *varname* has to be the name of a local variable or a parameter of the transition trigger or local transition variable. In fact, it is possible to declare temporary variable inside the *actions* (Fig. 3.11).

```
s1 -> s2 {-/ tmp:bool:= b1 and b2; b3:= tmp or b3 }
```

Figure 3.11: UMC transition with temporary variable and assignment action.

Asynchronous and synchronous sending

The execution of an asynchronous action causes the routing of the signal message in the event queue of the target object.

```
target_object.signal_name(expr_1,...,expr_n);
```

Figure 3.12: Syntax of UMC asynchornous.

Looking at Fig. 3.12, the *target_object* must be a name of an allocated object (variable, parameter or global static object). The *signal_name* instead, has to be declared in the *Signal:* section of target object. If the *target_name* is not specified, the *self* keyword will be implicitly assumed.

In case the name is not declared, it will cause a system error sending the *Runtime_error* signal to ERR object.

```
target_object.operation_name(expr_1,...,expr_n);
varname := target_object.operation_name(expr_1,...,expr_n);
```

Figure 3.13: UMC transition with temporary variable and assignment action.

The operation call action is used to achieve a synchronous action and it involves the identification of the *target_object*, the operation name and list arguments if necessary (Fig. 3.13).

The *operation_name* is one of the names declared in the *Operation:* section and the number of passed parameters has to match, e.g. req(sender: obj, id_itinerary: int). The parameters are passed by value to the transition, which means that all the changes are not reflected out of the transition. The value returned from an operation call can be assign to a variable to keep track of value received and to use it in other calls. If the operation call does not have a return action, a final implicit return action will be executed. Otherwise, the return causes the sending of an *operation_return* action to the caller.

Fig. 3.14 shows same examples.

```
c1 -> c2 { - / server.write_operation(123)}
s1 -> s2 {write_operation(val) / var := val}
s1 -> s2 {write_operation(val) / var := val; return}
c1 -> c2 { - / var := server.read_var_operation}
s1 -> s2 {read_var_operation / return(var)}
```

Figure 3.14: 1) Operation call with no return value, 2) operation executed with implicit return action, 3) operation call executed with explicit return action, 4) function call and 5) function call execution with explicit return action

Composite actions

Composite actions are typically conditional if they involve an *IF* clause (Fig. 3.15) and finite(Fig. 3.16) if they involve a *FOR* loops.

```
if condition then { actions-list } else { actions-list };
if condition then { actions-list };
```

Figure 3.15: if condition UMC: *condition* is a boolean expression and $action - list$ are sequence of events.

```
for iterator in min_expre .. max_expr{ actions-list };
```

Figure 3.16: for loop UMC: *iterator* acts as variable initialized with min_expre and and every cycle after executing the $action - list$, it increases until max_expre.

3.2 UMC Objects

Once the necessary classes have been declared, we can start defining the deployed system as a set of instantiated objects.

```
object_name: class_name
        (obj_attribute_1 => initial_value_1,
                ...,
         obj_attribute_n => initial_value_n);

object_name: class_name;
```

Figure 3.17: Syntax of UMC object. Up: object declaration with initialization. Down: the second without.

All the objects have to be defined under the *Objects:* section . Each declared object introduce into the system, the object name, the class name and

optional specific initial values for some its parameters. After the declaration, the object starts acting as global variable and it is visible inside all the class declarations.

3.3 UMC Abstraction

Abstractions have no part in the definition of the system's dynamic behavior. In fact, their function is to specify what the user wants to see as well as to verify the properties of the model. All the rules in Abstraction section are taken in account and applied where necessary. The rules are divided into two categories: action rules and state rules.

Action rules

The action rules assist to define the event we wish to monitor when a transition occurs.

```
Action: source_obj:target_obj.event(arg_1,...,arg_n) -> main_label(flag,,..., flag)
```

Figure 3.18: UMC action abstraction.

In Fig. 3.18, the source and target object must be declared and defined, the events are signals or operation and arg_i is its parameter.

$$Action : lostevent(e, *) -> discarded\_message(e, *)$$

This is an important action: *lostevent(e,*)* is sent when an event message is lost and *discarded_ message(e,*)* is the label in the state chart. Especially, it is used to observe when an event is removed by the events queue of an object or discarded because there is no enabled transition which can start.

$$Action : obj : Runtime\_Error -> Design\_Error(obj)$$

As previously said, an error message could be generated (*Runtime_Error* and sent to the ERR object labelling the state of the chart as *Design_Error*. There are other important actions, such as assigning values or highlighting the transition call, but they are out of scope of this work.

State rules

The state rules allow to define which structural aspect the system the user wants to observe.

```
State:
      ground_state_predicate_pattern and
            ...
      ground_state_predicate_pattern   -> main_label(flag,...,flag)
```

Figure 3.19: UMC state abstraction.

Fig. 3.19 shows the syntax of the state abstraction. In this case, it is allowed to conjugate *ground_state_predicate_patter* and it could be a predicate or local variables of an active object. This example in Fig. 3.20 will help you to understand:

```
State:  obj2.speed < 30  -> obj2_slow
```

Figure 3.20: Example of state rule.

Here, if the variable *speed* of obj2 is less the 30, in the state chart will be dislayed *obj2_slow*. It is possible having multiple labels in a configuration of the state chart.

3.4 Proposed configuration

The configuration proposed in this work starts from a model which is already present inside the framework UMC. As mentioned in 2.2.1, the railway interlocking is composed by three main classes: linear, point and signal.

3.4.1 Linear class

Linear present eight signals: *req(sender:obj,id_itinerary:int)*, *ack(...)*, *nack(...)*, *commit(...)*, *agree(...)*, *disagree(...)*, *green(id_itinerary:int)* and *red(...)*.

SensorOn(sender:obj,id_itinerary:int) and *sensorOff(sender:obj,id_itinerary:int)* are the two operations.

The variables **next**, **prev** and **sign** are vectors of objects as long as the number of routes, while the variable **train** is a an object.

The possible states into which the object can transit are listed below (the guards are describes in Fig. 4.10):

- **NOT_RESERVED**: Initial state of all the linear objects. It can transit to state **WAIT_ACK** receiving a trigger *req(...)* event or it can transit to state **WAIT_COMMIT**. No chancing state means a *nack(...)* event is received.

- **WAIT_ACK**: it can transit to state **WAIT_COMMIT** sending an *ack(...)* event to the element in **prev** or to **WAIT_AGREE** sending a *commit(...)* event to the next element. No state change if it receives a *req(...)* event and sends back a *nack(...)* event, while it transit back to **NOT_RESERVED** when there is a *disagree(...)* event.

- **WAIT_COMMIT**: receiving a *commit(...)* event and passing the various guards, it can transit to **WAIT_AGREE** receiving *com-*

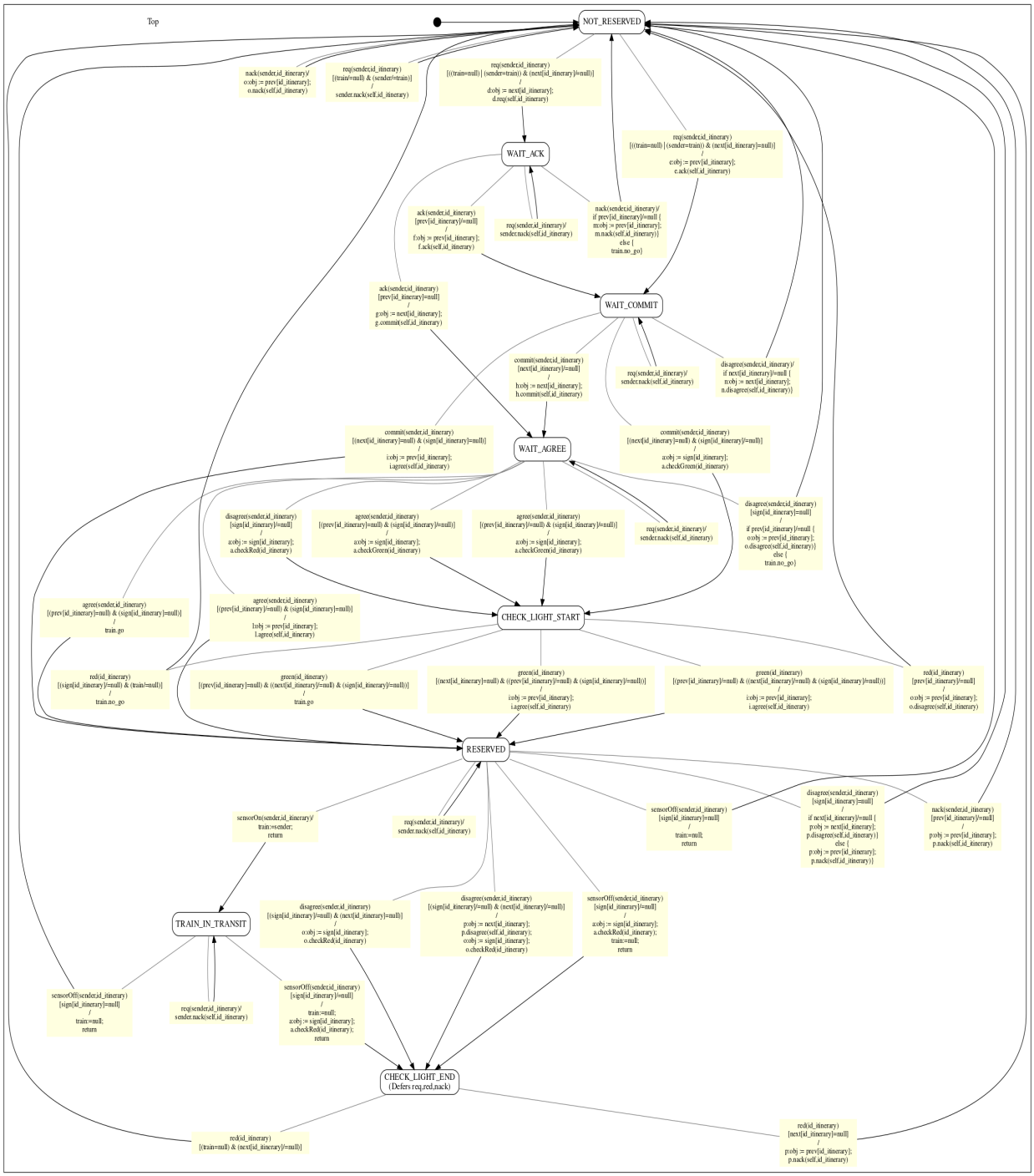

Figure 3.21: Class Linear Model.

mit(...) event, to **RESERVED** sending an *agree(...)* event, to **CHECK_LIGHT_START** sending an *agree(...)* event to the signal and can transit back to **NOT_RESERVED** receiving a *disagree(...)* event.

- **WAIT_AGREE**: receiving an *agree(...)* or a *disagree(...)* event and passing the different guards, the state can transit to **CHECK_LIGHT_START** sending a *checkgreen(...)* or *checkred(...)* event to the signal, if present. Otherwise, it can transit directly to the state **RESERVED** sending an *agree(...)* event to the element in **prev** or in case the element in **prev** is *null* as well as the signal, it sends a *go* operation to the train.

- **CHECK_LIGHT_START**: Receiving a *green(...)* event, it transit to **RESERVED** sending an *agree(...)* event to the element in **prev** or a *go* operation to the train. If a *red(...)* event is received, it transit to **NOT_RESERVED** sending a *disgree(...)* event to the element in prev or a *no_go* operation to the train.

- **RESERVED**: this state means that the linear is reserved for the train. It transit to **TRAIN_IN_TRANSIT** receiving a *sensorOn(...)* operation assigning the train to the linear, to **CHECK_LIGHT_END** receiving *sensorOff(...)* operation and sending *checkred(...)* event to the signal or *disagree(...)* event sending *disagree(...)* event to the element in **next**. It transist back to **NON_RESERVED** receiving a *sensorOff(...)* operation or *disagree(...)* event without any signals. No state change instead, receiving *req* event.

- **TRAIN_IN_TRANSIT**: the train is passing by this element. Receiving *sensorOff(...)* operation if signals *not null*, it can transit to

CHECK_LIGHT_END sending a *checkred(...)* event, otherwise it transit back to **NON_RESERVED** setting the variable train to *null*. No state change receiving a *req* event.

- **CHECK_LIGHT_END**: This state check the state of the signal and defers three event: *req(...)*, *red(...)* and *nack(...)*. We can receive a *red* event and sending a *nack(...)* event to go back to **NOT_RESERVED**.

3.4.2 Point class

The point class/object is very important since it is a routing element and it can cause incidents or derails if malfunctioning. This class has the same *Signals:* and *Operations:* section of linear class, without *green(...)* and *red(...)*. The variables **next**, **prev** and **conf** are vectors of objects while, **reverse** is boolean, **itinerary** an integer and **train** an object initialized to *null*.

The states are described below and the guards in Fig. 3.22

- **NOT_RESERVED**: Initial state for all the points initialized. It can transit to **WAIT_ACK** receiving a *req(...)* event propagating it to the follow object. No state change receiving a *nack(...)* event.

- **WAIT_ACK**: In this state we are deferring *req(...)* and *nack(...)* event. Receiving an *ack(...)* event, it can transit to **WAIT_COMMIT** propagating the *ack(...)* event to object in **next**. In case of *nack(...)* event received, it will transit back to **NOT_RESERVED**.

- **WAIT_COMMIT**: A *commit(...)* event will trigger the transition to **WAIT_AGREE**. A *disagree(...)* event will force to transit back to **NOT_RESERVED**.

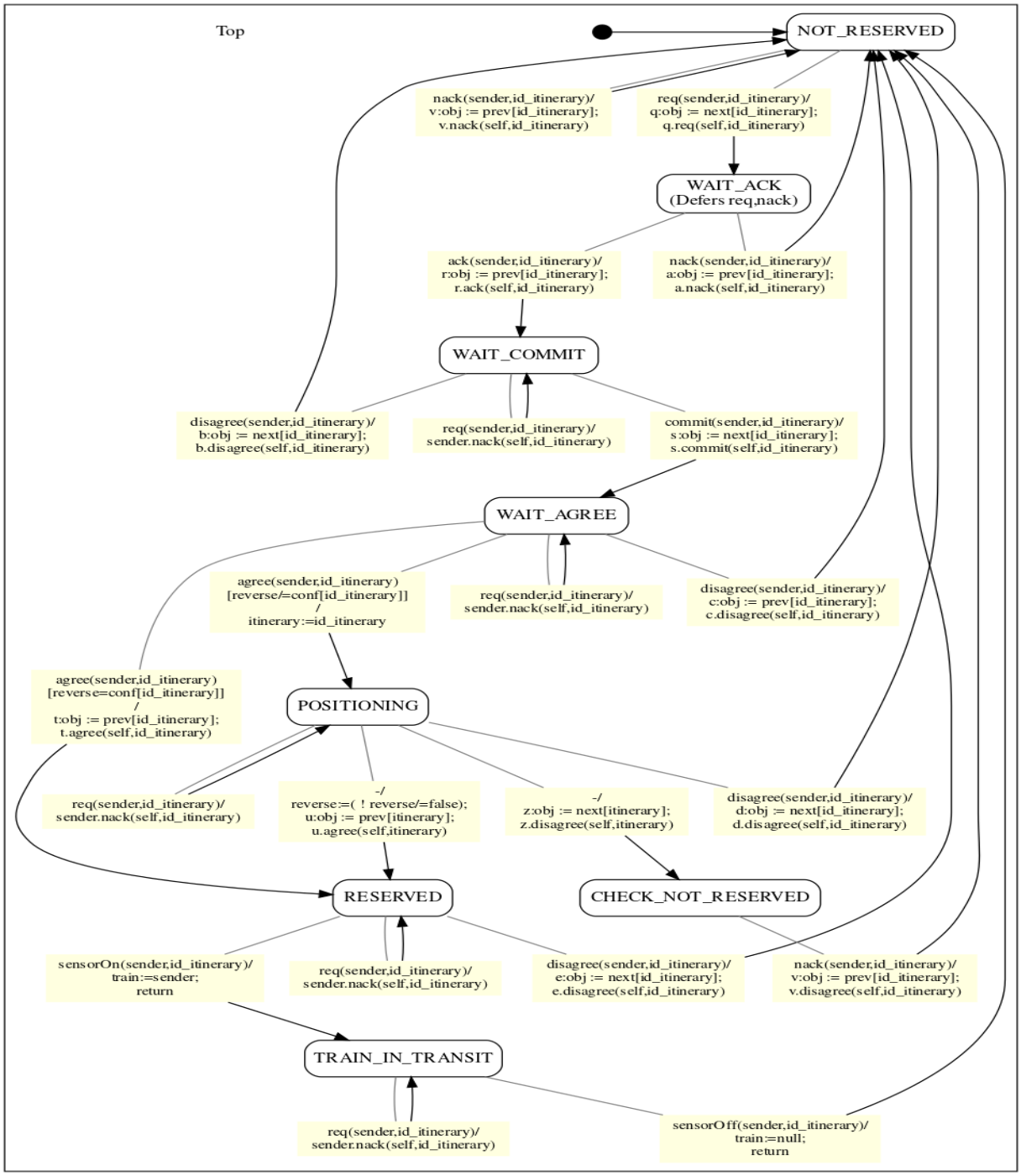

Figure 3.22: Class Point Model.

- **WAIT_AGREE**: From this state we can transit to **POSITIONING** receiving an *agree* message (in case of reverse value is different the itinerary configuration) and to **RESERVED** (in the other case). The*disagree(...)* event will trigger the transit back to **NOT_RESERVED**.

- **POSITIONING**: The reverse has to be in the right position in order to respect the itinerary configuration. In this work, no constrains have been created to go to **RESERVED** or **CHECK_NOT_RESERVED** since we wanted to verify both cases. No state change with a *req(...)* event and back to the initial state in case of *disagree(...)* event.

- **CHECK_NOT_RESERVED**: It is the state which forbids the positioning and transits back to **NOT_RESERVED**.

- **RESERVED**: When a *sensorOn(...)* operation is received, it will transit to the state **TRAIN_IN_TRANSIT**. Instead, It will transit back to **NOT_RESERVED** receiving a *disagree(...)* event.

- **TRAIN_IN_TRANSIT**: Final state that indicates the train crossing. It will transit to **NOT_RESERVED** receiving a *sensorOff(...)* operation.

3.4.3 Signal class

The signal class represent a marker board in railway terminology and it is introduced to recreate a system similar to the existing one. The signal class/object (not to be confused with *Signals:* section) has just two signals: *checkgreen(id_itinerary)* and *checkred(...)* that trigger *green(...)* and *red(...)* events of the linear.

The two variables available in this class are **linear** (object) and **red** (boolean). During the **WAIT_AGREE**, each signal that is on a route is commanded to **GREEN**. In the event of a signal failure, it does not per puts the continuation, but returns to the initial state. As soon as the train that had booked the relevant itinerary has passed, the variable **red** of the signal object will turns *true*, which means that the element is available.

Since this class exists only if linear objects have signal objects, its complexity is reduced. The two state are listed below and the guards in Fig. 3.23

- **RED**: Initial state of signal objects. It transits to **GREEN** when receives a *checkgreen(...)* event. No state change receiving a *checkred(...)* event.
- **GREEN**: It means that the linear object associated is free and booked for that train. It transits back to **RED** when receiving a *checkred(...)* event.

3.4.4 Train class

It represents the nodes that run along the itinerary. The algorithm starts with the train sending a *req(...)* event to the first node of his itinerary. The signals *go* and *no_go* in this class are used to tell the train whether to leave or not.

id_itinerary and **position** are integer variables, **node** is an object and **route_node** a vector of objects. The states are **READY**, **WAIT_OK**, **MOVE** and **STOP** and the guards to transit between states are shown in Fig. 3.24.

Respectively:

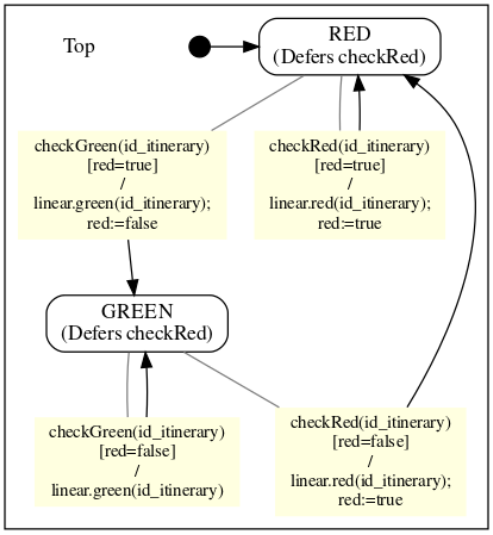

Figure 3.23: Class Signal Model.

- **READY**: Initial state. It can transit to **WAIT_OK** sending a request to the first node of "route_nodes", that is the same where the train is.
- **WAIT_OK**: The train is waiting either the *go* event to transit in **MOVE** or *no_go* event to transit back in **READY**.
- **MOVE**: The train starts to move and it will do it until the guards is respected
- **STOP**: Final state which means the train arrived without problems at the end of its itinerary.

The allocation of these objects will be discussed in the next chapter. As an example, we will show the configuration of one route.

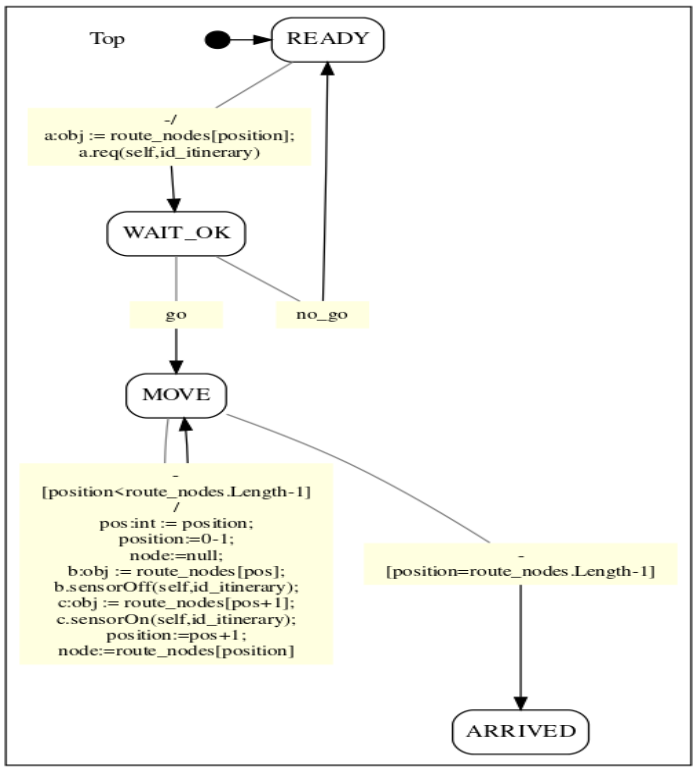

Figure 3.24: Class Train Model.

3.4.5 Abstractions

The abstractions used in this work can be divided in three actions for all the cases and as many states as there are trains, signals and points:

1. $Action : \$1(\$*) \rightarrow \$1(\$*)$
2. $Action : \$obj : Runtime\_Error -> Design\_Error(\$obj)$
3. $Action : lostevent(\$e, \$*) -> Discarded\_message(\$e, \$*)$

The first is a default action and it is used to show the which actions brings to the next state (configuration). The second one and the third one, as described in section 3.3, are checks for discarded messages and to raise errors.

The system has the goal to ensure certain properties and the state abstractions(Fig. 3.25) are defined based on:

- No derails: every time a point is reserved, it has to be configured according to the selected route. If it does not happen and the opposite configuration is set up, we will encounter an accident or derail.
- Possible derails: if a point is not in the route but it is the next or prev element of the first or last element, we will keep in account them as possible derails in case the configuration is wrong.
- No accident: this is only created when there are at least two train. It checks that two trains are never in the same linear.
- Signals fault: if the linear objects have signal objects attached, this abstraction is created. It checks that the signals is red (i.e. variable red is true) when the train is passing in that linear's signal object.

```
Abstractions{
        Action $1($*) -> $1($*)
        Action: $obj:Runtime_Error -> Design_Error($obj)
        Action: lostevent($e,$*) -> discarded_message($e,$*)
        State : train1.node == _042 -> train1_arrived
        State : _42M.conf[0] == false -> POSSIBLE_DERAIL
        State : _41BM.conf[0] == false -> POSSIBLE_DERAIL
        State : inState(train1.MOVE) and _AI042.red == true and _A942.train == train1 -> FAULT__AI042
        State : inState(train1.MOVE) and _QM.red == true and _042.train == train1 -> FAULT__QM

}
```

Figure 3.25: Example of route abstraction

The last but not the least property that we verify in this work is the **Stabilization**. It means that there is always a computation that leads the train to the end points of each itinerary request made by a train stationed at the beginning of the itinerary, and all other computations (due to errors or other causes) result in an abort of the request. In the next chapter we will

explain the tool used to convert the railway XML into UMC model giving also global information of the network layout and interlocking.

Chapter 4

ParserXML: from XML to UMC

The main scope of this work, in addition to formal verification of the proposed model, is to implement a tool capable of parsing railway XML file into UMC class giving also, in a more readable way, all the details regarding the composition of the network layout and all the routes described in the XML file.

An initial work has been carried on by Lindberg and Hansen from University of Denmark and it consisted of a parser for Linh's XML format to enable the processing of system models [9].

The code provided to us was a tool capable to parser railway XML file into file .txt that display the complete structure of the network layout and interlocking (code written in Java).

We thought we should have a tool that would combine this parser, improving it in the aspect of information provided on structure and path, and the model checking provided by UMC. While parsing the file, we also give the possibility to write the UMC code related to the chosen features, that we will show later.

4.1 Railway XML

Text-based markup language called XML (eXtensible Markup Language) offers a flexible and organized approach to express data. It is both human and machine readable and it is structured as a hierarchical tree of elements.

The railway XML used in this work consist of a main root called "interlocking" that is divided in "network" and "routetable".

4.1.1 Network

The network element has an unique identifier *id* and contains two kind of sub-element called **trackSection** and **markerboard**.

Track Section

The track section has an identifier **id**, a **type** (linear or point) and a **length** parameter that we will not take in account. Each track section can have from one to three neighbor sub-elements according to the railway configuration. The neighbor element gives info about who is the neighbor and in which side (in case of type linear, it will be *up* or *down* and in case of type point, *stem*, *plus* and *minus*).

```
▼<trackSection id="37M" length="100" type="point">
    <neighbor ref="546" side="plus"/>
    <neighbor ref="548" side="minus"/>
    <neighbor ref="A931" side="stem"/>
  </trackSection>
▼<trackSection id="551" length="100" type="linear">
    <neighbor ref="36M" side="down"/>
    <neighbor ref="A952" side="up"/>
  </trackSection>
```

Figure 4.1: Example of track sections.

Markerboard

The markerboard represents the class Signal implemented in configuration proposed (section 3.4.3). It has an identifier **id**, a variable **mounted** the indicates the direction in which it works (*up* or *down*), a variable **distance** that we will not keep in account and a variable **track** that is the identifier of the linear object to which the signal object is attached.

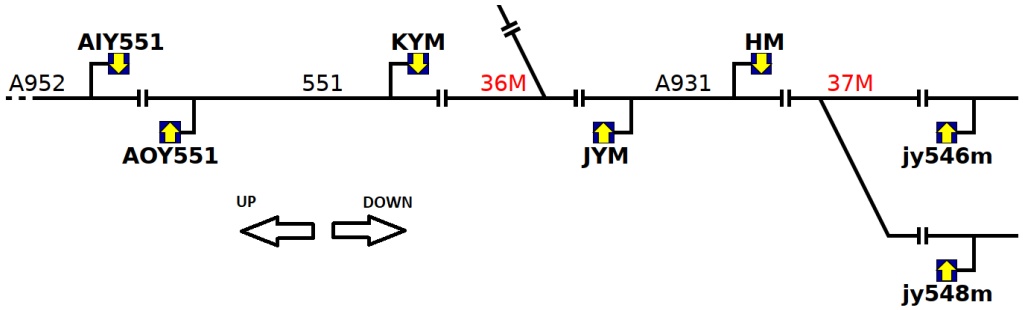

Figure 4.2: Graphical layout of the linear **551** and the point **37M** of the XML file above.

```
<markerboard distance="20.0" id="QM" mounted="down" track="042"/>
<markerboard distance="20.0" id="QXM" mounted="down" track="041"/>
<markerboard distance="20.0" id="AO041" mounted="up" track="041"/>
<markerboard distance="20.0" id="AOX042" mounted="up" track="042"/>
```

Figure 4.3: Example of markerboard sections.

4.1.2 RouteTable

The route table contains all the route that are implemented in the network. It has an identifier **id** and a variable **network** referring to the previous network declared. A route is identified with:

- an **id**: the route's identifier
- a **source**: it indicates first the signal of the linear in which the train is allocated
- a **destination**: it indicates the signal of the last linear
- a **direction**: it indicates the path the train have to follow

Each route in Fig. 4.4, has several conditions that show us information depending from the type of it:

- *point*: it means that a point object is into the route and has to be configured with the value in *plus* (variable **conf** assign to *true*)

- *signal*: it mean that the signal object and the linear object correlated could be required in this route. To recognize them, we performed a double check with the ones involved into the path. The variable **ref** contains the **id** of the signal object.

- *trackvacancy*: this condition indicates that, in addition to **source**, which other element is in the route.

- *mutualblocking*: it says which other routes has at least one element in common creating a conflict. The variable **ref** indicates the **id** of the route overlapping.

```
▼<route id="r_13" source="HM" destination="j546m" dir="down">
   <condition type="point" val="plus" ref="37M"/>
   <condition type="signal" ref="AIY546"/>
   <condition type="signal" ref="jy546m"/>
   <condition type="trackvacancy" ref="37M"/>
   <condition type="trackvacancy" ref="546"/>
   <condition type="mutualblocking" ref="r_06"/>
   <condition type="mutualblocking" ref="r_07"/>
   <condition type="mutualblocking" ref="r_14"/>
   <condition type="mutualblocking" ref="r_38"/>
   <condition type="mutualblocking" ref="r_39"/>
 </route>
```

Figure 4.4: Example of XML route.

4.2 Implementation

The tool presents an user-friendly GUI (Graphic User Interface) that allows the user either to select the preferred configuration to extract(e.g. one train or two train) the overview and the corresponding UMC code, as described in chapter 3, or the full overview of railway network and interlocking. A system of sub-folders has been created to manage the corresponding .txt file and to achieve a clear organisation of the work starting from the name of XML file, as shown in Fig. 4.5.

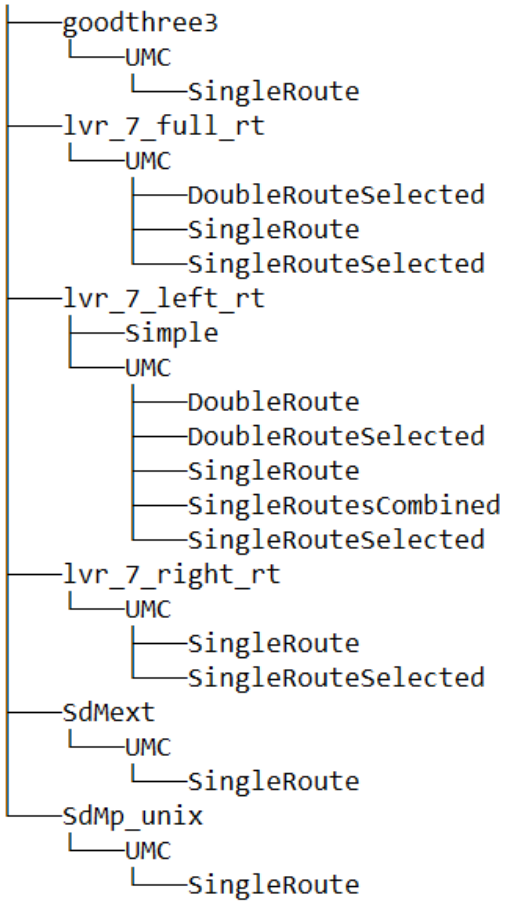

Figure 4.5: Directories tree created.

ParserXML class

The core of the tool resides in the *ParserXML* class. Every times the function *run(...)* is called, this object is created and at the end of his execution, automatically destroyed thank to the help of smart pointers.

Its main tasks are:

- to parse [8] the XML truck section into a C++ class, respectively Linear, Point and Signals.
- To create *NetworkLayout* and *Interlocking* objects.

These two objects belong to *parserXML*: the first one has every details about each element of the railway (e.g neighbors, directions, configuration), while the second one knows about every route (e.g. what signal or point is involved, what conflicts or overlap it could encounter).

The second most important class is *writer*. It is implemented using the singleton patter. It work as an interface and the two classes that inherit it are *writerUMC* and *writerSimple*. When the function *writeFile* is called from one of this object, the *parserXML* object is passed to this function to write the text.

WriterUMC class

In case of *writerUMC*, we are going to create a .txt file following these possible configurations:

- one route: this setup create for each route, a file .txt containing the network of the route, the interlocking of the route and the UMC code with the object and abstraction related to the route.
- one route combined: all the routes with less than three elements are merged with another one, the first one found between the routes that continue the route (last element of the first route and the first element of the second route has to match). This setup creates only the routes that have been merged to create longer paths.
- one route selected: It creates the overview of just this route and its UMC code.
- two routes: it is similar to "one route" configuration but with the addition of a second train. The route chosen are those that do not conflict or overlap.

- two routes selected: same concept of "one route selected" but in this case, we have two routes to select. It is not allowed to select the same route twice.

The logic code about each UMC class implemented in chapter 3, has been saved into a string value called *codeUMC*, a parameter of this class.

```
/* NetworkLayout */

value
maxLinear: Int = 20,
maxPoint: Int = 7,
maxMb: Int = 31
axiom
pointStem[_42M - 0] = _41AM - 1,
pointPlus[_42M - 0] = _042 - 10,
pointMinus[_42M - 0] = _556 - 7,
pointStem[_41AM - 1] = _42M - 0,
pointPlus[_41AM - 1] = _542 - 14,
pointMinus[_41AM - 1] = _41BM - 2,
pointStem[_41BM - 2] = _40M - 3,
pointPlus[_41BM - 2] = _041 - 11,
pointMinus[_41BM - 2] = _41AM - 1,
pointStem[_40M - 3] = _41BM - 2,
pointPlus[_40M - 3] = _543 - 15,
pointMinus[_40M - 3] = _38M - 4,
pointStem[_38M - 4] = _40M - 3,
pointPlus[_38M - 4] = _544 - 18,
pointMinus[_38M - 4] = _36M - 5,
pointStem[_36M - 5] = _A931 - 22,
pointPlus[_36M - 5] = _38M - 4,
pointMinus[_36M - 5] = _551 - 20,
pointStem[_37M - 6] = _A931 - 22,
pointPlus[_37M - 6] = _546 - 23,
pointMinus[_37M - 6] = _548 - 24,

linearUp[_556 - 7] = _557 - 8,
linearDown[_556 - 7] = _42M - 0,
linearUp[_557 - 8] = _A951 - 9,
linearDown[_557 - 8] = _556 - 7,
linearUp[_A951 - 9] = null - -1,
```

Figure 4.6: Part of Network-Layout overview.

```
/* Interlocking */

value
maxRoutes: Int = 39,
maxPathLength: Int = 8,
maxChunks: Int = 7
axiom
routeSrc[0] = 13
routeDirection[0] = down,
routeDest[0] = 10,
routePoints[0] = { PLUS, INTER, PLUS, INTER, INTER,
INTER, INTER },
routePath[0] = { 13, 10, -1, -1, -1, -1, -1, -1 },
routeSignal[0] = { false, false, false, false, true,
false, false, true, true, false, false, false, false,
true, false, false, false, false, false, false,
false, false, false, false, false, false, false,
false, false, false, false },
routeOverlap[0] = { false, false, false, false,
false, false, false, false, false, false, false,
false, false, false, false, false, false, false,
false, false, false, false, false, false, false,
false, false },
routeConflicts[0, 0] = false,
routeConflicts[0, 1] = false,
routeConflicts[0, 2] = false,
routeConflicts[0, 3] = false,
routeConflicts[0, 4] = false,
routeConflicts[0, 5] = false,
routeConflicts[0, 6] = false,
routeConflicts[0, 7] = false,
routeConflicts[0, 8] = false,
routeConflicts[0, 9] = false,
routeConflicts[0, 10] = true,
routeConflicts[0, 11] = true,
```

Figure 4.7: Part of complete Interlocking overview.

WriterSimple class

This complete overview is the work we got from [9]. At the beginning, every element of the railway were label with just an numerical identifier, so it was difficult to understand or keep track of each element. What we did, it

is to map every identifier with its real name to be able to understand better the architecture and compare with the graphic visualization implemented. We also adjust few details regarding the interlocking part.

In Fig. 4.6, we can see we how many linears, points and signal we actually have in this railway XML. In Fig. 4.7, the number of route, the length of the longest path and the max number of points is reported. For each route, we see the configuration of the points ("INTER" means that the point is not taken in account), which signal has to be activated and all the conflicts.

4.3 User interface

Creating a user interface for a parser can greatly enhance its usability, accessibility, and efficiency, especially for users who are not familiar with command-line interfaces or prefer a more visual interaction.
To achieve this result, we used WxWidgets, a cross-platform framework for creating graphical user interfaces in C++ [2].

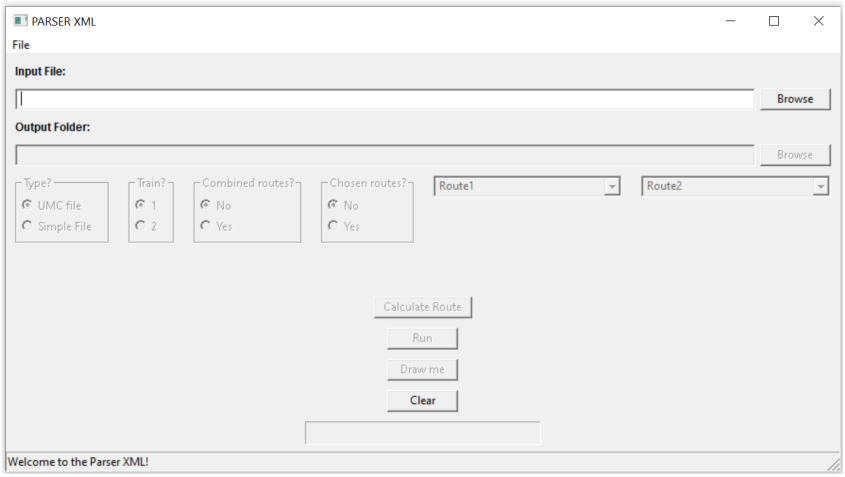

Figure 4.8: GUI of ParserXML tool.

Fig. 4.8 is shown the user interface.

The only field available is *Input File*, where the user can insert an XML file or, if needed, browse the folders looking for it - in case of different extension, an error is raised with a window pop-up. If the file is correct, the field *Output Folder* will start working. Here, the user has to select a directory where the new folder will be created.

When both fields are filled, almost all the function are activeted.

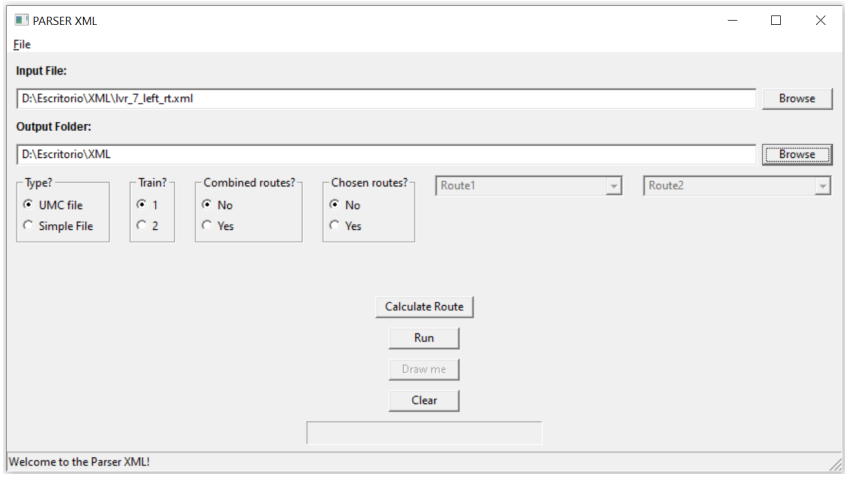

Figure 4.9: GUI of ParserXML tool filled.

Let's move on to the description of these elements:

- radiobox *Type*: the user decides which kind of file wants (i.e. *writerUMC* or *writerSimple*).
- radiobox *Train*: configuration with one or two trains. The configuration with two trains does not allow combined routes.
- radiobox *Combined Routes*: when it is set up to yes, deactivate the radiobox *Chosen Route* and it will combine all the routes with length less then three.

- radiobox *Chosen Routes*: allow the user to select the route on the lists. If only one train is selected, just the first list will be activated. The first time the user needs to visualize the route, he has to press the button *Calculate Route*.

- list route one: it allows the user to select the first route.

- list route two: it allows the user to select the second route.

- button *Calculate Route*: when pressed it fills the routes lists counting how many routes are in the taken in account in the XML. It is required when the user want to select the routes.

- button *Run*: when it is pressed, it starts and completes all the processes, in particular parsing, mapping and writing. At this moment, all the file .txt of the configuration predefined are created following the system of sub-directories in Fig. 4.5. A pop-up message will appear in case the operation is successfully completed. An important action is the creation of a json file [10], containing the information to create the approximate representation.

- button *Draw Me*: this button will launch a script that recreate the graphic visualization of the railway layout that we will explain the the next section.

- button *Clean*: when pressed it clean all the fields as if it had just started.

The are some pop-up message that are displayed when certain action is well executed or not.

4.4 Graphic visualization

The main issue encountered in trying to create an approximate representation of the railway map, was the lack of geographic coordinates inside the XML file and the different positions that a point can assume (e.g. *stem* on the left and *minus* branch above the *plus* branch; stem on the left and *minus* below the *plus* branch. To avoid this issue, we implemented a dual graph where each section is represented by a node and each joint is represented by an edge. Among the nodes, points can be easily distinguished by linears. In fact, the former have three edges while the latter have two. The edges represents the connection between them.

However, this approximate visual representation makes it possible to correctly represent the connections between the nodes of a distributed system, that is the structure (which it could be logic with of UMC or even physical with a possible implementation) of the railway network as shown in Fig. 4.10 In order to do that, we integrated to our C++ code, a python script giving in input the json file, the name of the file, route one and route two (they are optional).

Every time the button *Draw Me* is pressed, the script is launched creating the plot of the image. Every graph drawn is saved into the folder *fig* of the project. To give a more intuitive representation:

- in every point we labelled the edges displaying if it is a *plus*, *minus* or *stem*.
- the graph will highlight the routes when selected in the GUI.
- in the left corner of the image there is the table of signals. The left column indicates the linear objects and the right one, the *down* and the *up* signal objects.

To achieve this result, we used the Networkx [7], which is a Python package for the creation, manipulation, and study of complex networks, or graphs, including both directed and undirected graphs. It provides tools for working with graphs as mathematical objects, and also for drawing and visualizing graphs. Notice that, sometimes the label might be reversed due to the reading of the points in the .json file.

Linears	Signals
556	D: NYM U:LYM
557	D: LZM U:AOY557
A951	D: AIY557 U:null
042	D: QM U:AOX042
041	D: QXM U:AO041
A941	D: AIX041 U:null
A942	D: AI042 U:null
542	D: FM U:PXM
543	D: FXM U:PM
096	D: AO096 U:DXM
095	D: AOX095 U:DM
A976	D: null U:AIX096
A975	D: null U:AI095
544	D: GM U:GYM
551	D: KYM U:AOY551
A952	D: AIY551 U:null
A931	D: HM U:JYM
A911	D: JM U:HYM
545	D: j545m U:jy545m
546	D: j546m U:jy546m
547	D: j547m U:jy547m
548	D: j548m U:jy548m
A965	D: AI545 U:null
A966	D: null U:AIY546
A967	D: AI547 U:null
A968	D: null U:AIY548

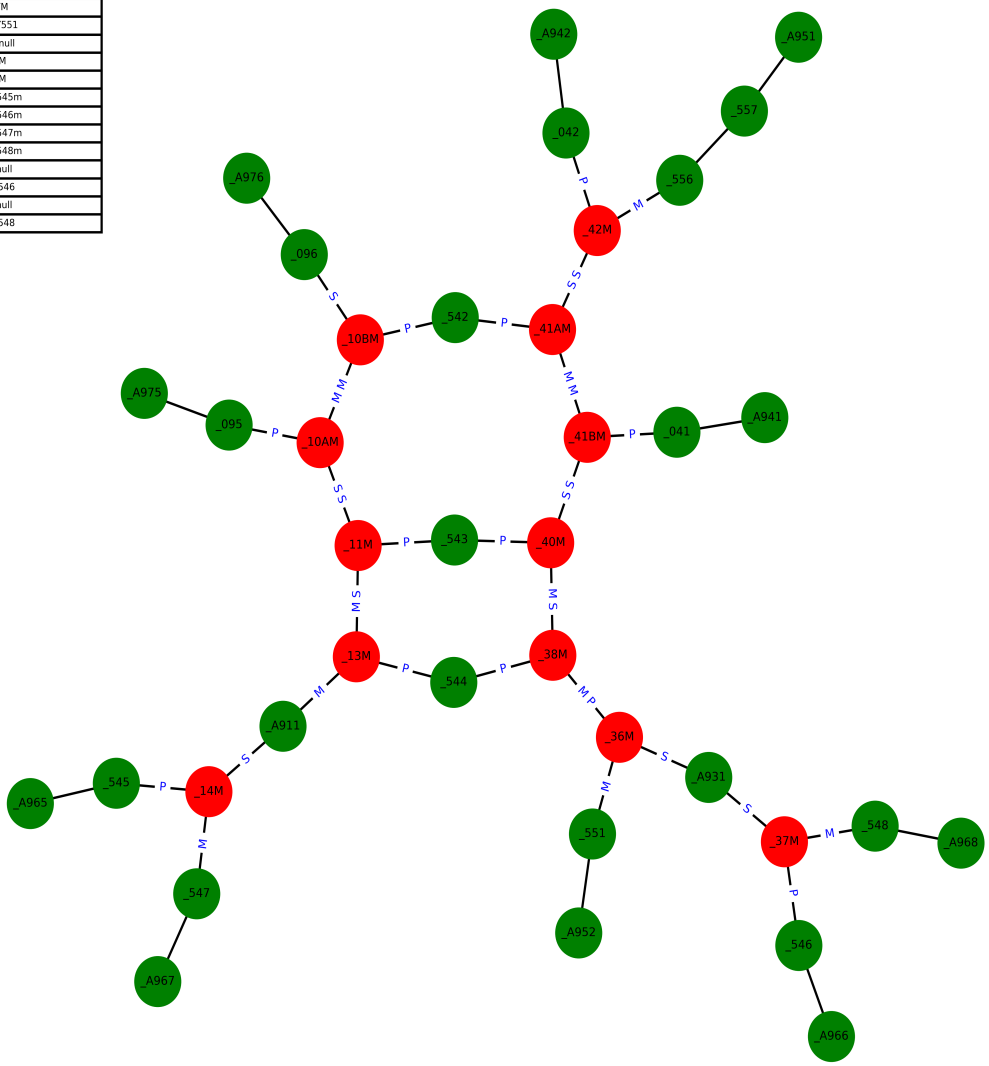

Figure 4.10: Approximate representation.

Chapter 5

Results

In this chapter we will show some results regarding the parsing and the formal verification of the proposed model, done through UMC. The approximate representation generated will be compared with a real configuration of a railway interlocking.

Firstly, we will show how the parsing is done between a route represented in XML and the objects allocation in UMC models. Then, we verify the abstraction generated on each route granting all the properties previously listen in 3.4.5,

5.1 Parsing

The left part of Fig. 5.1 represents a route described in XML and each condition gives an information about its configuration. The right part instead, is the allocation of the objects parsed from the XML route. Every object parsed on the right is referable to one of the conditions or the info provided from **source** and destination fields.
JYM is the signal object of the first linear object and *AO041* of the last

linear objects of the route.

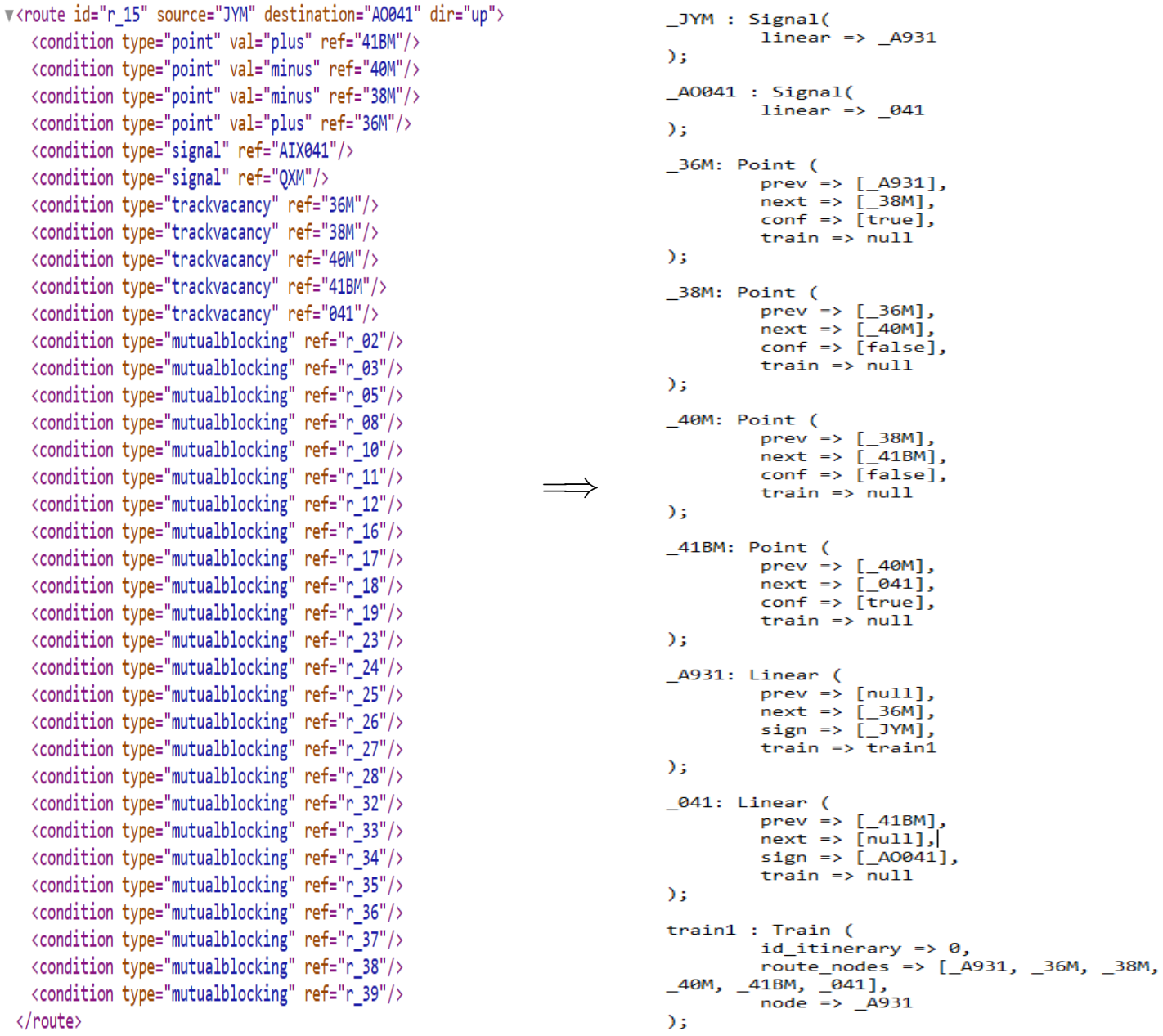

Figure 5.1: Parsing of a route into UMC code (i.e object allocation).

It is important to say that *JYM* is also used to retrieve the corresponding

linear object, *A931*. The four consecutive point objects, *36M*, *38M*, *40M* and *41BM* respectively, are referable to the first four conditions giving information about the configuration and its reference.

The last object on the right is the train object *train*1, which it is generated combining the linear object obtained from the **source** signal object and the references of the type *trackvacancy* conditions. In case of two routes, all the overlapping objects there will be allocated once and two train objects there will be instantiated. Notice that all the variables are filled by a mapping algorithm implemented into the *ParserXML* class. The UMC code obtained is completed with all the logic around the objects and the abstractions to verify the interlocking properties.

5.2 Formal verification

The abstractions mentioned in 3.4.5 are used to verify the expected behaviour of the model proposed.

Once we have the full UMC code, we can embed this code into the UMC framework and verify the correctness of the language.

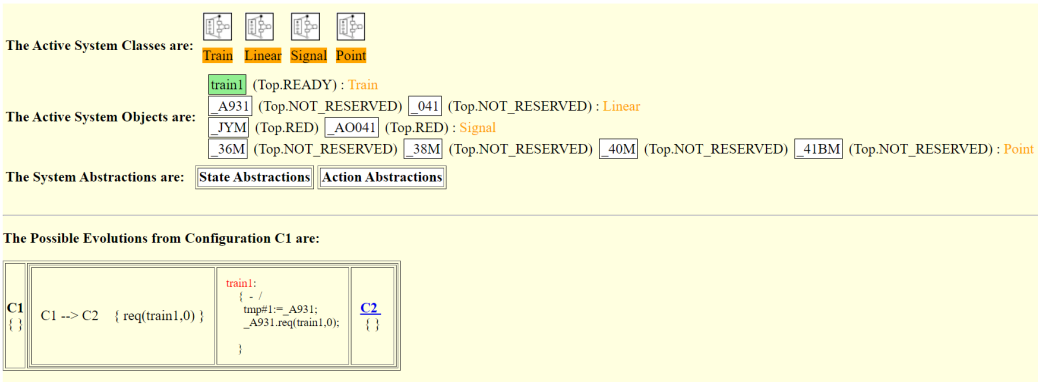

Figure 5.2: UMC Model view.

The Fig. 5.2 shows the interface of the framework UMC, after the model was loaded inside the system. The figure suggests the correctness of the model generated by the parser, since the framework well processed the model and implemented all the logic and the objects, which would not have happened in the case of the wrong model. The figure also shows the next transition that can be executed bringing the train to execute a request to the first element of the route. In case of two routes, there will two possible transition at the beginning.

The abstractions in Fig. 5.3 are checking if the situations of derails and faults signals can be generated in the model. We also generate a label when the train has arrived to the last element of the route, meaning the stabilization of the transition between objects. Two error labels can be raised when we have a discarded message or a design error.

```
Abstractions{
        Action $1($*) -> $1($*)
        Action: $obj:Runtime_Error -> Design_Error($obj)
        Action: lostevent($e,$*) -> discarded_message($e,$*)
        State : train1.node == _041 -> train1_arrived
        State : _36M.conf[0] == false and _36M.train == train1 and inState(train1.MOVE) -> DERAIL_train1
        State : _38M.conf[0] == true and _38M.train == train1 and inState(train1.MOVE) -> DERAIL_train1
        State : _40M.conf[0] == true and _40M.train == train1 and inState(train1.MOVE) -> DERAIL_train1
        State : _41BM.conf[0] == false and _41BM.train == train1 and inState(train1.MOVE) -> DERAIL_train1
        State : inState(train1.MOVE) and _JYM.red == true and _A931.train == train1 -> FAULT__JYM
        State : inState(train1.MOVE) and _AO041.red == true and _041.train == train1 -> FAULT__AO041

}
```

Figure 5.3: Abstractions of the route generated.

Every property has been verified in almost all the possible routes generated over six XML files. Almost all the routes have been verified of their properties of no derails, no signal faults, no accidents and stabilization. The approximates representations have been generated from all the tested XML file and in all the cases the approximation is referable to real one.

We will shows several situation and results.

5.2.1 Configuration I: one train and one route

This configuration represents the one described above. The properties that we verified are no derails, stabilization and no signal fault.

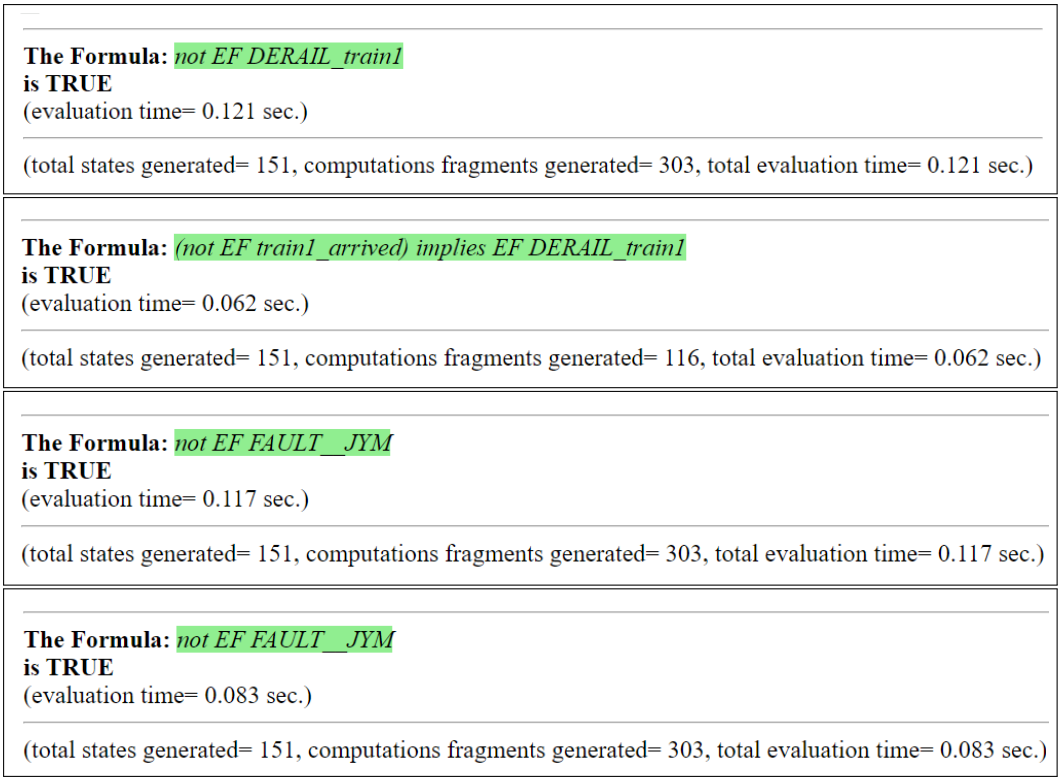

Figure 5.4: Properties verified.

The first box verifies that "it does not exist a future where *DERAIL_train1* can happen". The second one that "it does not exist a future where *train1_arrived* implies that it exists a future where *DERAIL_train1* can happen". The last two indicate that it does not exist a future where *FAULT_** can happen.

5.2.2 Configuration II: two trains and two routes with no crossing points

This configuration has two routes with no crossing points and each route is executed by one train. No derails, no accidents, no signal fault and stabilization properties are verified in Fig. 5.5. Notice that possible derail means that the corresponding point could be not used in this route, so in case the positioning is not correct for a possible following step of the train, there could be a derail.

The Formula: *not EF POSSIBLE_DERAIL*
is TRUE
(evaluation time= 0.158 sec.)

(total states generated= 784, computations fragments generated= 1569, total evaluation time= 0.158 sec.)

The Formula: *((not EF train1_arrived) or not EF train2_arrived) implies EF ACCIDENT*
is TRUE
(evaluation time= 0.060 sec.)

(total states generated= 784, computations fragments generated= 118, total evaluation time= 0.060 sec.)

The Formula: *not EF ACCIDENT*
is TRUE
(evaluation time= 0.260 sec.)

(total states generated= 784, computations fragments generated= 1569, total evaluation time= 0.260 sec.)

The Formula: *not EF FAULT__AOY557*
is TRUE
(evaluation time= 0.166 sec.)

(total states generated= 784, computations fragments generated= 1569, total evaluation time= 0.166 sec.)

The Formula: *((not EF train1_arrived) or not EF train2_arrived) implies ((EF ACCIDENT) or EF FAULT_AOY557)*
is TRUE
(evaluation time= 0.064 sec.)

(total states generated= 784, computations fragments generated= 118, total evaluation time= 0.064 sec.)

Figure 5.5: Properties verified.

Linears	Signals
556	D: NYM U:LYM
557	D: LZM U:AOY557
A951	D: AIY557 U:null
042	D: QM U:AOX042
041	D: QXM U:AO041
A941	D: AIX041 U:null
A942	D: AI042 U:null
542	D: FM U:PXM
543	D: FXM U:PM
AR542	D: null U:AIX542
AR543	D: null U:AI543
544	D: GM U:GYM
AR544	D: null U:AIY544
551	D: KYM U:AOY551
A952	D: AIY551 U:null
A931	D: HM U:JYM
546	D: j546m U:jy546m
548	D: j548m U:jy548m
A966	D: null U:AIY546
A968	D: null U:AIY548

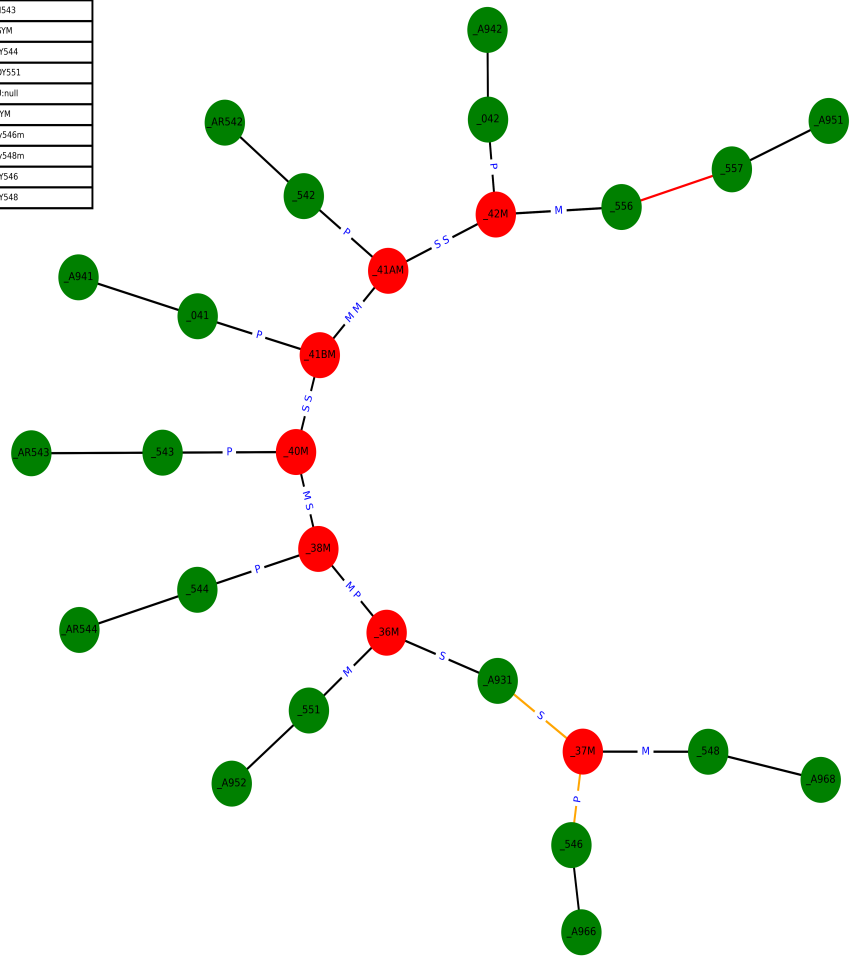

Figure 5.6: Approximate representation of selected routes.

Fig. 5.6 shows the approximate representation with highlighted both routes selected.

5.2.3 Configuration III: two trains and two routes with crossing points

This configuration presents two routes with crossing points and one train each route. The communication continues until one of the two trains takes the lead and starts its path. Finally, the other train can finish its path. In

The Formula: *not EF (DERAIL_train1 or DERAIL_train2)*
is TRUE
(evaluation time= 3.868 sec.)

(total states generated= 27170, computations fragments generated= 108681, total evaluation time= 3.868 sec.)

The Formula: *not EF ACCIDENT*
is TRUE
(evaluation time= 3.013 sec.)

(total states generated= 27170, computations fragments generated= 54341, total evaluation time= 3.013 sec.)

The Formula: *not EF (FAULT__AO041 or (FAULT__JYM or (FAULT__AOX042 or FAULT__PM)))*
is TRUE
(evaluation time= 5.509 sec.)

(total states generated= 27170, computations fragments generated= 217361, total evaluation time= 5.509 sec.)

Figure 5.7: Properties verified.

Fig. 5.7 we can see the space of states generated is increased compared to the other configurations, since we allocated sixteen objects to recreate the two routes.

5.2.4 Configuration IV: Subway network representation

As part of the verification of our approximate representation, we try to parser a subway interlocking since it is more complex under the architectural

aspect. Comparing the results shown in Fig. 5.8 and Fig. 5.9, we agreed that as first representation it is a good achievement (keeping into account the missing geographical coordinates).

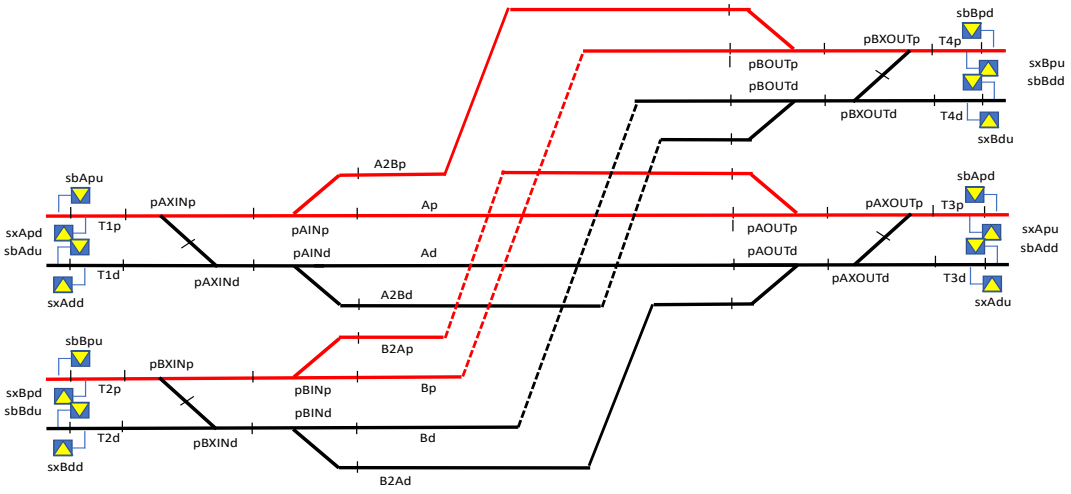

Figure 5.8: Schematic representation the physical layout of the subway network.

Linears	Signals
T1p	D: sxApd U:null
T2p	D: sxBpd U:null
T3p	D: null U:sxApu
T4p	D: null U:sxBpu
bT1p	D: null U:sbApu
bT2p	D: null U:sbBpu
bT3p	D: sbApd U:null
bT4p	D: sbBpd U:null
Ap	D: null U:null
Bp	D: null U:null
A2Bp	D: null U:null
B2Ap	D: null U:null
T1d	D: sxAdd U:null
T2d	D: sxBdd U:null
T3d	D: null U:sxAdu
T4d	D: null U:sxBdu
bT1d	D: null U:sbAdu
bT2d	D: null U:sbBdu
bT3d	D: sbAdd U:null
bT4d	D: sbBdd U:null
Ad	D: null U:null
Bd	D: null U:null
A2Bd	D: null U:null
B2Ad	D: null U:null

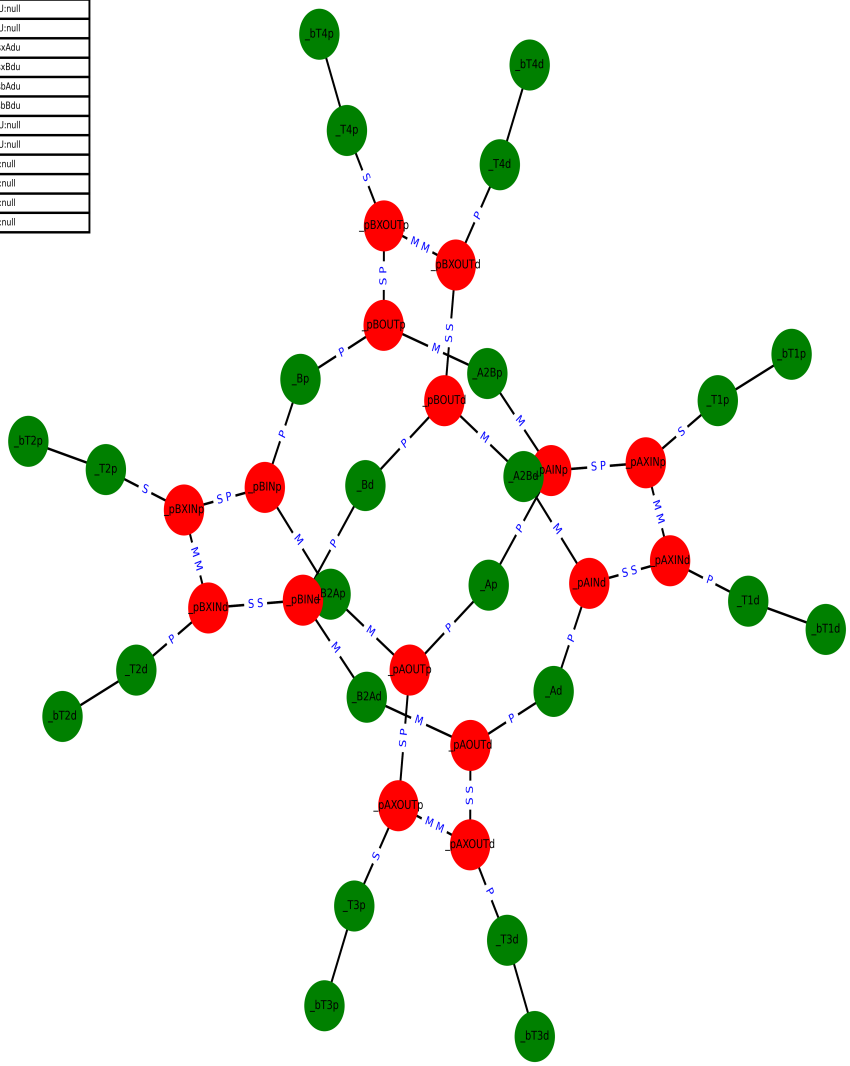

Figure 5.9: Approximate representation of subway network.

Chapter 6

Conclusions

In this work, we firstly introduced the main concepts of Software dependability, discovering that since the systems are becoming increasingly complex, it is important ensure robust checks. We presented the architecture and the logic behind an interlocking and why it is important to verify its reliability through model checking. We explained the UMC framework and its utility in model checking and construction, bringing a fast way to verify that systems work as designed while minimizing the cost, the complexity and the potential of mistakes or failure. We implemented and developed a tool to parser XML files related to railway interlocking into UMC models, giving the possibility to have a full overview of the network layout and route table while also providing an approximate representation using graphs to give a first impression of the railway network. A user interface has been developed to ensure ease of use of the tool, providing the possibility to set up different routes and overviews configurations.

We also learned that the main drawbacks are connected to the model checking, which today still represents a blocking point in this field. In further studies would be interesting to try new framework to verify systems that can

give you more accessibility and completeness. It would be interesting either to add new features to this tool, implementing other kind of parser (i.e json to RailML, XML to RailML) or, to improve the ability of the UMC code generated increasing its complexity and getting closer to the most modern systems. In addition, the approximate representation could be improved by providing more details about each element, integrating a framework to give to possibility to interact with nodes and implementing train objects that can be moved over the paths.

References

[1] Railway applications – Communication, Signalling and Processing systems – Software for railway control and protection systems, 2011.

[2] wxWidgets: Cross-Platform Graphical User Interface Library. `https://github.com/wxWidgets/wxWidgets`, 2021.

[3] Yousef Al-houmaily. Atomic commit protocols, their integration, and their optimisations in distributed database systems. *IJIIDS*, 4:373–412, 09 2010.

[4] C. DELVOSALLE. Domino Effects Phenomena : Definition, Overview and Classification. *European Seminar on Domino Effects, Leuven, Belgium, 1996*, 1996.

[5] Alessandro Fantechi, Anne E. Haxthausen, and Michel Bøje Randahl Nielsen. Model Checking Geographically Distributed Interlocking Systems using UMC. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 278–286, 2017.

[6] Samuele Foni and Lorenzo Pardini. Elementi di Software Dependability, 2016. Unpublished.

[7] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Networkx: A Python Package for the Study of Complex Networks. `https://github.com/networkx/networkx`, 2021.

[8] Daniel Krajzewicz. Rapidxml: An attempt to create the fastest XML parser possible. `https://github.com/danielkrajzewicz/RapidXML`, 2021.

[9] Cecilie Lindberg and Mikkel Blach Hansen. A parser for Linh's XML Format. Technical University of Denmark - DTU, 2020.

[10] Niels Nlohmann. Json for Modern C++. `https://github.com/nlohmann/json`, 2021.

[11] Klaus Schneider. *Verification of Reactive Systems: Formal Methods and Algorithms*. Springer, 2004.

[12] D. Skeen and M. Stonebraker. A Formal Model of Crash Recovery in a Distributed System. *IEEE Transactions on Software Engineering*, SE-9(3):219–228, 1983.

[13] Maurice H. Ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. From EU Projects to a Family of Model Checkers. In *Lecture Notes in Computer Science*, volume 8950, pages 312–328. Springer, 2015.