Masterarbeit

The Simplex Architecture in Practice – Runtime Assurance for Safety-Critical Railway Systems

Die Simplex-Architektur in der Praxis – Konsistenzprüfung zur Laufzeit für sicherheitskritische Eisenbahnsysteme

Clemens Tiedt

Hasso Plattner Institute at University of Potsdam

March 6, 2024

Masterarbeit

The Simplex Architecture in Practice – Runtime Assurance for Safety-Critical Railway Systems

Die Simplex-Architektur in der Praxis – Konsistenzprüfung zur Laufzeit für sicherheitskritische Eisenbahnsysteme

by
Clemens Tiedt

Supervisors
Prof. Dr. Andreas Polze, Robert Schmid, Katja Assaf
Professur für Betriebssysteme und Middleware

Hasso Plattner Institute at University of Potsdam

March 6, 2024

Abstract

The railway sector is experiencing a shift towards standardized interfaces between command, control and signalling components. The primary European initiative to develop such interfaces is EULYNX whose specifications are under active development. While these interfaces increase interoperability and decouple component lifeycles, updates to interfaces require updates to the components implementing them. Since safety is the primary concern in railway operations, components have to go through a thorough certification process before they can be used in the field. Updates to safety-critical components require a recertification of the component. However, the fundamental tasks of railway components rarely change with interface updates. Most interface updates only add information (such as additional diagnostics) to messages without changing their existing contents.

This thesis introduces a concept based on the Simplex architecture to develop a component that implements an updated interface whose safety properties are consistent with an existing, trusted component. We postulate that it is possible to inherit the safety properties of an existing component at the cost of availability.

We demonstrate the viability of this concept with a prototype for an axle counting object controller. The prototype uses an existing, certified object controller implementing the older NeuPro interface to increase the safety of an object controller which implements the newer EULYNX interface. Using three test cases based on the EULYNX standard, we show that the prototype handles both correct and incorrect behavior from the untrusted controller, remaining safe at the cost of reduced availability. An analysis of the prototype's source code with the *cyclomatic complexity* and *cognitive complexity* metrics shows that its core safety component, the decision module, requires a low amount of complexity. Comparing our Simplex-based approach to a EULYNX-NeuPro translation module, our approach does not need to change with interface updates and likely requires a less complex implementation.

To make the prototype certifiable, two major changes are necessary. Firstly, the prototype runs on a standard Linux computer. In order to be certifiable, it must be adapted for a safety platform such as ARINC 653. Secondly, the prototype uses an event loop architecture for time constraints which is not realtime-capable and would have to be replaced with a periodic architecture and deterministic execution model.

Contents

1 Introduction **3**
1.1 CCS Overview 3
1.2 Digitalization in the Railway Sector 4
1.3 Introduction to Train Detection Systems 5
1.4 Further areas of application 7
1.5 Contribution 7

2 Background and Related Work **9**
2.1 The Simplex Architecture 9
2.1.1 The Basic Simplex Architecture 9
2.1.2 Decision Module 10
2.1.3 Architecture Variants 12
2.2 Other Dependability Patterns 13
2.2.1 Triple Modular Redundancy 13
2.2.2 N-Version Programming 14
2.2.3 Encoded Execution 15
2.3 Related Work 16
2.3.1 Railway Safety 16
2.3.2 Other Simplex Applications 18
2.3.3 Other Non-Simplex Applications 19

3 Concept **21**
3.1 Use Case 21
3.2 Requirements 22
3.3 System Design 22
3.3.1 Fault Model 22
3.3.2 Comparison of Safety Patterns 24
3.3.3 Error Handling Strategies 26
3.3.4 Comparison of NeuPro and EULYNX 27
3.4 Architecture 29
3.4.1 Communication 29
3.4.2 Unreliable Subsystem 31
3.4.3 Trustworthy Subsystem 31
3.4.4 Decision Module 31

4 Implementation **33**
4.1 Tooling 33
4.1.1 RaSTA Implementation and gRPC-RaSTA Bridge 33
4.1.2 SCI Implementation 33

4.1.3 Rust . . . 34
4.2 Hardware . . . 34
4.3 Software Architecture . . . 35
4.3.1 Axle Counter and Configuration . . . 36
4.3.2 Subsystems . . . 37
4.3.3 Decision Module . . . 37
4.4 Error Handling Strategies . . . 39
4.5 Timing Requirements . . . 43

5 Evaluation **45**
5.1 Experimental Evaluation . . . 45
5.1.1 Scenario . . . 45
5.1.2 Test Cases . . . 46
5.1.3 Experimental Setup . . . 47
5.1.4 Results . . . 47
5.2 Complexity of Decision Module . . . 48
5.3 Comparison with NeuPro-EULYNX translator . . . 50

6 Conclusion and Future Work **53**
6.1 Conclusion . . . 53
6.2 Future Work . . . 54

References **55**

A Appendix **61**

Disclaimer

Throughout this thesis, I use the pronoun "we" in order to be consistent with the writing style in the field. This applies to subsection 2.1.2 in particular which is quoted verbatim from [1]. Despite the use of "we", the work presented in this thesis was done by me individually.

1 Introduction

New standards and architectures in the railway sector, such as EULYNX, enable a level of interoperability between infrastructure components that has been impossible in the existing command, control and signalling (CCS) architecture. This interoperability allows railway infrastructure managers to combine components from different manufacturers in one installation and makes it easier for new manufacturers to enter the market by allowing them to create individual components that are compatible with other manufacturers' products.

At the same time, safety is the main concern for railway infrastructure. New products have to go through extensive testing to acquire the certification required to use them in actual installations. This thesis explores how trust in railway infrastructure can be inherited across interface updates.

1.1 CCS Overview

The primary task of CCS infrastructure is to ensure the safe operation of the railway system. At the center of a CCS installation stands the interlocking. From an interlocking, a *signaller* manages railway traffic. While the signaller controls the routing of trains, the interlocking is responsible for the safety of these operations. Depending on the interlocking type, it monitors one or multiple stations.

Field elements are used to implement routing decisions and report the trackside status back to the interlocking. Examples of field elements include:

Signals are used to communicate routing information to train drivers. Most modern CCS installations use light signals which can show different *signal aspects* that indicate not only whether a train is allowed to continue onward, but can also contain information such as the allowed speed and information on the next signal.

Points are used to physically route trains. Usually, points have a straight and a branching track, although points where both tracks are branching can also exist.

Train Detection Systems are used by the interlocking to check if tracks are vacant or occupied. Simpler track circuits can only detect if a track is occupied, whereas more sophisticated axle counters can report exactly how many axles of a train are in the monitored track section.

In a conventional CCS architecture, the track network is divided into blocks which form the basis of the safety logic. Blocks are separated by signals and their occupancy is monitored using train detection systems. By ensuring that at most one train is in a block at a time, the interlocking guarantees safety from collisions.

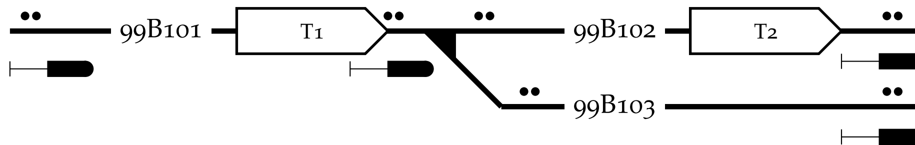


Figure 1.1: An example track network. The tracks are divided into three blocks, 99B101 - 99B103 that are each delimited by axle counting heads.

An example of a track network and field elements is shown in Figure 1.1. The network contains one point (◥) and four signals (⊢▬). The network is divided into three blocks which are delimited by axle counting heads (••) at the start and end. The sections 99B101 and 99B102 are occupied by T1 and T2 respectively. Until T2 moves out of section 99B102, T1 cannot safely enter it.

Overall, the safety responsibilities of the interlocking can be summarized as guaranteeing these properties:

1. Safe Route
2. Safe Speed
3. Safe Distance

1.2 Digitalization in the Railway Sector

Technology has been used to increase the safety of railway operations for a long time. The first mechanical interlockings were developed in the 19th century. These interlockings implemented the safety logic of their routes mechanically. Signals and points are controlled via levers. The lever controlling a route's signal is physically blocked until all points on that route are set to the correct orientation. Towards the end of the 19th century, electro-mechanical interlockings were developed [34]. These still use a mechanical safety logic, but can control field elements electrically.

The next step after electro-mechanical interlockings came in the form of relay interlockings in the 1930s. While a mechanical interlocking ensures the safety of a route, the signaller still needs to manually set all field elements to the correct position. Relay-based interlockings display a schematic overview of the controlled area with push buttons for field elements. Depending on the type of relay interlocking, the signaller has to use these buttons to set required field elements for a route to the correct position or just press the buttons for the start and end signal with the interlocking automatically controlling field elements.

In the 1980s, the first electronic interlockings were developed [23]. Previous types of interlocking had to be built specifically for one area of supervision. For example, adding a new point to a relay interlocking requires the addition of new relay groups as well as rewiring to connect these groups to the existing interlocking. Since the safety logic of an electronic interlocking is implemented in software, it is much easier to change. However, the lower dependability of computers compared to purpose-built hardware means that more effort is necessary to ensure the safety of an electronic interlocking.

In a conventional CCS installation, all components are provided by the same manufacturer. This means that no standardized communication interfaces between components are necessary. However, if changes or repairs to the installation become necessary, they can only be made by the original manufacturer.

Even existing electronic interlockings follow this principle. In contrast to previous signalling technology, digital systems need updates or replacements much more often. Additionally, they depend on components (such as processors) that are built by external manufacturers. If components are purpose-built such as relay groups, replacements can be manufactured as long as the interlocking remains in use. However, if complex digital components are not manufactured anymore, no replacement parts for existing interlockings can be built anymore.

The next generation of interlockings, referred to as *digital interlockings* addresses these issues. Like electronic interlockings, digital interlockings are computer-based. However, they use a component-based architecture with standardized interfaces. EULYNX is an EU-wide project to define these interfaces. The project was started in 2014 by a consortium of European railway companies. It expands on previous national efforts such as the German "Neuausrichtung in der Produktionssteuerung" (new orientation of production control, abbreviated as NeuPro). EULYNX defines a set of interfaces between the interlocking and field elements. These standardized interfaces make it possible to use field elements and interlocking systems from different manufacturers together. Figure 1.2a shows these interfaces and which components they connect to.

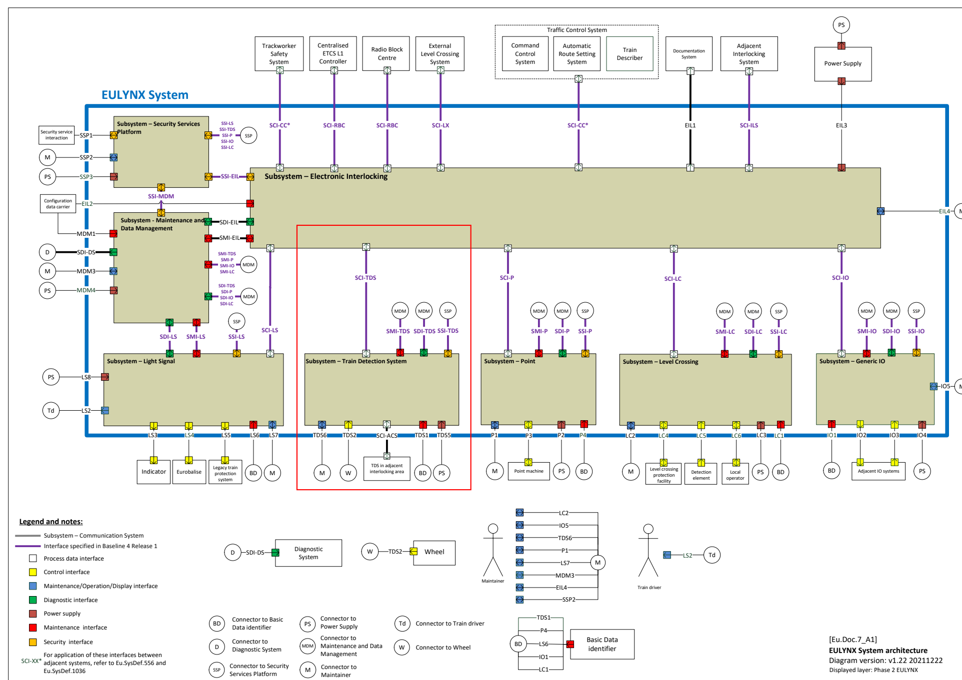
Within the EULYNX specification, there exist subsystems for different field elements. Each subsystem is associated with a *Standard Communication Interface* (SCI) which is a network protocol on the application layer. The SCI protocols operate on top of the *Rail Safe Transport Application* (RaSTA) protocol which uses redundant communication channels and implements retransmission. EULYNX also specifies management and debugging interfaces. However, we only focus on SCI communication in this thesis.

1.3 Introduction to Train Detection Systems

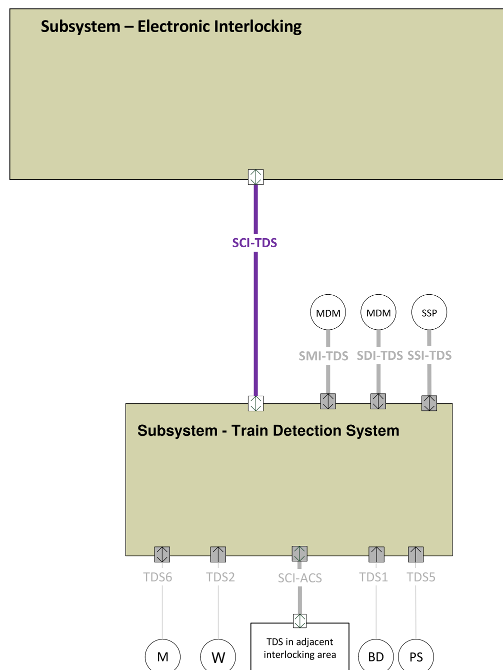
In order to guarantee the safety of a route, it is important to know the locations of all trains in the area supervised by a signaller. For example, if a train received movement permission for a route still occupied by another train, they could collide. Originally, it was the signaller's responsibility to ensure that a track is empty by sight. However, this approach is susceptible to human error and does not scale well for bigger interlockings.

These disadvantages led to the development of train detection systems (TDS). When using a TDS, the tracks are divided into sections which can be monitored individually. The track circuit, one form of TDS, was developed as early as 1864 [25]. Track circuits use an electrical circuit in the train tracks of a section that is shorted when a train occupies that section. While track circuits address the disadvantages of train detection by sight, they come with disadvantages of their own. They can only reliably prove if a section is vacant, since an occupied and disturbed section both register as a short of the track circuit.

Axle counters are another type of TDS. Axle counters use train detection points (TDP) attached to the tracks to monitor the passing of trains. A TDP is placed at the beginning and end of each section. When an axle passes the entry TDP, the section is reported



(a) Overview of the EULYNX System Architecture, taken from [16]. The highlighted section shows the interfaces of the subsystem Train Detection System.



(b) The excerpt of the EULYNX architecture relevant to this thesis. Interfaces not considered in this thesis are grayed out.

Figure 1.2: The EULYNX System Architecture and the relevant interfaces for this thesis.

as occupied. After all axles counted in by the entry TDP have passed the exit TDP, the section is reported as vacant again. Modern axle counters can monitor multiple sections by using the exit TDP of one section as the entry TDP of the next one. Compared to track circuits, axle counting requires more sophisticated equipment, but provides more fine-grained information about the occupancy status of a section. However, if an axle counter experiences a failure, the occupancy status of its monitored sections is unknown since the status is only stored in the axle counter's state. This does not happen with a track circuit, since it will be shorted or clear again as soon as power is restored.

Both NeuPro and EULYNX specify protocols for communicating with TDS. These include management and debugging interfaces, but for this thesis we focus on SCI-TDS (as shown in Figure 1.2b). In the terminology of EULYNX, one TDS monitors one or more *Track Vacancy Proving Sections* (TVPS). A TVPS has an occupancy status which can be *vacant*, *occupied* or *disturbed*.

1.4 Further areas of application

In the railway sector, high dependability requirements are a challenge to the development of digital systems. Similar trends appear in other industries as well. For example, electrical infrastructure must be highly available and reliable. As Petersen et al. outline in [35], digitalization of the energy infrastructure often takes the form of automated monitoring and control. For example, a smart energy grid can scale its energy output depending on current demand. However, increased digitalization opens up new attack surfaces, such as critical systems connected to the internet.

1.5 Contribution

The railway industry is becoming increasingly digitalized. Part of this digitalization is the development of standardized interfaces, such as NeuPro and EULYNX. It is common for these interfaces to be updated regularly to support new capabilities. Updates to an interface also necessitate updates to the components implementing the interface. While regular updates are common practice in the software industry, they are unusual for conventional railway technology since updates require recertification.

However, the safety-critical tasks performed by many railway components do not change substantially or at all through updated interfaces. Proving the safety of an entirely new component is difficult and costly. This difficulty could be reduced if it were possible to use an existing, trusted component to demonstrate the safety of a new component. This thesis explores whether this inheritance of trust is possible for an axle counting object controller.

As the proof of concept in Chapter 3 - Chapter 5 is based upon the Simplex architecture, the necessary background information is provided in Chapter 2. Further similar architecture patterns used for safety-critical applications are also discussed in this chapter. In Chapter 3, we develop an architecture for an axle counter with inherited trust. The architecture is based on requirements derived from the axle counting use case. We present a prototype implementation of this architecture in Chapter 4. While the prototype in its

current state is not certifiable, we discuss the changes necessary for deployment on a safe platform, such as ARINC 653. Chapter 5 provides a qualitative, experimental evaluation of the prototype. Using a scenario derived from the EULYNX specification, we show that the prototype correctly implements the fault model introduced as part of the architecture. Apart from the experimental evaluation, we also consider the complexity of the prototype's source code. We show that the component responsible for making safety-critical decisions in the prototype is of low complexity. In Chapter 6, we summarize the results of our work in this thesis to develop an object controller with inherited trust. We also discuss the further applicability of our approach beyond the axle counting use case.

2 Background and Related Work

Designing dependable systems can be difficult. In order to reduce this difficulty, a number of dependability patterns have been developed over the years. In this chapter, we introduce a selection of these patterns. We focus on the Simplex architecture specifically, as it is the pattern implemented in our prototype. Section 2.1 provides and overview of the Simplex architecture in general, as well as its components and variants. In Section 2.2, we discuss a selection of other dependability patterns with a focus on the types of errors they can handle. Some of these patterns, such as Triple Modular Redundancy, are already successfully used in railway systems. In Section 2.3, we discuss how dependable systems are developed today. We start with an overview of techniques from the railway industry. We also consider other Simplex-based applications as well as similar applications outside the railway domain.

2.1 The Simplex Architecture

The Simplex Architecture is a system design pattern to ensure the safety of a system during its runtime. Since there are a number of variations on this pattern described in the literature, this section provides an overview of the terminology and common structures in different simplex implementations.

2.1.1 The Basic Simplex Architecture

The most common version of the Simplex Architecture consists of two controller modules and a decision logic module as shown in Figure 2.1.

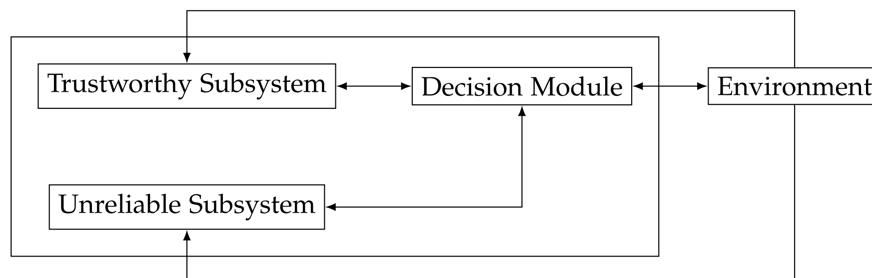


Figure 2.1: The Basic Simplex Architecture with two controllers.

Depending on the specific implementation, different names for these modules are common. Since the controllers in this thesis are fully functional individual units, we refer to them as *subsystems*. For their roles, this thesis adopts the terminology of a *trustworthy* and *unreliable* system as used by Crenshaw et.al. [11]. In other implementations, the

roles are described with the terms *high assurance* and *high performance* [38] or *baseline* and *experimental* [37].

The Simplex architecture is a means of increasing the dependability of a system by extending the dependability properties of the trustworthy subsystem to the unreliable subsystem. At runtime, both subsystem work in parallel. The decision module receives the outputs of both systems and compares them. If the outputs differ or the actions of the unreliable subsystem would put the system as a whole into an unsafe or unrecoverable state, the decision module gives control to the trustworthy subsystem.

This makes it possible to use a system whose dependability is uncertain in scenarios where a high dependability is necessary. For example, machine learning approaches offer performance improvements in many domains, but it is difficult to prove their dependability. By correctly recognizing when the actions of a machine learning implementation would violate the system's dependability constraints and switching to a conventional implementation, the dependability properties of the conventional implementation can be extended to the machine learning implementation. Similarly, the unreliable subsystem could run on commercial off the shelf (COTS) hardware for performance reasons. The decreased reliability of COTS hardware could be compensated by a more reliable fallback implementation.

2.1.2 Decision Module

The following subsection is quoted verbatim from [1] because it was written by the same author:

As the decision module evaluates the state of the system and controls the output of the overall system, it is a crucial component. It uses a *switching rule* to decide when to transfer control from the untrusted controller to the trusted controller so that the system never violates its *operation constraints*. These constraints refer to the physical or safety limitations of the system, e.g. a speed limit that may not be exceeded. However, the decision module has to switch to the trusted controller before the system leaves its *safe state*. For this purpose, a *recovery region* is defined.

Safety and Recovery Region Figure 2.2 shows the process of deriving a safety and recovery region. This process begins with a system description, which can take different forms. A system with linear dynamics can be described by a set of linear matrix inequalities. Systems with discrete states can be described by a state machine. However, many systems are described by a combination of discrete and continuous variables called hybrid systems. Based on the system description, the safety region is usually defined manually (e.g. based on domain expert knowledge). The safety region is then used to derive a recovery region using one of the following methods:

- Lyapunov Functions [38]
- Reachability Analysis [41]
- Predefined Values [36]
- Barrier Certificates [41] [13]

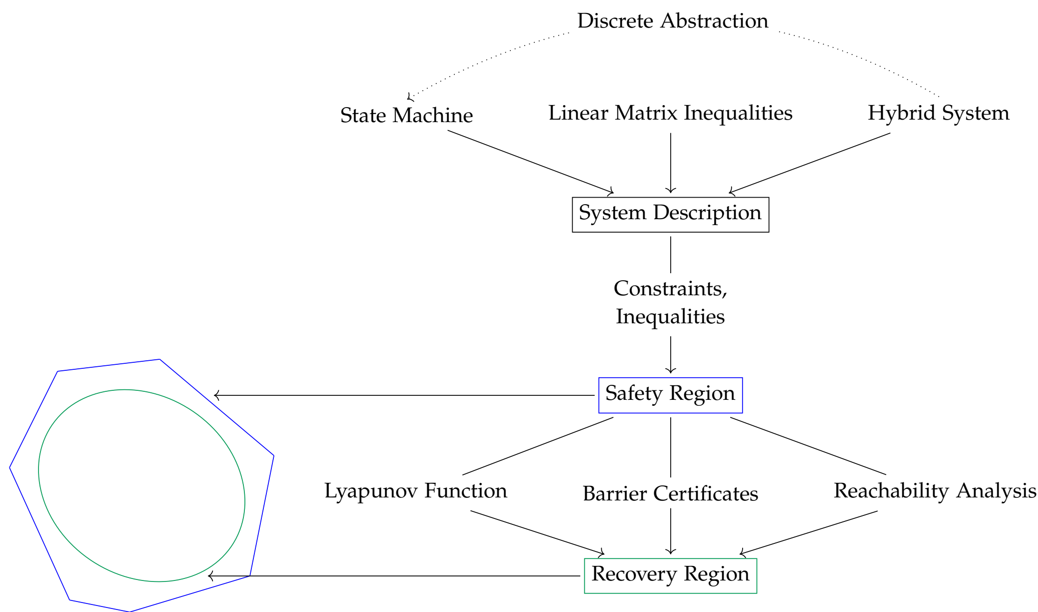


Figure 2.2: Different Approaches for the Decision Module

Lyapunov Functions The original simplex architecture used Lyapunov functions, a concept that originated in stability theory. They can be used to show the stability of dynamic systems. In the context of the simplex architecture, this relates to the recovery region for a system with linear dynamics. Using a Lyapunov function, the recovery region can be computed from a set of linear matrix inequalities, which is computationally inexpensive. However, the requirement for the system to have linear dynamics reduces the applicability of this approach. A special case of using Lyapunov functions is presented in the NetSimplex architecture (NetS) [42]. This Simplex variant includes network delays in the parameters used to determine the recovery region.

Reachability Analysis Reachability analysis provides a more generalized approach. Here, the set of possible states is enumerated, and the recovery region is computed by checking how quickly an unsafe state can be reached from each state. The decision module switches to the trusted controller if an unsafe state is reachable within a predefined number of state transitions. The reachability algorithm used determines the shape of the computed recovery region. Reachability analysis requires no constraints on the system's dynamics, making it broadly applicable. However, for a large state space, it can quickly become computationally expensive. The requirement to enumerate the system's possible states means that reachability analysis is only applicable to discrete systems. However, there are methods to create a discrete model of a continuous system. One such method is discrete abstraction as presented by [5], which discretizes the passing of time in a hybrid system.

Predefined Values The most common approach to defining a recovery region is using predefined values. Here, the constraints of the recovery region are manually defined, e.g.

based on previous experience. Predefined values are the least computationally expensive way to define a recovery region and easily incorporate human expertise. However, they also rely on this expertise and may define a more conservative recovery region than other approaches. An example of this approach can be found in the Component-based Simplex Architecture (CBS) [36]. In this variation, the switching conditions are defined in the form of contracts that the untrusted controller must adhere to. Further, all components of the system are Simplex-based and the active controller in one component may be part of another component's contract. This means that a switch in one component may cause dependent components to switch as well.

Barrier Certificates The more recent Barrier Certificate-based Simplex architecture uses barrier certificates to compute the recovery region. These can be used as an alternative to discrete abstraction when dealing with hybrid systems. A barrier certificate is a set of barrier functions, one of which is defined for each possible discrete state of the system (here called the system's *mode*). A barrier function takes a continuous state of the system as its input and maps it to a real number. The barrier functions are defined so that a state is mapped to a value less than or equal to zero if and only if no unsafe states are reachable. Thus, the zero-level set of a barrier function (i.e. all states whose value is less than or equal to zero) forms the recovery region in a mode.

Switching Rule The switching rule uses the safety and recovery regions to determine if the system should switch from the untrusted to the trusted controller. Depending on the type of system, the switching rule may be checked at every state transition (for discrete systems) or at a regular time interval (for continuous systems). When the untrusted controller proposes an action, the switching rule checks if this action would put the system into an unsafe state. It may also use a limited reachability analysis to check if the system's trajectory (e.g. if the state depends on external influences not controlled by the system) may lead to an unsafe state.

Many Simplex implementations only support switching from the untrusted to the trusted controller, but there are some implementations of reverse switching where the decision module may give back control to the untrusted controller after the system has returned to a safe state.

2.1.3 Architecture Variants

Over the years, different variants of the Simplex architecture have been developed, often with specific kinds of applications in mind. Based on the classification of Simplex variants described in [1], this subsection discusses their applicability to the train detection system (TDS) use case. Only variants that are directly derived from the original Simplex Architecture are discussed here, since their derived variants exhibit the same basic characteristics.

Original Simplex The original Simplex architecture has already been described in subsection 2.1.1. It can handle different kinds of unreliable controllers and can be adapted to different use cases by modifying the decision logic.

Multiple Controller This variant is similar to the original Simplex Architecture, but employs multiple unreliable subsystems. Since the simplicity of the implementation is an important criterion in this thesis, Multiple Controller Simplex is not applicable here.

Nested Simplex This variant uses another Simplex system as its trustworthy subsystem. Since there exists no Simplex-based TDS, it is not applicable to this thesis.

System-level Simplex While other Simplex variants can only tolerate faults on the application level, the System-level Simplex Architecture can also tolerate hardware faults. Since the implementation work of this thesis includes no direct interaction with hardware, this additional fault tolerance is not necessary.

L1 Simplex This variant uses elements of control theory to detect and mitigate physical failures as well as software failures. It can not only switch to the trustworthy subsystem if the unreliable subsystem would bring the system into an unsafe state, but also if the uncertainty in the system exceeds a predetermined threshold.

Neural Simplex Since the unreliable subsystem in this thesis does not use machine learning, this architecture variant is not applicable.

Distributed Simplex Since the individual object controller implemented in this thesis is not a distributed system, this variant is not applicable.

As this comparison shows, most Simplex variants are designed for use cases with different characteristics than a TDS. This leaves the Original Simplex and L1 Simplex as candidates. While the L1 Simplex Architecture can provide additional guarantees against physical failures, it requires additional complexity compared to the Original Simplex Architecture. Since the strength of the Simplex architecture for the purpose of certification is its simplicity, we only consider the original Simplex architecture from here on.

2.2 Other Dependability Patterns

There exist a variety of patterns that increase the dependability of a system. As described by Avizienis et al. [3], there exist different dependability attributes which may be ensured or even traded off in different ways. Although all patterns described in this section are intended to raise the dependability of a system, they affect different dependability attributes. In our discussion of these patterns, we focus on the types of faults they can and cannot handle.

2.2.1 Triple Modular Redundancy

A common pattern to protect against random failures is Triple Modular Redundancy (TMR) [15].

As Figure 2.3 shows, it is superficially similar to the Simplex architecture. The complete system is composed of three replicas and a switching component. As the term *replica* implies, these are not different implementations, but multiple instances of the same system. The replicas operate in parallel and the switch compares their outputs. If two or more of the replicas send the same output, the switch forwards that output to the environment.

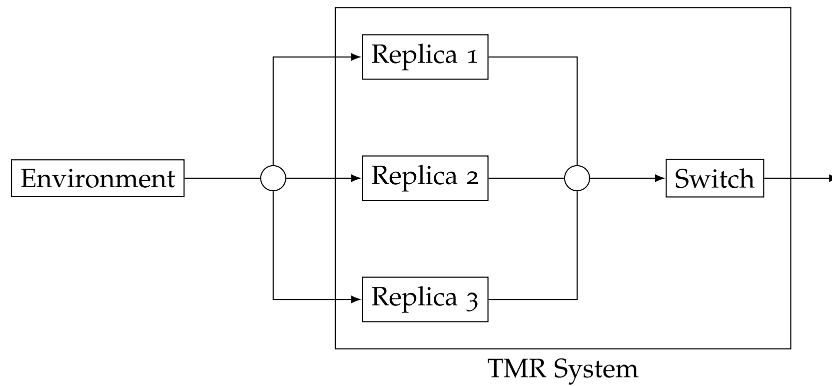


Figure 2.3: Triple Modular Redundancy architecture.

Since there are an uneven number of replicas, TMR can detect failures in up to two replicas and even correct failures in a single replica. It also requires only a small amount of additional implementation effort because all replicas reuse the same implementation.

However, TMR can only find and correct random failures. If there is a logical fault in the replicas, it will affect all of them equally and therefore be undetectable to the switch. Also, all replicas are treated as equally authoritative. For example, if all three gave different outputs, the switch would have no way to determine which output is correct. Even more severe errors could happen if two replicas experience the same type of fault, leading to their outputs overriding the correct one.

2.2.2 N-Version Programming

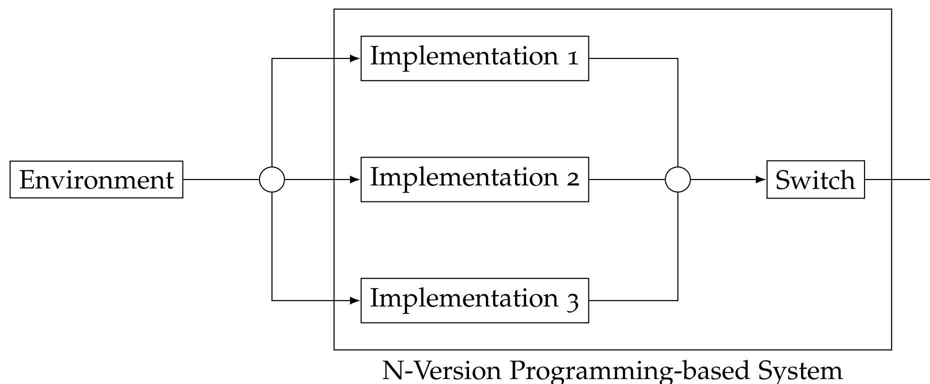


Figure 2.4: N-version programming architecture.

This pattern was introduced by Chen and Avizienis [10] and is similar to both TMR and the Simplex architecture. As shown in Figure 2.4, there are multiple subsystems that are each implemented independently, similarly to the Simplex architecture. However, none of these is treated as authoritative. Instead, similarly to TMR, a voting mechanism is used

to handle disagreements between subsystems. Depending on the number of subsystems, n-version programming can detect or tolerate subsystem failures.

A system implemented using n-version programming can tolerate the same faults as a system implemented using TMR. Further, it can tolerate systemic faults in subsystems, so long as they are not common between the subsystems. Beyond the additional computational also cost incurred by TMR, n-version programming requires additional implementation effort. In order to avoid common mode failures, the subsystems should be implemented in different ways from each other. For example, this could mean implementing them in different programming languages or using different algorithms for the same goal.

2.2.3 Encoded Execution

Another pattern to detect runtime errors in a system is encoded execution, first presented in [20]. This approach uses error correction codes to determine the correctness of the system at critical points during runtime. These codes can be separate or nonseparate from the value. [2]. If a value's error code is invalid, an error (e.g. a bit flip in main memory) must have occurred and the system has to handle it.

```rust
const A: i64 = 5;

fn mul(lhs: i64, rhs: i64) -> i64 {
    (lhs * rhs) / A
}

fn encode(value: i32) -> i64 {
    value as i64 * A
}

fn decode(value: i64) -> Result<i32, String> {
    if value % A == 0 {
        Ok((value / A) as i32)
    } else {
        Err(format!("{value} is not divisible by {A}, remainder: ", value % A))
    }
}

fn main() {
    let x: i32 = 2;
    let y: i32 = 7;
    let z = mul(encode(x), encode(y));
    println!("{:?}", decode(z));
}
```

Listing 1: Example implementation of AN encoding for multiplication in Rust

An example of a nonseparate error correction code is AN encoding. [19] This encoding only works on integer values, but can be applied to aggregate values by e.g. encoding each member of a structure individually. Here, values are multiplied with an integer constant A. Ideally, A should be a large prime. An encoded value x_c is valid if it is divisible by A. Instructions operating on these values must be aware of the encoding.

Listing 1 shows an example implementation of multiplication using AN encoding. For two encoded values x_c and y_c, naive multiplication would result in

$$x_c \cdot y_c = A \cdot x \cdot A \cdot y = A^2 \cdot x \cdot y$$

In order to correct this value, the result has to be divided by A. Other arithmetic operations have to be modified similarly. Note also that the encoded values use a larger datatype to ensure the encoding does not cause an overflow.

Encoded execution can be implemented with different error correction codes. More complex codes can make it more likely to detect faults, but add more runtime overhead as well. Encoded execution can also be combined with other safety patterns. For example, multiple AN-encoded instances with different values for A can be used as a form of n-version programming. [39].

2.3 Related Work

Various approaches for the development of safety-critical systems have been developed over the years. In relation to this thesis, there are three categories of related work:

1. Safety-critical systems in the railway domain
2. Applications of the Simplex architecture
3. Similar applications outside the railway domain and Simplex architecture

Since our use case concerns the railway domain, we first consider established methods for ensuring the safety of railway applications. While formal methods have commonly been used to prove the safety of railway applications, some approaches to reduce the burden of certification have been developed in recent years.

While we are not aware of any railway applications using the Simplex architecture, the Simplex architecture has been successfully demonstrated in other transportation-related use cases. While most of these use cases apply the Simplex architecture to the vehicle instead of the infrastructure, they still share the realtime constraints of our use case.

The final category of related work concerns approaches to redundant sensor or bus usage that are neither part of the railway domain nor a Simplex-based application. Since very few Simplex-based applications focus on the dependability of sensors, the works in this category fill a niche relevant to our use case.

2.3.1 Railway Safety

Most railway applications rely on formal methods to ensure safety. These methods work by specifying systems formally, instead of using natural language. The formal model can then be used to check that the system fulfills necessary constraints. A possible constraint

could be "A train may not exceed a speed of 300km/h". An example of the use of formal method is given by Shaoying et al. [27, 26] who demonstrate the use of the formal language SOFL to develop a railway crossing controller. The controller is specified in the form of a state machine whose safety properties can be statically checked. Iliasov et al. [24] use the SafeCap toolkit to show the safety of an entire interlocking. SafeCap is a modelling platform that includes automated theorem solvers which can be used to prove that a model fulfills safety constraints.

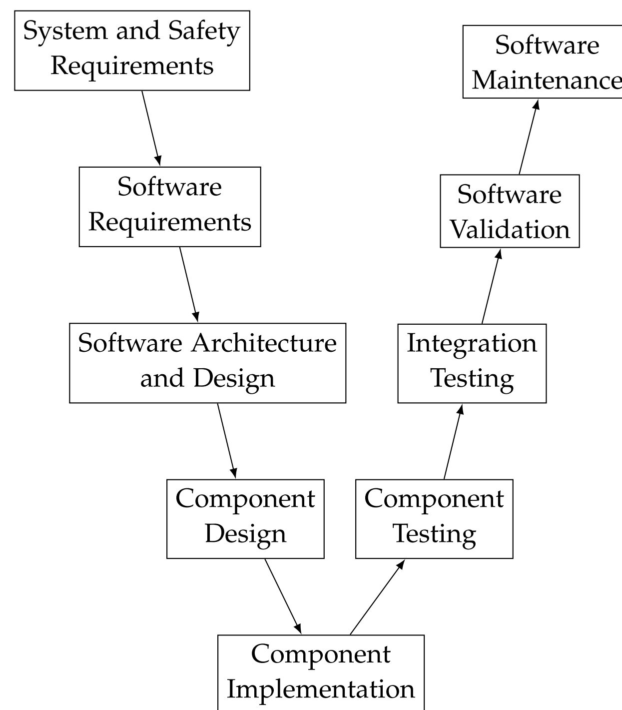


Figure 2.5: The V Model for software development as specified in DIN EN 50128 [14].

The safety characteristics of a system can be summarized using the safety integrity level (SIL). Non-safety components are referred to as SIL0 whereas systems with the highest safety requirements are SIL4. An established way of certifying a safety-critical railway application in Germany is to show that its development process is compliant with the guidelines in DIN EN 50128 [14]. These guidelines include the use of the V model. As shown in Figure 2.5, the V model describes a development process starting with a requirement analysis and system specification. The specification starts on the more abstract system level and then moves to the more granular component level. After the design is finished, the components are implemented. The system is then extensively tested based on the specifications developed earlier in the process. The V model also considers the ongoing maintenance part of the development process.

However, other approaches are being explored as well. For example, Bilbao et al. [6] describe a system that uses a multicore CPU for redundancy and falls back on degraded modes to keep operating in case of failure. While both are necessary to create a safe system,

this work focuses more strongly on the system architecture rather than the development process.

Other work focuses on the ability to update railway applications. Gala et al. [21] discuss the use of virtualization to make applications independent of the hardware they were originally deployed on. Since the certification process considers hardware and software of a system together, this approach has the potential to greatly reduce the effort required for certification. Another approach to updates is discussed by Moumouris and Zehnder [32]. Many systems combine safety-critical and non-critical components. If these components are isolated from each other, updates to non-critical components should not require a recertification of the system. The authors present a proof of concept using a segregating operating system to achieve this degree of isolation.

2.3.2 Other Simplex Applications

The Simplex Architecture is a well-established pattern for use in safety-critical applications in certain domains as illustrated by [1]. While there are no publications describing the use of the Simplex Architecture for railway applications, a variety of applications in other transportation domains exist.

One use case of the Simplex Architecture in the automotive domain is described by Bak et. al. in [5]. The paper presents the method of discrete abstraction which we elaborate on in subsection 2.1.2. It uses the prevention of offroad vehicle rollover as an example use case of this method. The hybrid model of the vehicle (which includes variables such as its speed, the ground angle and the steering angle) is discretized to find thresholds for when the safety controller (which reduces vehicle's speed and steering angle) should take control. This application uses the System-level Simplex Architecture [4] by implementing its decision logic in a Field Programmable Gate Array.

Another automotive use case is discussed by Feth et al. in [18]. The authors describe a platooning algorithm which uses the Simplex Architecture to ensure safety by avoiding collisions.

Apart from the automotive domain, the Simplex architecture has been widely applied in aviation. Nagarajan et al. [33] present an application that uses a Simplex-based approach as a geofence for autonomous aircraft. Similarly to our use case, an unsafe action from the untrusted controller, such as leaving the geofenced area, causes the system to gracefully cease operating by terminating the flight. The Simplex architecture has also been used in aviation to implement collision avoidance. Mehmood et al. describe one such use case in [31]. They use the black-box Simplex architecture, a variant of the Simplex architecture in which the trustworthy controller is unverified, to avoid collisions between F16 airplanes. Some of the same authors also used the distributed Simplex architecture for a collision avoidance use case [30]. In a distributed Simplex system, multiple actors each use the Simplex architecture and can take each other's states into account when deciding whether to switch.

2.3.3 Other Non-Simplex Applications

Axle counting object controllers have the characteristics of sensors. While this is unusual for Simplex applications, there exist methods that use redundancy to improve the dependability of sensors.

Granig et al. [22] developed a concept to account for dependability requirements in IoT devices. They propose a model for sensors where a second sensor is either used for calculating the mean, assuming small deviations in both, or for diagnostics. The latter approach is similar to a Simplex-based system. However, a significant difference to our use case is the continuous value space of their output, whereas ours is discrete.

Redundancy schemes are also used to increase the dependability of CAN bus communication. Xiang-Dong et al. [40] describe a method using redundant CAN bus connections with an FPGA-based controller to achieve better performance over software-based redundancy. Unlike a Simplex-based approach, the connections are treated as equally trustworthy.

3 Concept

Based on the research question and related work, this chapter describes the requirements and design of the Simplex-based object controller developed in this thesis. Since in practice the development of an object controller is safety critical and, thus, must follow the design process specified in EN 50128 [14], we base the development of our system on this standard while omitting all steps infeasible or unnecessary for a proof of concept. Consequently, we specify the applicable requirements in Section 3.2. The evaluation of possible implementations for each requirement and the design decision can be found in subsection 3.3.1 - subsection 3.3.4. Finally, based on these considerations and requirements, we conclude with the final architecture of our proof of concept (Section 3.4).

3.1 Use Case

As outlined in Chapter 1, the increasing digitization of the railway industry provides new chances and challenges. Unlike conventional railway infrastructure components, digital components require regular updates and have shorter individual lifecycles. All railway infrastructure components must be certified in order to use them in the field. The certification process is intended to ensure the safety and reliability of a component. While the approach of certifying a component once and using it as-is for up to 40 years works well for non-digital components, it is fundamentally incompatible with the requirement for updates and shorter lifetimes. As the certification process works now, an update requires a full recertification of the component.

Since the author of this thesis does not have extensive knowledge of the certification process, this thesis focuses on developing a system whose trust is not affected by updates, rather than one which would not require recertification after an update.

As a specific use case, this thesis examines the safety of an axle counting object controller. Within EULYNX, it fulfills the role of a Train Detection System (TDS). An axle counter monitors the occupancy status of one or more Track Vacancy Proving Sections (TVPS).

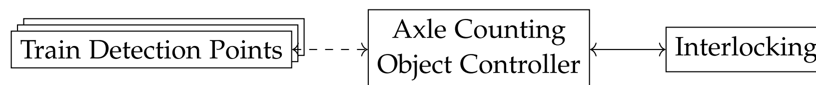


Figure 3.1: Role of the axle counter in the railway infrastructure.

The axle counter receives axle counting events from its connected train detection points (TDPs). This connection uses a proprietary, manufacturer-dependent protocol. When the occupancy status of a monitored TVPS changes, the axle counter sends a message to the interlocking. Since we consider a EULYNX system in this thesis, the axle counter and interlocking communicate using the *Standard Communication Interface - Train Detection System* (SCI-TDS). These connections are shown in Figure 3.1.

The interlocking is responsible for ensuring the safety of routes and communicating with trackside elements. As such, it communicates with the system developed in this thesis. The interlocking is provided by the EULYNX Live Lab.

3.2 Requirements

In order to implement the safe TDS outlined above, we introduce the following requirements for the system developed in this thesis:

R1 EULYNX Compatibility The system developed in this thesis shall communicate using EULYNX interfaces. It must be compatible with the interlocking used in the EULYNX Live Lab.

One of the effects of the increased digitization in the railway industry is the move towards standardization. Standardized interfaces as specified by EULYNX enable the combination of components from different manufacturers and decouple the lifecycles of components. In order to test the system developed in this thesis in a EULYNX environment, it must implement the relevant EULYNX interfaces. Since the system is a proof of concept, it only supports the SCI-TDS protocol and does not support maintenance interfaces which would be part of a full object controller.

R2 Trust in the System The system developed in this thesis shall use an existing, trusted system as the source of its trust.

The development of an object controller from the ground up is out of scope for this thesis. Instead, it explores methods to increase the dependability of an existing object controller.

R3 Error Handling The system shall tolerate faults that may cause safety violations.

Safety is the most important dependability goal in the railway context. Even if the system can detect errors by comparing outputs with a known safe implementation, it may only fail in well-defined ways that do not compromise its safety.

3.3 System Design

In this section, we discuss the design decisions made to ensure the implementation fulfills the requirements defined above.

3.3.1 Fault Model

To achieve dependability, we introduce a fault model that is later used to verify the dependability of the system. The fault model is based on the work of Avizienis et al. [3] who define five dependability attributes: availability, reliability, safety, integrity and maintainability as presented in Table 3.1.

Out of these attribute, safety is the most important since a safety violation may result in human or property damage. The TDS can only influence the state of the wider signalling

Attribute	Definition	Meaning in TDS use case
Availability	readiness for correct service	TVPS are correctly reported as vacant
Reliability	continuity of correct service	The TDS remains available and safe
Safety	absence of catastrophic consequences on the user(s) and the environment	TVPS are never incorrectly reported as vacant
Integrity	absence of improper system alterations	Telegrams are forwarded correctly
Maintainability	ability to undergo modifications, and repairs	Subsystems can be exchanged or upgraded without additional development effort

Table 3.1: Dependability attributes as defined in [3] (definitions quoted from p.5) and interpretation in the TDS context

system through the status reports of its monitored TVPS. From the perspective of the interlocking, a Track Vacancy Proving Section (TVPS) monitored by the Train Detection System (TDS) can have three states: Vacant, Occupied or Disturbed. If the reported and actual states are the same, no safety violation can occur. However, consider the consequences of a mismatch between reported and actual states. Each cell in Table 3.2 shows a possible combination of reported and actual states. The first line shows the effect on the availability of the TVPS, the second the effect on its integrity. A TVPS is considered available if it is reported as vacant. If a safety violation is possible, it is listed in a third line. The colors indicate how severely a mismatch impacts the TVPS' dependability.

Reported \ Actual	Vacant	Occupied	Disturbed
Vacant	Available Correct	Available Incorrect Safety Violation	Available Incorrect Safety Violation
Occupied	Unavailable Incorrect	Unavailable Correct	Unavailable Wrong Reason
Disturbed	Unavailable Incorrect	Unavailable Wrong Reason	Unavailable Correct

Table 3.2: Consequences of mismatch between reported and actual TVPS states (usability and correctness)

As the table shows, incorrectly reporting a TVPS as vacant carries the risk of a safety violation. If the TVPS is occupied, allowing another train to enter it may cause a collision. If it is disturbed, the TVPS may be unavailable and a train entering it might e.g. derail.

Reporting a TVPS as occupied should not lead to safety violations. If it is actually vacant, the report is incorrect, but an unavailable TVPS can cause no safety violations. The

same applies to a disturbed TVPS since in both states the TVPS may not be entered by a train.

Since a disturbed TVPS is also unavailable, the same conclusion applies to falsely reporting a TVPS as disturbed.

Therefore, the system should prioritise the detection of errors as follows:

Detect incorrectly available TVPS These cases are the only ones that can result in a safety violation. Therefore detecting them has the highest priority.

Detect incorrectly unavailable TVPS As discussed, an unavailable TVPS cannot cause safety violations. However, it unnecessarily reduces availability and should be avoided if possible.

Detect unavailable TVPS for wrong reasons While these cases are incorrect, they do not violate the safety or availability of the system. They may, however, indicate the presence of other faults in the TDS. Therefore, detecting them is helpful, but not necessary.

Since we do not assume that the encapsulated object controller operates correctly, the system must tolerate failures of the object controller that compromise its safety or availability. In [12], Flavin Cristian categorizes faults into crash, omission, timing, computation and byzantine faults. Detecting these different categories requires varying amounts of complexity and additional information.

Crash Fault If the object controller becomes unavailable, the system must be able to detect this and report the TVPS as unavailable.

Omission Fault, Timing Fault, Computation Fault, Byzantine Fault With only one TDS these faults are not detectable because the safety layer has no reference to detect missing or incorrect messages. However, these faults must be handled as they can cause the critical state mismatches discussed above.

3.3.2 Comparison of Safety Patterns

In order to fulfill requirement R2, the system makes use of a safety pattern as discussed in Chapter 2. This section focuses on detection of errors. For more information on error handling refer to subsection 3.3.3. The selected safety pattern must be able to to detect the following types of errors:

Crashes stop the object controller from operating at all. They may be caused by implementation faults (e.g. improper memory management resulting in a segmentation fault) or outside faults (such as a faulty power supply).

Correctness Errors These errors do not stop the object controller from operating, but cause it to give incorrect outputs as discussed in subsection 3.3.1.

These requirements already exclude encoded execution as a viable option. Assuming the object controller is provided as a specialized computer with the relevant software preinstalled and configured, or even just as a binary executable, encoded execution cannot be applied to it. Even if the object controller's source code is accessible and compiler

tooling to inject encoded execution is available, encoded execution can only detect a subset of relevant failures. Specifically, encoded execution is able to detect random hardware failures. Encoded Execution is unsuitable for detecting logical errors in the object controller implementation. However, it can be combined with other safety patterns to increase their robustness against random failures.

Another option discussed is triple modular redundancy (TMR), a pattern running three identical controllers in parallel to provide two-out-of-three redundancy. This pattern can detect both crashes and correctness errors if they occur randomly. If an error is random, it should only affect one replica. Since TMR compares the outputs of all replicas, it can detect random errors in one or two replicas. However, if an error is systematic, it becomes invisible to TMR. Systematic errors affect all replicas equally which means the voting component detects no discrepancy between outputs. Therefore, TMR can detect some correctness errors, but not all.

N-version programming follows a similar approach to TMR, but does not use identical replicas. This allows it to detect systematic errors in any of its versions.

The final option is the Simplex Architecture. As discussed in Chapter 2, it shares some characteristics with N-version programming. It makes use of a trustworthy system to fall back on in case of an error. When the goal is only to detect an error (i.e. a discrepancy between subsystems), this pattern is as powerful as N-version programming.

Pattern	Goal	DC	DCE	Initial Development Effort	Upgrade Effort
Encoded Execution	Detect Hardware Faults	●	○	Low	Medium
Triple Modular Redundancy	Protect against failures in one replica	●	◐	Low	Medium
N-Version Programming	Protect against logical software faults	●	●	High	High
Simplex Architecture	Reuse trust in existing system	●	●	High	Low

Table 3.3: Comparison of the different safety patterns. *DC* is short for "Detect Crashes", *DCE* is short for "Detect Correctness Errors". ● means the pattern can detect all errors of the category, ◐ means it can detect some and ○ means it can detect none.

Apart from their power to detect different types of errors, the different patterns differ in development and upgrade effort as shown in Table 3.3. Encoded Execution and TMR are the easiest to integrate into a system. Encoded Execution only requires the use of an encoded compiler or runtime environment. TMR requires the development of a voting component, but the replicas all reuse the same code. N-version programming and the Simplex Architecture both require a higher development effort since they require multiple full implementations of the same system.

Upgrade effort refers to the effort required to verify the safety of the system after an update to the encapsulated object controller. For Encoded Execution and TMR this

effort is not affected by the safety pattern. On the other hand, N-version programming requires high additional effort if multiple implementations have to be updated at the same time. If only the unreliable subsystem is updated, the Simplex Architecture requires the lowest upgrade effort. Since the safety of the system does not depend on the unreliable subsystem, the safety of the entire system does not have to be verified again.

Due to their limited ability to detect correctness errors, Encoded Execution and TMR are not appropriate for the TDS use case. While N-version programming and the Simplex Architecture can detect the same categories of errors, the Simplex Architecture results in lower costs for system upgrades. Since the trustworthy subsystem in this thesis is already certified and never upgraded (cf. Section 3.4), the Simplex Architecture provides an advantage over N-version programming. Therefore, we opt for the Simplex Architecture as our safety pattern.

However, it should be noted that while all of the patterns discussed aim to increase the safety of the system they are applied to, they protect against different types of safety violations. Not all of the patterns can detect correctness errors, but multiple patterns can be combined to increase the overall safety and reliability of the system. For example, the system developed in this thesis uses the Simplex Architecture, but the trustworthy subsystem independently uses a TMR setup.

3.3.3 Error Handling Strategies

If a discrepancy is detected between the output of the trustworthy and unreliable subsystems, there are different possible reactions. In a conventional Simplex system, the decision module would at this point switch to forwarding the outputs of the trustworthy subsystem. However, the safety argumentation in this use case hinges on the simplicity of the decision logic. If the decision module has to translate between EULYNX and NeuPro messages, a simpler system than a Simplex solution could be implemented that makes existing NeuPro object controllers EULYNX-compatible by translating telegrams between it and the interlocking.

1. Report TVPS as *Disturbed, Unable to be forced to clear*
2. Close Process Data Interface (PDI) Connection
3. Close RaSTA Connection

In Eu.TDS.6860, the SCI-TDS specification defines how the TDS should act in case of a critical failure of a TVPS. In case of a critical failure, the TDS reports the affected TVPS as disturbed and unable to be forced to clear. While this solution is encourages by the EULYNX standard, it requires the decision module to construct EULYNX messages. If the EULYNX standard is at any point updated, the decision module would have to be updated as well. It would therefore add significant complexity to the decision module.

The second option is to close the PDI interface, i.e. the connection underlying the SCI communication. If a field element detects an issue and can no longer maintain the SCI connection to the interlocking, it can send the `Msg_PDI_Not_Available` message to the interlocking (Eu.Gen-SCI.452) to close the connection. While this solution also requires sending an SCI message, it closes the connection meaning that no messages from the interlocking need to be interpreted. Further, the `Msg_PDI_Not_Available` message has

no payload which makes it easy to construct without implementing the entire SCI-TDS protocol.

The last option is to close the object controller's RaSTA connection. The SCI connection exists on top of a RaSTA connection. By disconnecting on the RaSTA layer, the connection to the interlocking can be severed. Since the interlocking could no longer know the state of the TDS, it would have to assume that its TVPS are no longer usable. This solution has the advantage of requiring no SCI communication in the decision module.

These different strategies are all implemented in the Simplex-based object controller and compared in Section 4.4.

3.3.4 Comparison of NeuPro and EULYNX

Since EULYNX is partially based on NeuPro, the interfaces defined by both standards are very similar. They share interface names, message types and general telegram structure.

(a) NeuPro

Byte(s)	Contents
00	Protocol Type
01 – 02	Message Type
03 – 22	Sender Identifier
23 – 42	Receiver Identifier
43 – 127	Payload (up to 85 bytes)

(b) EULYNX

Byte(s)	Contents
00	Protocol Type
01 – 02	Message Type
03 – 22	Sender Identifier
23 – 42	Receiver Identifier
43 – 1023	Payload (up to 981 bytes)

Table 3.4: Structure of a NeuPro (left) and EULYNX telegram (right)

As Table 3.4 shows, the headers of both protocols are identical, the only difference being that EULYNX supports significantly larger payloads. This is the case because the EULYNX protocols were more recently developed than the NeuPro protocols. SCI-TDS does not use this additional payload capacity. As mentioned, protocol types and message types are in the case of SCI-TDS also identical between NeuPro and EULYNX.

(a) NeuPro

Byte(s)	Contents
00	Occupancy Status
01	Ability to be forced to clear
02 – 03	Filling Level

(b) EULYNX

Byte(s)	Contents
00	Occupancy Status
01	Ability to be forced to clear
02 – 03	Filling Level
04	POM Status
05	Disturbance Status
06	Change Trigger

Table 3.5: Comparison of NeuPro and EULYNX payloads for the TVPS Occupancy Status telegram

While the headers are identical, the differences in the paylod can be classified into additional information and changed semantics.

Additional Information For many telegrams, the EULYNX specification contains additional information not present in the NeuPro version. An example of this can be seen in the TVPS Occupancy Status telegram. The payloads are shown in Table 3.5. The NeuPro version contains three fields, whereas the EULYNX version contains six. The first three are identical, but the additional ones contain diagnostic information. The EULYNX telegram includes information on the power status of the TVPS, whether it is disturbed and what caused the TVPS to send the telegram.

Changed Semantics Beyond simply adding information, EULYNX also makes changes to how some fields are interpreted. In the case of the TVPS Occupancy Status, this includes the ability to be forced to clear. In both versions, this field is a boolean value represented using one byte of data. However, in NeuPro, admissible values are 0 for *false* and 1 for *true*, whereas EULYNX uses 1 to mean *false* and 2 to mean *true*. It seems likely that these values were chosen to avoid a scenario in which an empty telegram could be interpreted as a correct value, although the correct transmission should already be handled by checksums on the RaSTA layer.

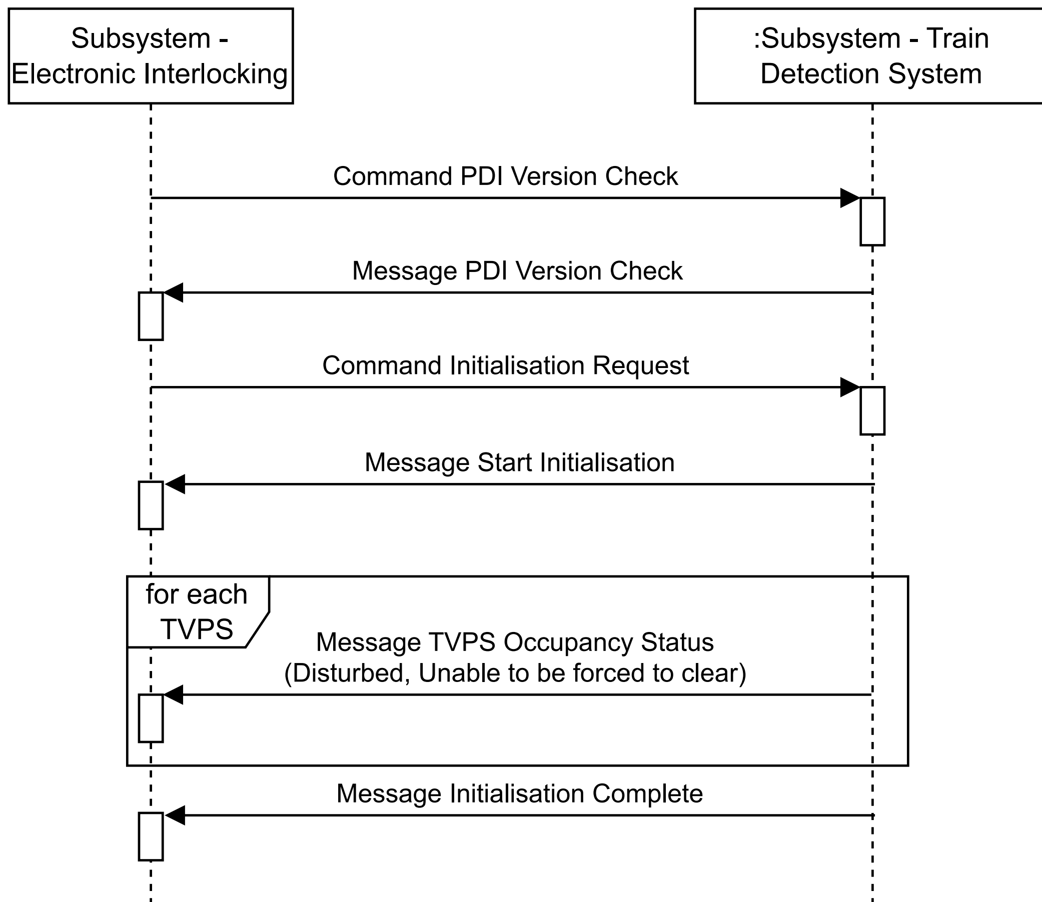


Figure 3.2: TDS Initialisation Message Flow as specified in EULYNX

In the TDS subsystem, the message flow is identical between NeuPro and EULYNX (an example for system initialization is shown in Figure 3.2). This allows the decision module to compare messages without needing to understand their exact contents except for the small number of message types considered safety-critical.

3.4 Architecture

As described above, the system developed in this thesis uses the Simplex Architecture to ensure the safety of an object controller. This means it requires an unreliable and a trustworthy object controller to use as subsystems. In order to test the system, it also requires a connection to an interlocking. This section gives an overview of the components used, their interfaces and how they are used in the Simplex-based Object Controller.

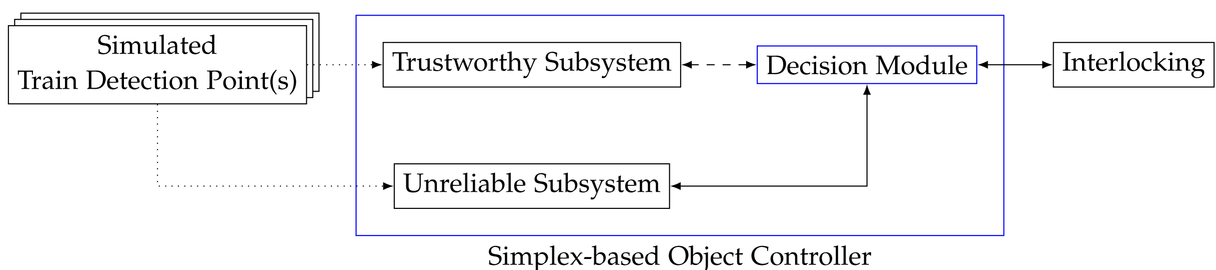


Figure 3.3: The Architecture of the entire system including the data source (Train Detection Point) and communication partner (Interlocking). Solid lines represent EULYNX SCI-TDS connections. Dashed lines represent NeuPro SCI-TDS connections. Dotted lines represent the EULYNX Live Lab TDP Simulator protocol.

An overview of the system architecture is given in Figure 3.3. Components developed as part of this thesis are highlighted in blue. The environment to develop and test the Simplex-based object controller was provided by the EULYNX Live Lab.[1] Consequently, the focus was put on the implementation of the decision module and the composition of the existing components.

3.4.1 Communication

As shown in Figure 3.3, the Simplex-based Object Controller communicates outside the system using two interfaces: The EULYNX Live Lab TDP Simulator protocol and EULYNX SCI-TDS. The first of these is not standardized, but a custom protocol developed for the EULYNX Live Lab. It can be used to increase or decrease the axle count in a TVPS.

SCI-TDS is used for the communication between the decision module and the interlocking.

Out of the messages listed in Table 3.6, messages sent by the interlocking are forwarded to the subsystems unchanged. Only messages sent by the axle counter, i.e. the subsystems, need to be checked by the decision module. Since the EULYNX axle counting simulator

[1] https://lab.eulynx.live/

EULYNX	NeuPro	Type	Description	Sender	Used
PDI-Version Check, Initialisation Request	BTP-Versionsabgleich, Aufrüstanforderung	C	Commands in the initialisation sequence	IXL	●
Close PDI, Release PDI for Maintenance	Not applicable	C	Gracefully close a PDI connection	IXL	○
Force Clear	AZG	C	Forcibly set the TVPS' status to vacant	IXL	●
Update Filling Level	Achszähl-füllstand-Aktualisierung	C	Request the current state of a TVPS	IXL	○
Disable Restriction to Force Clear	AZGH	C	Allow a TVPS' status to be forced to clear	IXL	●
Cancel	Not applicable	C	Cancel a Force Clear command	IXL	○
PDI-Version Check, Start Initialisation, Status Report Completed, Initialisation Completed	BTP-Versionsabgleich, Aufrüstbeginn, Aufrüstende	M	Messages in the initialisation sequence	ACM	●
TVPS Occupancy Status	GFM-A-Belegungs-zustand	M	Reports a change of status in a TVPS	ACM	●
TVPS FC-P failed, TVPS FC-P-A failed	Not applicable	M	Report an unsuccesful attempt to force clear	ACM	○
Additional Information	Not applicable	M	Reports additional information related to TVPS occupancy status changes	ACM	○
TDP Status	ZDP-Befahrungszustand	M	Reports status changes of a TDP	ACM	○

Table 3.6: Overview of EULYNX commands and messages. If a NeuPro equivalent exists, it is listed in the *NeuPro* column. *Type* shows if the telegram type is a message (M) or command (C). *Sender* shows if the telegram is sent by the interlocking (IXL) or axle counter (ACM). *Used* shows if the components in our architecture send the message, ● indicating the message can occur, ○ indicating it is not used.

only emits a subset of the supported messages, this reduces the number of messages required to check even further.

3.4.2 Unreliable Subsystem

The unreliable system is an axle counting simulator[2] that was developed as part of the EULYNX Live distributed laboratory infrastructure. It uses a EULYNX interface for communication with an interlocking, but only implements part of the specification. In the Simplex-based object controller, it communicates with the interlocking through the decision module using SCI-TDS. Since both the simulator and the interlocking communicate via EULYNX SCI-TDS, telegrams are forwarded by the decision module without translating them. The simulator is not connected to actual TDPs, but receives simulated axle counting events via a gRPC interface.

3.4.3 Trustworthy Subsystem

In this thesis, the trustworthy system is a Thales AzLM axle counter that has already been certified for use in Germany. It uses a NeuPro interface to communicate with an electronic interlocking. In the Simplex-based object controller, it does not communicate with the interlocking directly, but with the decision module. It receives axle counting events from its connected TDPs via a proprietary protocol. For the purpose of the evaluation, axle counting events can also be simulated.

3.4.4 Decision Module

The design and implementation of the decision module are the main contributions of this thesis. In many Simplex applications, the decision module monitors the state of the environment independently of the subsystems. Based on the environment state and the proposed actions of the subsystems, it can then decide whether to switch. This approach is, however, not applicable to the axle counting use case. To detect a potential error in the unreliable subsystem, the system compares its output to the trusted subsystem's output which is assumed to be correct. If both subsystems report an identical axle counting event within a specified time frame, the decision module can safely forward the event to the interlocking. However, if the events differ or only one subsystem reports an event, the decision module knows that an error has occurred. As discussed in subsection 3.3.4, this naive approach is possible because the NeuPro and EULYNX protocols use the same order or messages.

As discussed in subsection 3.3.1, an axle counter can only cause a safety violation by sending a false occupancy status report. Therefore, the only message type out of the ones shown in Table 3.6 we consider in detail is *TVPS Occupancy Status*. The equivalent NeuPro and EULYNX messages share three payload fields: Occupancy Status, Ability to Force Clear and Filling Level. Out of these fields, only occupancy status affects the ability to use the axle counter and must be compared between subsystems. For other messages, the decision module still compares message types to ensure the subsystems follow the same message flow.

[2] https://github.com/eulynx-live/subsystems/tree/main/src/TrainDetectionSystem

In other use cases, the decision module could then switch to the trustworthy subsystem. This is not possible for the axle counters, since the trustworthy axle counter is not EULYNX-compatible. It would be possible to implement a translation component, but this thesis opts for the Simplex approach since it offers a number of advantages:

Simpler Implementation A NeuPro-EULYNX translation component would have to be kept up to date with the latest EULYNX release. Since it would be a safety-critical part of the object controller, it would also have to be recertified with every update. The decision module as described here would be simpler to implement and updates would be constrained to the unreliable subsystem. Since the unreliable subsystem does not need to be safe, it can be updated without requiring certification.

Use of EULYNX-specific information As discussed in subsection 3.3.4, EULYNX transmits additional information about the status of TVPS. Since a NeuPro axle counter does not collect this information, it would not be available to the interlocking when using a translation component. Since the Simplex approach forwards the messages of the EULYNX axle counter so long as they do not stand in conflict with the NeuPro axle counter's messages, the additional information specified by EULYNX remains usable.

4 Implementation

This chapter describes the implementation of the Simplex-based object controller. We begin by describing the tooling and hardware used in the implementation in Section 4.1 and Section 4.2. We then discuss how our prototype implements the architecture established earlier, focusing on how patterns in the implementation reflect the safety requirements. Finally, we discuss the system's timing behavior and possible adaptation to a realtime platform in Section 4.5

4.1 Tooling

For many components of the Simplex system, this thesis reuses existing components. This is done in order to achieve a lower implementation effort and approximate a real-world scenario in which the subsystems would not be specifically tailored to the decision module.

4.1.1 RaSTA Implementation and gRPC-RaSTA Bridge

As explained in Chapter 1, the RaSTA protocol is used for network communication in railway contexts. There exist a small number of open-source RaSTA implementations. The first is written in C and maintained by Markus Heinrich.[3] The EULYNX Live RaSTA implementation[4] is based on this one, but makes changes to the software architecture, such as using an event loop instead of multiple threads. Finally, there is an implementation maintained by the Swiss Federal Railways.[5] This implementation is production quality and was developed according to DIN EN 50128 [14]. However, since it was only released late in the process of writing this thesis, it could not be used for the Simplex-based object controller.

This thesis uses the EULYNX Live RaSTA implementation for network communication. However, since the event-based programming interface makes the implementation more difficult to use, it also provides a gRPC bridge. The gRPC bridge presents an abstraction from the underlying event loop and real-time considerations (e.g. heartbeat messages).

4.1.2 SCI Implementation

EULYNX defines the SCI suite of protocols for application-layer communication. In order to compare subsystem messages, this thesis makes use of an SCI implementation developed by the author for a previous project.[6] This implementation is written in Rust which

[3] https://github.com/Railway-CCS/rasta-protocol
[4] https://github.com/eulynx-live/librasta/
[5] https://github.com/SchweizerischeBundesbahnen/sbb-rasta-stack
[6] https://github.com/ctiedt/rasta-rs

makes it directly usable for the Simplex-based object controller and uses no dependencies apart from Rust's standard library.

4.1.3 Rust

The Simplex-based object controller is implemented in Rust.[7] Rust is a systems programming language focused on memory safety and performance. This makes it an ideal tool for the development of safety-critical and realtime systems. A qualified compiler exists in the form of Ferrocene.[8]

Rust ensures memory safety through its ownership system. This system statically tracks at compile time how values are moved in memory and automatically frees memory after it is no longer used. This prevents common memory management errors, such as use after free or double free. Since the automatic memory management is not based on garbage collection at runtime, Rust programs perform deterministically and achieve a similar level of performance to C.

4.2 Hardware

The Simplex Architecture does not directly address the robustness of the hardware the system runs on. While a Simplex-based system can tolerate hardware faults in the unreliable subsystem if the subsystem runs on a different machine than the decision module, the decision module must be assumed to work reliably. In this thesis, the decision module runs on a Revolution Pi,[9] a Raspberry Pi-based computer hardened for industrial use cases.

As described in Section 3.4, the trustworthy subsystem is a fully functional object controller. As such, it runs on separate hardware from the decision module in a 2-out-of-3 redundancy configuration. It is connected to the same wired network as the decision module and communicates using RaSTA over ethernet.

The TDS simulator is provided as a Docker container. Since the decision module ensures the safety of the system as a whole, the TDS simulator does not need to run on safe hardware. In the prototype developed in this thesis, it runs on a regular, non-hardened x86-64 desktop PC. To simplify the setup of the prototype, the TDS simulator does not use a RaSTA connection to communicate with the decision module. Instead, it sends and receives SCI messages over a gRPC connection.

Similarly, the interlocking is provided as a Docker container. Since it is not part of the object controller, it can run on an arbitrary machine. In this case, it runs on the same computer as the unreliable subsystem.

[7] https://rust-lang.org
[8] https://ferrous-systems.com/ferrocene/
[9] https://revolutionpi.de/

4.3 Software Architecture

The Simplex-based Object Controller implements the architecture discussed in Section 3.4. However, we introduce a number of additional software components in order to create a more maintainable implementation.

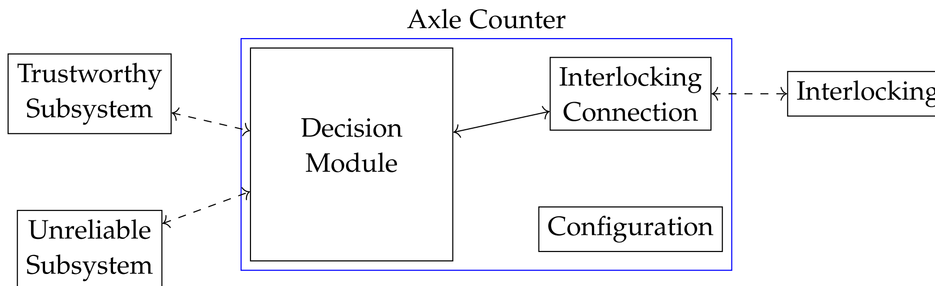


Figure 4.1: The architecture of the Simplex-based Object Controller including software components. Components in the blue box were developed as part of this thesis. Dashed lines represent RaSTA connections.

At a high level, the system is represented by the `AxleCounter` structure as shown in Figure 4.1. The `AxleCounter` maintains the system configuration, connection to the interlocking and actual decision module.

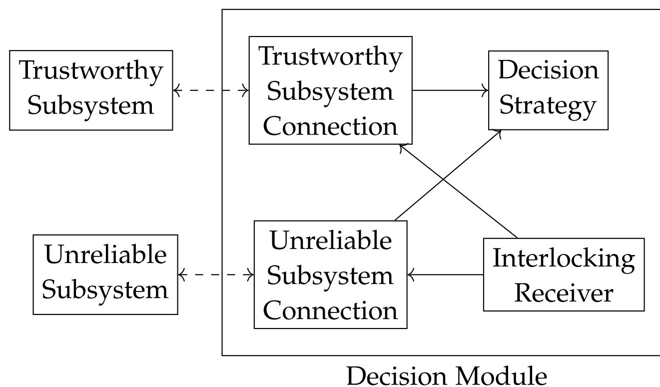


Figure 4.2: Software Architecture of the Decision Module. Dashed arrows represent RaSTA connections.

The decision module itself also contains multiple software components. These are shown in Figure 4.2:

Decision Strategy This component contains the actual behavior of the decision module. The implementation of the decision strategy can be switched to implement different behaviors for the decision module.

Interlocking Receiver The decision module receives and forwards messages from the interlocking to the subsystems through this component.

Subsystems These components maintain the connection to the actual subsystems which are connected to the Simplex-based Object Controller. They forward messages from the interlocking to these systems and forward messages from the systems to the decision strategy.

4.3.1 Axle Counter and Configuration

As discussed earlier, the system is represented at a high level by the `AxleCounter` structure.

```
rasta_id = "256"
ixl_address = "http://127.0.0.1:5100"
ixl_delay = 10000
timeout = 5000
strategy = "AlwaysUnreliable"

[trustworthy]
ixl_address = "0.0.0.0:8001"

[unreliable]
ixl_address = "0.0.0.0:8002"
```

Listing 2: Example configuration of the Simplex system

The `AxleCounter` is created from a configuration file, an example of which is shown in Listing 2. The configuration consists of the following parameters:

RaSTA ID The Simplex system's RaSTA ID. This is required for establishing a connection to the interlocking.

Interlocking Address The URL on which the interlocking listens for gRPC connections.

Interlocking Connection Delay The connection to the interlocking can only be made once the subsystems have established a gRPC connection to the decision module. This timer is used to ensure this order of operations is kept.

Message Timeout This timeout defines how long the decision module waits after receiving a message from one subsystem for a message from the other subsystem before assuming it timed out.

Decision Strategy The switching strategy to be used by the decision module. Here, it is represented by a string which is mapped to an enumeration listing all supported strategies.

Subsystem Addresses The addresses on which the gRPC servers for the subsystems listen for connections.

Notably, this configuration does not contain any information about the SCI IDs or TVPS of the subsystems. Instead, the subsystems must be configured with identically named TVPS and SCI IDs to simplify the decision module.

4.3.2 Subsystems

The subsystems are represented in code by the `Subsystem` trait as shown in Listing 3. This makes it possible to implement different subsystem connections depending on the decision module's target platform.

```
pub trait Subsystem {
    fn addr(&self) -> SocketAddr;
    fn incoming_messages(&self) -> Receiver<SCITelegram>;
    fn subscribe_to_ixl(&mut self, receiver: Receiver<SCITelegram>);
    fn start_listening(self);
}
```

Listing 3: The Subsystem trait

The subsystem connection implementation must provide methods to

1. Receive messages sent by the subsystem in the decision module
2. Forward messages from the interlocking to the subsystem
3. Connect to the subsystem and start forwarding messages

Since both subsystems are required to share the same SCI ID and TVPS names, most messages can be forwarded without any intervention. Only for messages which have different payloads in NeuPro and EULYNX, interpretation has to take place.

For the prototype presented in this thesis, the `Subsystem` trait is only implemented for the `GrpcSubsystem` structure which uses a gRPC connection to communicate with the TDS simulator or through a RaSTA bridge with the trustworthy subsystem.

4.3.3 Decision Module

The decision module is represented by a structure which contains the subsystems and a decision strategy. It forwards messages between the interlocking and subsystems, translating between EULYNX and NeuPro if necessary. Since the subsystems are connected to the Simplex system using the gRPC-RaSTA bridge, the decision module provides two gRPC servers. Messages from the interlocking are always forwarded to both subsystems, regardless of which subsystem is currently active. If a subsystem sends a message to the decision module, it is only forwarded if the subsystem is currently active.

At one time, only one subsystem is considered active. This information is stored in the decision module and managed by the decision strategy. This strategy is a structure that implements the `DecisionStrategy` trait which is shown in Listing 4.

This approach makes it possible to define different behaviors for the decision module. By defining the `INITIAL_SUBSYSTEM` associated constant, the decision strategy determines which subsystem is active when the decision module is started. While it is running, the decision module continuously awaits messages from both subsystems. After it has received a message from one subsystem, it waits until it receives a message from the other subsystem or the configurable timer has run out. The EULYNX SCI-TDS specification defines timers for various purposes, such as how long a TVPS waits until reporting an occupancy status

```
pub trait DecisionStrategy: Send + Sync {
    const INITIAL_SUBSYSTEM: ActiveSubsystem;

    fn switch_to(
        &self,
        msg_trustworthy: &Option<SCITelegram>,
        msg_unreliable: &Option<SCITelegram>,
    ) -> Option<ActiveSubsystem>;
}
```

Listing 4: The DecisionStrategy trait

change to the interlocking. These timers are configurable with a recommended range from 100 ms to 10 s. Since these value of these timers in the subsystems is unknown, the decision module's timeout can also be configured as part of the axle counter configuration. After receiving both messages or a timeout, the decision module then passes these messages to the decision strategy's `DecisionStrategy::switch_to` method. If the timer for a subsystem ran out, it instead passes a `None` value to the method. The decision strategy then decides if the active subsystem should be switched. It returns `None` if the active subsystem should not be switched, or `Some(AS)` where `AS` is the subsystem that should be switched to.

We consider the decision strategy and decision module separately here. Therefore, the decision strategy outputs whether the decision module should switch to the trustworthy subsystem, even though it closes the connection to the interlocking in practice. The default implementation of the decision strategy makes the decision whether to switch in multiple steps:

1. Did one of the systems time out? If so, switch to the trustworthy subsystem.
2. Did the subsystems send messages of different types? If so, switch to the trustworthy subsystem.
3. Did both subsystems send an occupancy status message? If not, do not switch.
4. If both subsystems sent an occupancy status message, compare the payloads. Compare only the fields that are shared between NeuPro and EULYNX, taking into account differences in semantics. If they differ, switch to the trustworthy subsystem. Otherwise, do not switch.

Other message types do not need to be compared because they either do not have any payload or only exist in EULYNX. Since the NeuPro axle counter is safe without them, we do not consider them safety-critical.

Listing 5 shows an implementation of this strategy. Note that the case in which neither subsystem sent a message is marked as unreachable. If it occurs during the system's runtime, it causes a panic thereby shutting down the system. The case is marked as unreachable since the decision strategy is only invoked if at least one subsystem sent a message. In the case where both subsystems sent a message, their message types are compared first. The `SCITelegram` type only provides convenient access to the message's

```
    fn switch_to(
    &self,
    msg_trustworthy: &Option<SCITelegram>,
    msg_unreliable: &Option<SCITelegram>,
) -> Option<ActiveSubsystem> {
    match (msg_trustworthy, msg_unreliable) {
        (None, None) => unreachable!(),
        (None, Some(_)) | (Some(_), None) => Some(ActiveSubsystem::Trustworthy),
        (Some(tw), Some(ur)) => {
            if tw.message_type != ur.message_type {
                return Some(ActiveSubsystem::Trustworthy);
            }
            if tw.message_type == SCIMessageType::scitds_tvps_occupancy_status() {
                let occupancy_status_differs = tw.payload[0] != ur.payload[0];
                if occupancy_status_differs {
                    Some(ActiveSubsystem::Trustworthy)
                } else {
                    None
                }
            } else {
                None
            }
        }
    }
}
```

Listing 5: The `switch_to` implementation of the default decision strategy

fields and does not perform any interpretation or conversion on the message. Since message types have the same numerical values between NeuPro and EULYNX, they can simply be compared. If both subsystems sent an occupancy status message, their payloads must be compared. The occupancy status itself is encoded as a symbolic one byte integer and is the same between NeuPro and EULYNX. The ability to force section status to clear is a case of changed semantics as discussed in subsection 3.3.4. In NeuPro, the values **`false`** and **`true`** are encoded as 0 and 1, whereas EULYNX encodes them as 1 and 2. In order to compare them, the value in the NeuPro message must be increased by 1. Finally, the filling level is encoded equally between NeuPro and EULYNX, allowing a comparison without any conversion.

4.4 Error Handling Strategies

So far, we have not discussed the system's behavior if the decision strategy recommends a switch. As discussed in subsection 3.3.3, multiple error handling strategies can be implemented.

```
let (t, u) = futures::join!(
    tokio::time::timeout(timeout, trustworthy_incoming.recv()),
    tokio::time::timeout(timeout, unreliable_incoming.recv())
);

let t = t.ok().map(Result::unwrap);
let u = u.ok().map(Result::unwrap); // (1)

if t.is_none() && u.is_none() { // (2)
    continue;
}

if let Some(switch) = strategy.switch_to(&t, &u) { // (3)
    error!("Decision Module requires switch to {switch:?} subsystem.");
    *active.write().await = switch;
}

match *active.read().await { // (4)
    ActiveSubsystem::Trustworthy => yield t.unwrap(),
    ActiveSubsystem::Unreliable => yield u.unwrap(),
}
```

Listing 6: Excerpt from the decision module main loop for handling messages from the subsystems.

These strategies must be implemented in the `DecisionModule::run` method which handles incoming subsystem messages. Listing 6 shows the relevant excerpt of this method. Note that the excerpt does not contain any error handling yet. This excerpt runs in an infinite loop. For each iteration, the decision module performs the following steps which are also numbered as comments in the listing:

1. Wait for messages from both subsystems or a timeout.
2. If both subsystems' timers ran out, neither sent a message. Skip the remainder of the iteration and wait for a new message.
3. Invoke the decision strategy to decide if the active subsystem must be switched. Note that the `error!` macro is only used for logging purposes and does not change the decision module's behavior.
4. Depending on which subsystem is now active, forward its message.

The possible error handling strategies described earlier are:

1. Report TVPS as *Disturbed, Unable to be forced to clear*
2. Close Process Data Interface (PDI) Connection
3. Close RaSTA Connection

The first of these is the most complex to implement, since it requires the construction of SCI telegrams with payload and it keeps the connection to the interlocking alive. An example implementation is shown in Listing 7.

```
if let Some(switch) = strategy.switch_to(&t, &u) {
    error!("Decision Module requires switch to {switch:?} subsystem.");
    *active.write().await = switch;
    error_occurred = true; // (1)
}

if error_occurred { // (2)
    let tvps_id = &t.unwrap().sender_id;
    yield SCITelegram::tvps_occupancy_status(
        tvps_id,
        ixl_id,
        OccupancyStatus::Disturbed,
        false, // Ability to force clear
        0, // Filling Level
        POMStatus::NotApplicable,
        DisturbanceStatus::Operational,
        ChangeTrigger::TechnicalFailure
    );
} else {
    match *active.read().await {
        ActiveSubsystem::Trustworthy => yield t.unwrap(),
        ActiveSubsystem::Unreliable => yield u.unwrap(),
    }
}
```

Listing 7: Implementation of the first error handling strategy.

First, it introduces a variable `error_occurred` (comment 1). This variable must be initialized outside the decision module loop so it does not get reset in every iteration. If the decision strategy requires a change of active subsystem, this variable is set to true. If an error has occurred, the decision module no longer forwards a subsystem's telegram. Instead, it creates its own TVPS occupancy status telegram which reports the status as *disturbed, unable to be forced to clear* (comment 2). Note that in order to create this telegram, the decision module needs the name of the TVPS. This implementation also simplifies the error reporting. A more correct implementation should only report the status of all TVPSs as disturbed once and only send a new status telegram after an *Update Filling Level* command from the interlocking. Since messages received from the interlocking and messages sent by the decision module are handled separately, this would be more difficult to implement. Even this implementation has the disadvantage of requiring a library to create EULYNX SCI messages. If the EULYNX specification is updated and new fields are added to the TVPS Occupancy Status message, the decision module has to be updated as well.

The second strategy closes the PDI connection to the interlocking. An implementation is shown in Listing 8.

```
if let Some(switch) = strategy.switch_to(&t, &u) {
    error!("Decision Module requires switch to {switch:?} subsystem.");
    *active.write().await = switch;
    yield SCITelegram::close(
        ProtocolType::SciTDS,
        tds_id,
        ixl_id,
        SCICloseReason::ProtocolError,
    );
    tokio::time::sleep(timeout).await;
    std::process::exit(1);
}

match *active.read().await {
    ActiveSubsystem::Trustworthy => yield t.unwrap(),
    ActiveSubsystem::Unreliable => yield u.unwrap(),
}
```

Listing 8: Implementation of the second error handling strategy.

This strategy seems similar to the first one, but is slightly simpler to implement. Since it closes the connection to the interlocking, it does not require the introduction of any new state. Instead, if the decision strategy requires a switch of active subsystem, the decision module sends the telegram to close the connection, waits for some time to allow the telegram to be sent and then exits the process. However, it still requires an SCI telegram to be constructed and sent. This has the same disadvantages with regards to updates discussed before, although the generic part of EULYNX protocols is less likely to change with updates than the field element-specific parts.

The final strategy is to close the connection on the RaSTA level. If the TDS becomes unavailable, the interlocking must assume it is disturbed. Therefore, this strategy also fulfills the safety requirements.

As Listing 9 shows, this strategy has the shortest implementation. The decision module does not maintain a direct RaSTA connection, but uses the gRPC-RaSTA bridge instead. If the decision module's process exits, the RaSTA bridge automatically terminates the connection to the interlocking. Since it does not work on the level of the SCI connection, but the RaSTA connection, it requires no additional messages to be constructed.

Since the Simplex-based Object controller cannot remain available without the unreliable subsystem, the strategies discussed here all have the same impact on the system's availability.

The argumentation for trust in the decision module relies on its simplicity. Since the third error handling strategy is the simplest to implement, we decided to use it for the Simplex-based Object Controller.

```
if let Some(switch) = strategy.switch_to(&t, &u) {
    error!("Decision Module requires switch to {switch:?} subsystem.");
    *active.write().await = switch;
    std::process::exit(1);
}

match *active.read().await {
    ActiveSubsystem::Trustworthy => yield t.unwrap(),
    ActiveSubsystem::Unreliable => yield u.unwrap(),
}
```

Listing 9: Implementation of the third error handling strategy.

4.5 Timing Requirements

After a change in track occupancy, the axle counter must notify the interlocking of the change within a manufacturer-defined time frame. The subsystems send messages after a maximum delay t_{tw} and t_{un} for the trustworthy and unreliable subsystem respectively. We define t_s to be the greater of both delays, i.e.

$$t_s = \max\{t_{tw}, t_{un}\}$$

In addition, there is a manufacturer-defined inhibition timer t_i before a message is sent. Since both subsystems are connected to the same data source, they must both send a message within $t_s + t_i$ after registering an axle counting event.

In addition to the subsystems' delay t_s, the decision module adds a network delay t_n. We define this delay as the sum of network transmission times between both subsystems and the decision module. The decision strategy also takes some execution time t_{ds}. Since this time is significantly shorter than the other timers and delays, we do not consider it. In total, this means the time t between an axle counting event and the SCI-TDS message being sent from the Simplex-based Object Controller is

$$t = t_s + t_i + t_n$$

The prototype presented here follows a best-effort approach for timing behavior. The loop and timeout for receiving messages approximate a cyclical execution. However, since the prototype runs as a standard Linux application, it cannot provide any realtime guarantees. There are two possible ways to ensure realtime properties of the decision module:

Using Linux with `PREEMPT_RT` On its own, Linux is not suitable as a realtime kernel. However, with the `PREEMPT_RT` patch set and the `SCHED_DEADLINE` scheduling policy, it can be used as one. The Revolution Pi used for the prototype already runs a Linux distribution patched for realtime support.

Using another realtime environment, such as ARINC 653 While Linux can be patched to include realtime support, there exist a number of operating systems designed from

the ground up for realtime. The ARINC 653 standard defines an interface that can be used to develop applications that leverage these realtime capabilities. Each ARINC 653 application runs in a separate partition which is scheduled periodically. This makes it possible to clearly define the timing behavior of individual applications as well as interactions between partitions.

The current prototype heavily relies on gRPC for communication between components. However, this implementation detail is not exposed to the decision module. For a production-grade implementation, the subsystem implementations could be updated to directly use RaSTA for communication without changing the decision module.

5 Evaluation

In this chapter, we evaluate if the prototype of the Simplex-based Object Controller fulfills the requirements defined in Section 3.2. We show this using a qualitative, experimental approach. Further, we evaluate the complexity of the prototype's decision module and compare it with the alternative approach of a NeuPro to EULYNX translation component.

5.1 Experimental Evaluation

In order to show that the prototype developed in this thesis fulfills the safety requirements defined earlier, we provide a qualitative, experimental evaluation. This evaluation uses a scenario based on the EULYNX standard to show that incorrect behavior in the unreliable subsystem does not impact the safety of the Simplex-based Object Controller.

5.1.1 Scenario

We base the evaluation of our prototype on the EULYNX use case for normal operation, EU.TDS.5979 [17]. This use case represents a train passing through a Track Vacancy Proving Section (TVPS).

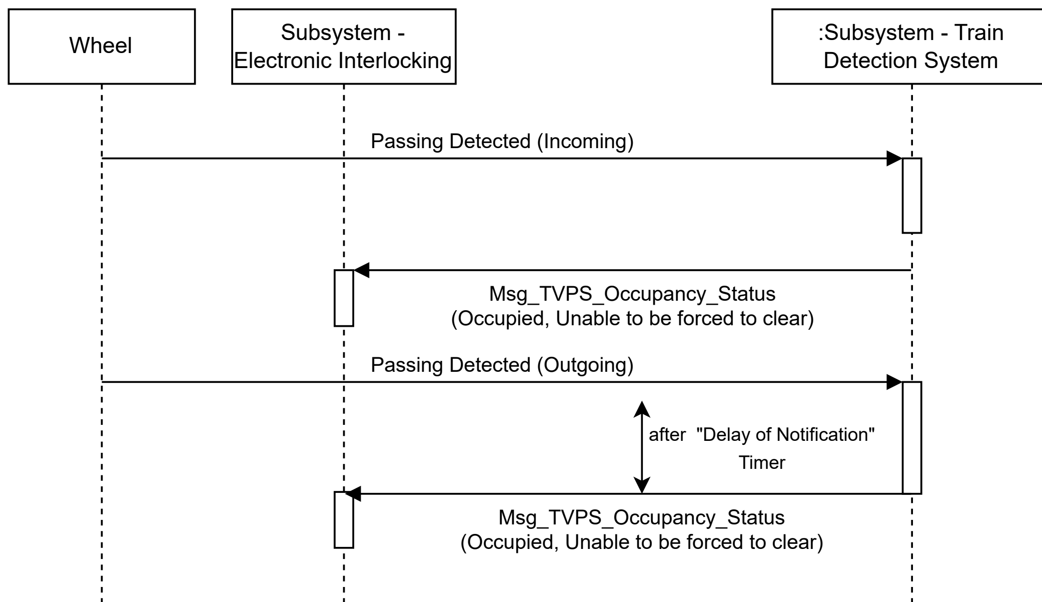


Figure 5.1: The scenario adapted from the *Normal Operation* use case (TDS UC2.1.1.1) as defined by EULYNX.

Our evaluation scenario (shown in Figure 5.1) is a simplified version of the EULYNX use case that consists of one axle entering and then leaving the TVPS. In the scenario, the TVPS must go through the following states:

1. Before the axle enters, the TVPS is in the state *Vacant, Unable to be forced to clear*.
2. After the axle has entered the TVPS, its status changes to *Occupied, Unable to be forced to clear*.
3. After the axle has exited and the preconfigured timer has run out, its status changes to *Vacant, Unable to be forced to clear*.

For each of these status changes, the system also has to meet the timing requirements defined in Section 4.5.

5.1.2 Test Cases

We define a number of test cases for the evaluation scenario based on the fault model defined in subsection 3.3.1. As discussed there, safety violations can occur if a TVPS is falsely reported as vacant.

Since both the states *Occupied* and *Disturbed* mean the TVPS in not usable, we refer to them together as *unavailable* states. The TVPS can enter an unsafe state in two cases:

1. The TVPS is unavailable and falsely reports becoming vacant.
2. The TVPS goes from vacant to unavailable, but does not report it.

	Test Case Name	Description	Expected behavior
T1	Both Correct	Both subsystems correctly register a train entering and leaving a TVPS.	The TVPS is reported as occupied and then as vacant.
T2	Unreliable Timeout	The unreliable subsystem does not register one of the axles entering and the decision module times out.	The TDS closes the connection.
T3	Incorrect Message	The unreliable subsystem registers one of the axles entering the TVPS as leaving.	The TDS closes the connection.

Table 5.1: The list of test cases.

Based on these possibilities, we define the test cases listed in Table 5.1. T1 checks if the system correctly forwards messages to the interlocking if both subsystems register the same events. T2 and T3 check if the system correctly terminates the connection to the interlocking if the unreliable subsystem does not send a message within the defined time frame or if sends a different message than the trustworthy subsystem.

5.1.3 Experimental Setup

The experimental setup uses the components described in Chapter 4. Both the trustworthy and unreliable subsystem implement the same interface for simulated TDPs. We implement the different test cases by varying the axle counting events sent to the subsystems. For T1, both subsystems receive the same events. For T2, the second axle entering is only sent to the trustworthy subsystem to cause a timeout in the decision module. For T3, both subsystems receive different axle counting events at the same time.

```yaml
- both_ok:
    unreliable: "http://100.116.102.88:5102/"
    trustworthy: "http://100.116.102.88:5210/"
    default_delay: 1
    steps:
      - increase_axle_count:
          unreliable: "99B101"
          trustworthy: "1"
      - increase_axle_count:
          unreliable: "99B101"
          trustworthy: "1"
      - decrease_axle_count:
          unreliable: "99B101"
          trustworthy: "1"
      - decrease_axle_count:
          unreliable: "99B101"
          trustworthy: "1"
```

Listing 10: Definition of the testcase for T1.

We implement these test cases using a simple framework for coordinating axle counting events. As shown in Listing 10, test cases are specified in a YAML format and consist of different steps which may be applied to one or both subsystems. The framework supports the following types of steps:

1. Increase the axle count of a section
2. Decrease the axle count of a section
3. Wait for a specified number of seconds

Between steps, a delay of `default_delay` seconds is inserted.

5.1.4 Results

In a qualitative evaluation, the Simplex-based Object Controller correctly forwarded equivalent messages in T1 and closed the RaSTA connection in T2 and T3.

This behavior reduces the availability of the object controller while keeping its safety intact. Reliability, the correctness of the output, and integrity requiring that the output is unaltered, are also traded off against availability. After a mismatch between the controllers

and the decision module severs the connection to the interlocking, both controllers and the decision module must be reset to their initial states. The additional effort required for the reset negatively impacts the system's maintainability. It may be possible to reset the subsystems remotely to reduce this burden, although our prototype does not implement this functionality.

5.2 Complexity of Decision Module

The assumption that the Simplex-based Object controller is easier to certify than one using other means of safety relies on the simplicity of the decision module. In order to validate this assumption, we evaluate the decision module's source code using the cyclomatic complexity [28] and cognitive complexity [9] metrics. While cyclomatic complexity focuses on the testability of a procedure, cognitive complexity provides a measure of how easy the code is to understand.

Cyclomatic complexity measures the number of linearly independent paths in the control flow of a program. The metric can be calculated from the control flow graph of the decision strategy shown in Figure 5.2 by counting the number of different paths from the start of the function (the tree's root node, marked in red) to a return statement (any leaf node, marked in blue). This results in a cyclomatic complexity of 6. McCabe defines values under 10 as low complexity [29], making the decision module easily testable.

Cognitive complexity is defined similarly to cyclomatic complexity, but weights some control flow concepts differently to better match an intuitive understanding of complexity [8]. This metric gives a score of 8 for the decision strategy. Since cognitive complexity penalizes nested control structures more heavily, the slightly higher result is to be expected. G. Ann Campbell, the author of [8], recommends a maximum complexity of 15 for a function [7]. The decision strategy does not exceed this value which means that the code is reasonably understandable.

The prototype decision module runs in an event loop which is not suitable for a production-grade implementation (refer to Section 4.5 for more information). Therefore, we do not consider the complexity of the part of the decision module that handles communication with the subsystems and interlocking. However, we discuss how the prototype could be ported to a realtime platform, such as ARINC 653.

In [32], Moumouris and Zehnder describe the design of an object controller built on a segregating platform. Notably, their prototype executes safety-critical and non-critical applications on the same hardware by using the operating system's segregation to isolate them. Given that the Simplex architecture also combines safety-critical and non-critical components, a similar partitioning scheme makes sense for the Simplex-based Object Controller. One possible scheme is shown in Figure 5.3.

This partitioning scheme places each component in its own partition. They can therefore be treated as independent for the purpose of updates. The unreliable subsystem can also be executed on the same hardware as the decision module. Besides the Simplex architecture components, the partitioning scheme also includes a RaSTA and TCP stack which are shared by the other partitions.

In order to port the prototype to a platform such as ARINC 653, the following work would be necessary:

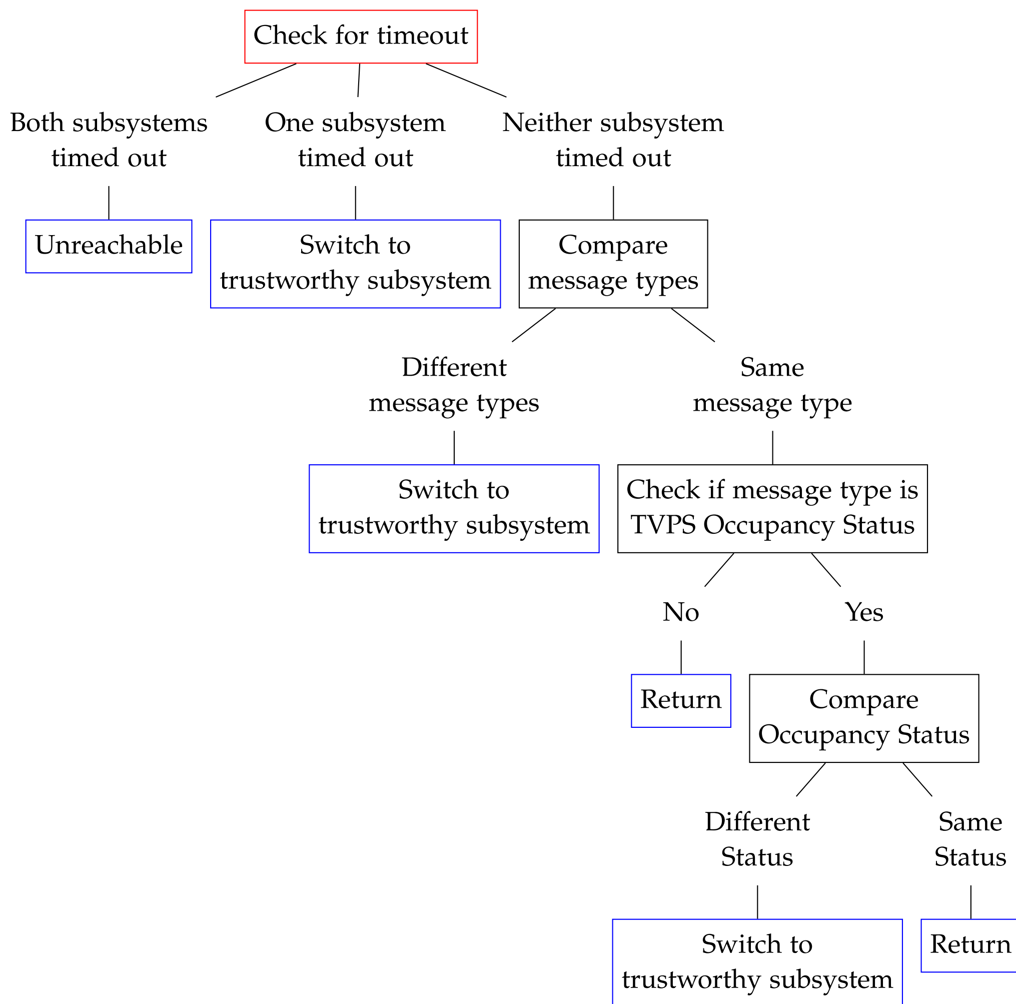


Figure 5.2: Control flow of the decision strategy implementation.

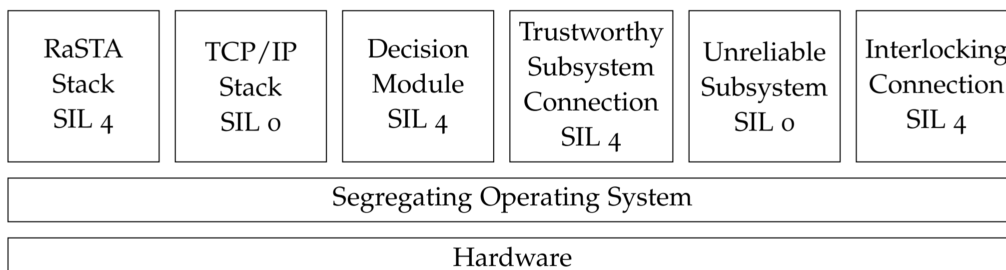


Figure 5.3: A possible partitioning scheme for the Simplex-based Object Controller on a segregating platform (such as ARINC 653).

1. Replacing tokio tasks with partitions
2. Replacing tokio timers with periodic and aperiodic tasks as appropriate
3. Replacing tokio channels with platform channels
4. Replacing the error handling

As discussed in Section 4.5, the prototype uses a best-effort approach for timing guarantees. It uses an event loop built with the tokio runtime for asynchronous tasks. In the prototype, the decision module and subsystem connections are each executed in their own tasks. On a segregating platform, these can be represented by one partition per component. ARINC 653 does not have an equivalent abstraction to timers in tokio. However, tasks can be defined to have deadlines. Further, ARINC 653 provides channels for communication between partitions. In combination, these can replicate the behavior implemented with timers in the prototype. The decision strategy could then be executed as a periodic task with a deadline equal to the timeout in the prototype. Instead of actively waiting for messages from both subsystems, the subsystem connections write messages to a channel when they arrive. When the decision strategy is scheduled to run, it attempts to read one message for each subsystem. If no message is available for a subsystem, this is treated the same as a timeout in the prototype.

5.3 Comparison with NeuPro-EULYNX translator

Another option to use an existing, trusted component (TC) with a new interface is a translation module (TM) implementing the updated interface, as shown in Figure 5.4. The same result could be achieved by updating the trusted controller itself. However, considering the trusted controller and translation module as two conceptual components makes describing and discussing the necessary changes and implications easier. Unlike the Simplex architecture, the translation-based approach has only two components, the trusted component and the translation module, on which the dependability attributes of the translation-based system depend. The Simplex approach trades off availability for safety because it cannot switch control to the trusted component. In contrast, availability will be higher in the translation-based approach, but the correctness of the system depends on the translation module as well. As illustrated in Table 5.2, to ensure the correct working of the system in the Simplex-based solution, TC and UC have to work correctly, while in the translation-based solution, TC and TM have to work correctly. However, the Simplex system remains safe even if the UC is faulty.

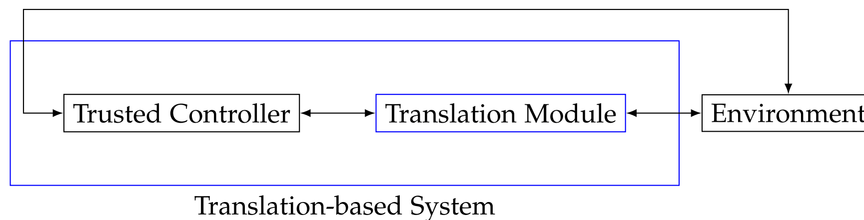


Figure 5.4: The architecture of a translation-based system.

Table 5.2: System states, which are correct, faulty, and unavailable, in dependence of the controller system states.

UC \ TC	correct	faulty	unavail.
correct	correct	unavail.	unavail.
faulty	unavail.	faulty	unavail.
unavail.	unavail.	unavail.	unavail.

(a) Simplex-based solution

TM \ TC	correct	faulty	unavail.
correct	correct	faulty	unavail.
faulty	faulty	faulty	unavail.
unavail.	unavail.	unavail.	unavail.

(b) Translation-based solution

While the translation-based solution is more straightforward and has a simpler architecture, the Simplex approach offers advantages. Updates to the interface specification are only reflected in the untrusted controller, leaving the trusted and, thus, safety-critical components untouched. This is true as long as the updated interface contains only additional, non-safety-critical information automatically forwarded by the decision module. However, if the interface update adds any safety-critical functionality, the trusted controller's safety guarantees will not suffice and recertification is necessary. Finally, a decision module can be easier to implement than a translation component. The decision module can be kept generic except for specific messages where changed semantics between interfaces must be considered.

In the translation-based approach, the translation module, which is safety-critical, always requires an update. Consequently, recertification is necessary for every update. Further, the translation module must be able to translate all messages of the older interface. Especially in cases where the interface specifications only exist in a human-readable format and must be implemented manually, this introduces another source for errors.

We did not develop a translation module to compare the prototype in this thesis against, but provide some estimations of how implementations would differ. We first consider the translation from EULYNX to NeuPro and then the other direction.

Since the shared commands between NeuPro and EULYNX have identical payloads, no translation would be necessary in this direction. There are a small number of commands only available in EULYNX (see Table 3.6), but the trusted controller would simply reject unknown commands.

Translating messages sent by the trusted controller would involve more effort. In EULYNX, the default value for *not applicable* is not `0x00`, but `0xFF`. For message types with additional fields in EULYNX, these fields would have to be set to the value for *not applicable* by the translation module. The translation module would also have to take into account fields with changed semantics (see subsection 3.3.4 for a more detailed description).

We cannot give an exact metric of how much this would increase the complexity of the translation module compared to the decision strategy. However, the fact that it not only needs to understand NeuPro messages, but transform them into EULYNX messages would likely result in a more complex implementation.

6 Conclusion and Future Work

In Chapter 3 - Chapter 5, we developed and discussed an architecture for an object controller with inherited trust. Given the prototype and its evaluation, we now provide a conclusion, taking into account the limitations of our work. We close with a look at future work.

6.1 Conclusion

This thesis discussed the challenges of digitization in the railway sector and introduced a method for mitigating them. The railway sector started to become digitized in the 1980s, but recent years saw the move from monolithic systems to component-based systems with standardized interfaces. We focussed on the following challenges:

1. Railway infrastructure must go through a lengthy and complex certification process before it can be used in the field.
2. Equipment has to be kept up to date with interface specifications, even though its fundamental tasks do not change.

We discussed these challenges for the axle counting use case with the NeuPro and EULYNX standards. Since EULYNX is based on NeuPro, the standards are partially compatible. Commands remain identical between both standards and most changes to messages do not change existing fields and only add new ones. Based on the requirements defined in Chapter 3, we concluded that the Simplex architecture is an appropriate architectural pattern for transferring the trust in an existing, certified axle counting object controller to an unproven one. We discussed how the Simplex-based object controller must react to faults, showing that so long as the trustworthy subsystem remains in a safe state, the system as a whole does as well. Since the trustworthy subsystem implements a different interface than the unreliable one, we cannot switch control to it. Instead, we trade off availability against safety by closing the connection to the interlocking in case of mismatched subsystem outputs.

We presented a prototypical implementation of this architecture in Chapter 4. The prototype makes use of standardized components, including an actual, certified NeuPro axle counter. We gave an overview of how the Simplex architecture components are represented in the prototype's software architecture. Further, we specifically evaluated implementations of the different error handling strategies discusses in Chapter 3 and discussed the timing requirements of a EULYNX object controller. Since the prototype uses an event loop and timers, it cannot make any realtime guarantees.

We presented an evaluation of our prototype in Chapter 5. Using a modified scenario from the EULYNX standard, we evaluated the Simplex-based object controller's behavior

in three test cases representing both correct and erroneous outputs from the unreliable subsystem. This qualitative evaluation showed that the system remained safe and if possible also available. We further evaluated the code complexity of the prototype's decision module, showing that it is of low complexity. In this context we also discussed how the prototype could be adapted to a safety platform and how our concept compares to a translation-based approach. While the translation-based approach does not trade off availability, it cannot make use of updated interface features and requires updates to safety-critical components after an interface update.

Limitations While the prototype developed in this thesis demonstrates that the Simplex architecture is applicable to the railway domain, it makes some trade-offs that are not possible for a production-grade safety system. The prototype runs on a realtime-capable Linux operating system, but as discussed in Chapter 4 is not developed using realtime APIs. Further, the prototype is built in the Rust programming language. While a qualified compiler exists in the form of Ferrocene, the prototype makes extensive use of third-party libraries including *tonic* for gRPC support and the asynchronous runtime *tokio*.

These limitations mean that the prototype in its current state is not certifiable. However, the changes discussed in Chapter 4 and Chapter 5 to port the prototype to a safety platform would eliminate these concerns. On a safety platform, the asynchronous functionality provided by *tokio* could be replaced by the operating system's realtime capabilities and instead of using the gRPC-RaSTA bridge, the system could use a production-grade RaSTA implementation.

6.2 Future Work

This thesis demonstrated the applicability of the Simplex architecture to the axle counting use case. However, we see great potential in the concept of inherited trust for other use cases as well. As discussed, proving the safety of an entirely new system is a complex task. Transferring the trust from an existing, trusted system can greatly reduce this burden.

Since our prototype only showed the theoretical applicability of the Simplex architecture to a railway use case, a possible next step could be to port the prototype to a certifiable platform. Experts on existing certification processes could be involved in this development process to ensure the resulting system fulfills the necessary dependability requirements.

Bibliography

[1] K. Assaf, R. Schmid, C. Tiedt, F. Reiter, D. Friedenberger, and A. Polze. "Dependable Dynamic Real-Time Systems: Examples For The Applicability Of The Simplex Architecture". In: *Submitted to ISORC, currently under review*. 2024.

[2] A. Avizienis. "Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design". In: *IEEE Transactions on Computers* C-20.11 (Nov. 1971). Conference Name: IEEE Transactions on Computers, pages 1322–1331. ISSN: 1557-9956. DOI: 10.1109/T-C.1971.223134. URL: https://ieeexplore.ieee.org/abstract/document/1671727 (visited on Jan. 15, 2024).

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. "Basic concepts and taxonomy of dependable and secure computing". en. In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004), pages 11–33. ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.2. URL: http://ieeexplore.ieee.org/document/1335465/ (visited on Nov. 27, 2023).

[4] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha. "The System-Level Simplex Architecture for Improved Real-Time Embedded System Safety". In: *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*. 123 citations (Semantic Scholar/DOI) [2023-11-27]. San Francisco, CA, USA: IEEE, Apr. 2009, pages 99–107. ISBN: 978-0-7695-3636-1. DOI: 10.1109/RTAS.2009.20. URL: http://ieeexplore.ieee.org/document/4840571/ (visited on Nov. 27, 2023).

[5] S. Bak, A. Greer, and S. Mitra. "Hybrid Cyberphysical System Verification with Simplex Using Discrete Abstractions". In: *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. 18 citations (Semantic Scholar/DOI) [2023-08-21]. Stockholm, Sweden: IEEE, Apr. 2010, pages 143–152. ISBN: 978-1-4244-6690-0. DOI: 10.1109/RTAS.2010.27. URL: http://ieeexplore.ieee.org/document/5465972/ (visited on Aug. 21, 2023).

[6] A. Bilbao, I. Yarza, J. L. Montero, M. Azkarate-askasua, and N. Gonzalez. "A railway safety and security concept for low-power mixed-criticality systems". en. In: *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*. Emden: IEEE, July 2017, pages 59–64. ISBN: 978-1-5386-0837-1. DOI: 10.1109/INDIN.2017.8104747. URL: http://ieeexplore.ieee.org/document/8104747/ (visited on Jan. 22, 2024).

[7] G. A. Campbell. *Answer to "SonarQube: Qualify Cognitive Complexity"*. July 2017. URL: https://stackoverflow.com/a/45084107 (visited on Feb. 29, 2024).

[8] G. A. Campbell. *Cognitive Complexity - a new way of measuring understandability*. Technical report. 2023. URL: https://www.sonarsource.com/docs/CognitiveComplexity.pdf.

[9] G. A. Campbell. "Cognitive complexity: an overview and evaluation". en. In: *Proceedings of the 2018 International Conference on Technical Debt*. Gothenburg Sweden: ACM, May 2018, pages 57–58. ISBN: 978-1-4503-5713-5. DOI: 10.1145/3194164.3194186. URL: https://dl.acm.org/doi/10.1145/3194164.3194186 (visited on Feb. 21, 2024).

[10] L. Chen and A. Avizienis. "N-VERSION PROGRAMMINC: A FAULT-TOLERANCE APPROACH TO RELlABlLlTY OF SOFTWARE OPERATlON". en. In: *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'*. Pasadena, CA: IEEE, 1995, page 113. ISBN: 978-0-8186-7150-0. DOI: 10.1109/FTCSH.1995.532621. URL: http://ieeexplore.ieee.org/document/532621/ (visited on Jan. 15, 2024).

[11] T. L. Crenshaw, E. Gunter, C. L. Robinson, L. Sha, and P. R. Kumar. "The Simplex Reference Model: Limiting Fault-Propagation Due to Unreliable Components in Cyber-Physical System Architectures". In: *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. 127 citations (Semantic Scholar/DOI) [2023-08-21]. Tucson, AZ, USA: IEEE, Dec. 2007, pages 400–412. ISBN: 978-0-7695-3062-8. DOI: 10.1109/RTSS.2007.34. URL: http://ieeexplore.ieee.org/document/4408323/ (visited on Aug. 21, 2023).

[12] F. Cristian. "Understanding fault-tolerant distributed systems". en. In: *Communications of the ACM* 34.2 (Feb. 1991), pages 56–78. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/102792.102801. URL: https://dl.acm.org/doi/10.1145/102792.102801 (visited on Dec. 5, 2023).

[13] A. Damare, S. Roy, S. A. Smolka, and S. D. Stoller. "A Barrier Certificate-Based Simplex Architecture with Application to Microgrids". en. In: *Runtime Verification*. Edited by T. Dang and V. Stolz. Volume 13498. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pages 105–123. ISBN: 978-3-031-17195-6 978-3-031-17196-3. DOI: 10.1007/978-3-031-17196-3_6. URL: https://link.springer.com/10.1007/978-3-031-17196-3_6 (visited on Nov. 27, 2023).

[14] *DIN EN 50128:2012-03*. Norm. Deutsches Institut für Normung e.V., 2012.

[15] B. P. Douglass. *Real-Time Design Patterns: robust scalable architecture for Real-time systems*. The Addison-Wesley object technology series. Boston, MA: Addison-Wesley, 2003. ISBN: 978-0-201-69956-2.

[16] EULYNX Consortium. *EULYNX System Architecture*. Standard Baseline 4.0. 2022.

[17] EULYNX Consortium. *Requirements specification for subsystem TDS*. Standard Baseline 4.0. 2022.

[18] P. Feth, D. Schneider, and R. Adler. "A Conceptual Safety Supervisor Definition and Evaluation Framework for Autonomous Systems". In: *Computer Safety, Reliability, and Security*. Edited by S. Tonetta, E. Schoitsch, and F. Bitsch. Volume 10488. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pages 135–148. ISBN: 978-3-319-66265-7 978-3-319-66266-4. DOI: 10.1007/978-3-319-66266-4_9. URL: http://link.springer.com/10.1007/978-3-319-66266-4_9 (visited on June 6, 2023).

[19] C. Fetzer, U. Schiffel, and M. Süßkraut. "AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware". en. In: *Computer Safety, Reliability, and Security*. Edited by B. Buth, G. Rabe, and T. Seyfarth. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pages 283–296. ISBN: 978-3-642-04468-7. DOI: 10.1007/978-3-642-04468-7_23.

[20] P. Forin. "VITAL CODED MICROPROCESSOR PRINCIPLES AND APPLICATION FOR VARIOUS TRANSIT SYSTEMS". In: *Control, Computers, Communications in Transportation*. Edited by J. .-. Perrin. IFAC Symposia Series. Oxford: Pergamon, Jan. 1990, pages 79–84. ISBN: 978-0-08-037025-5. DOI: 10.1016/B978-0-08-037025-5.50017-7. URL: https://www.sciencedirect.com/science/article/pii/B9780080370255500177 (visited on Dec. 18, 2023).

[21] G. Gala, G. Fohler, P. Tummeltshammer, S. Resch, and R. Hametner. "RT-Cloud: Virtualization Technologies and Cloud Computing for Railway Use-Case". In: *2021 IEEE 24th International Symposium on Real-Time Distributed Computing (ISORC)*. ISSN: 2375-5261. June 2021, pages 105–113. DOI: 10.1109/ISORC52013.2021.00024. URL: https://ieeexplore.ieee.org/document/9469907 (visited on Feb. 25, 2024).

[22] W. Granig, L.-M. Faller, D. Hammerschmidt, and H. Zangl. "Dependability considerations of redundant sensor systems". In: *Reliability Engineering & System Safety* 190 (Oct. 2019), page 106522. ISSN: 0951-8320. DOI: 10.1016/j.ress.2019.106522. URL: https://www.sciencedirect.com/science/article/pii/S0951832018314017 (visited on Feb. 25, 2024).

[23] L. Huang. "The Past, Present and Future of Railway Interlocking System". en. In: *2020 IEEE 5th International Conference on Intelligent Transportation Engineering (ICITE)*. Beijing, China: IEEE, Sept. 2020, pages 170–174. ISBN: 978-1-72819-409-7. DOI: 10.1109/ICITE50838.2020.9231438. URL: https://ieeexplore.ieee.org/document/9231438/ (visited on Jan. 24, 2024).

[24] A. Iliasov, D. Taylor, L. Laibinis, and A. Romanovsky. "Formal verification of railway interlocking and its safety case". en. In: *Safety-Critical Systems Club* ().

[25] M. John. *A biographical dictionary of railway engineers*. eng. Open Library ID: OL4568025M. Newton Abbot [Eng.], North Pomfret, Vt: David & Charles, 1978. ISBN: 978-0-7153-7489-4.

[26] S. Liu, M. Asuka, K. Komaya, and Y. Nakamura. "An approach to specifying and verifying safety-critical systems with practical formal method SOFL". In: *Proceedings. Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No.98EX193)*. Aug. 1998, pages 100–114. DOI: 10.1109/ICECCS.1998.706660. URL: https://ieeexplore.ieee.org/document/706660 (visited on Feb. 28, 2024).

[27] S. Liu, M. Asuka, K. Komaya, and Y. Nakamura. "Applying SOFL to specify a railway crossing controller for industry". en. In: *Proceedings. 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*. Boca Raton, FL, USA: IEEE Comput. Soc, 1999, pages 16–27. ISBN: 978-0-7695-0081-2. DOI: 10.1109/WIFT.1998.766294. URL: http://ieeexplore.ieee.org/document/766294/ (visited on Jan. 22, 2024).

[28] T. McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.4 (Dec. 1976). Conference Name: IEEE Transactions on Software Engineering, pages 308–320. ISSN: 1939-3520. DOI: 10.1109/TSE.1976.233837. URL: https://ieeexplore.ieee.org/document/1702388 (visited on Feb. 21, 2024).

[29] T. McCabe. *Software Quality Metrics to Identify Risk*. 2008. URL: http://www.mccabe.com/ppt/SoftwareQualityMetricsToIdentifyRisk.ppt.

[30] U. Mehmood, S. Bak, S. A. Smolka, and S. D. Stoller. "Safe CPS from unsafe controllers". en. In: *Proceedings of the Workshop on Computation-Aware Algorithmic Design for Cyber-Physical Systems*. Nashville Tennessee: ACM, May 2021, pages 26–28. ISBN: 978-1-4503-8399-8. DOI: 10.1145/3457335.3461712. URL: https://dl.acm.org/doi/10.1145/3457335.3461712 (visited on Nov. 27, 2023).

[31] U. Mehmood, S. Sheikhi, S. Bak, S. A. Smolka, and S. D. Stoller. "The Black-Box Simplex Architecture for Runtime Assurance of Autonomous CPS". en. In: *NASA Formal Methods*. Edited by J. V. Deshmukh, K. Havelund, and I. Perez. Volume 13260. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pages 231–250. ISBN: 978-3-031-06772-3 978-3-031-06773-0. DOI: 10.1007/978-3-031-06773-0_12. URL: https://link.springer.com/10.1007/978-3-031-06773-0_12 (visited on Nov. 27, 2023).

[32] S. Moumouris and M. Zehnder. "Integrierte Safety und Security durch softwarebasierte Segregation im EULYNX Object Controller". In: *Signal und Draht* (2023).

[33] P. Nagarajan, S. K. Kannan, C. Torens, M. E. Vukas, and G. F. Wilber. "ASTM F3269 - An Industry Standard on Run Time Assurance for Aircraft Systems". en. In: *AIAA Scitech 2021 Forum*. 16 citations (Semantic Scholar/DOI) [2023-06-15]. VIRTUAL EVENT: American Institute of Aeronautics and Astronautics, Jan. 2021. ISBN: 978-1-62410-609-5. DOI: 10.2514/6.2021-0525. URL: https://arc.aiaa.org/doi/10.2514/6.2021-0525 (visited on June 15, 2023).

[34] J. Pachl. *Systemtechnik des Schienenverkehrs: Bahnbetrieb planen, steuern und sichern*. ger. 11. Auflage. Wiesbaden [Heidelberg]: Springer Vieweg, 2022. ISBN: 978-3-658-38265-0.

[35] T. Petersen, J. Stock, and H. Federrath. *Bedrohungsszenarien für Energieinfrastrukturen*. Technical report.

[36] D. Phan, J. Yang, M. Clark, R. Grosu, J. Schierman, S. Smolka, and S. Stoller. "A Component-Based Simplex Architecture for High-Assurance Cyber-Physical Systems". In: *2017 17th International Conference on Application of Concurrency to System Design (ACSD)*. 27 citations (Semantic Scholar/DOI) [2023-11-27]. Zaragoza: IEEE, June 2017, pages 49–58. ISBN: 978-1-5386-2867-6. DOI: 10.1109/ACSD.2017.23. URL: http://ieeexplore.ieee.org/document/8104025/ (visited on Nov. 27, 2023).

[37] D. Seto, B. Krogh, L. Sha, and A. Chutinan. "The Simplex architecture for safe online control system upgrades". In: *Proceedings of the 1998 American Control Conference. ACC (IEEE Cat. No.98CH36207)*. 110 citations (Semantic Scholar/DOI) [2023-06-16]. Philadelphia, PA, USA: IEEE, 1998, 3504–3508 vol.6. ISBN: 978-0-7803-4530-0. DOI:

10.1109/ACC.1998.703255. URL: http://ieeexplore.ieee.org/document/703255/ (visited on June 16, 2023).

[38] L. Sha. "Using simplicity to control complexity". en. In: *IEEE Software* 18.4 (July 2001). 361 citations (Semantic Scholar/DOI) [2023-06-06], pages 20–28. ISSN: 0740-7459. DOI: 10.1109/MS.2001.936213. URL: http://ieeexplore.ieee.org/document/936213/ (visited on Apr. 18, 2023).

[39] M. Süßkraut, A. Schmitt, and J. Kaienburg. "Safe Program Execution with Diversified Encoding". In: *Embedded World*. 2015, page 9.

[40] H. Xiang-Dong, Y. Hui-Mei, and Z. Xiao-Xu. "Design of dual redundancy CAN-bus controller based on FPGA". en. In: *2013 IEEE 8th Conference on Industrial Electronics and Applications (ICIEA)*. Melbourne, VIC: IEEE, June 2013, pages 843–847. ISBN: 978-1-4673-6322-8 978-1-4673-6320-4 978-1-4673-6321-1. DOI: 10.1109/ICIEA.2013.6566484. URL: https://ieeexplore.ieee.org/document/6566484 (visited on Jan. 16, 2024).

[41] J. Yang, M. A. Islam, A. Murthy, S. A. Smolka, and S. D. Stoller. "A Simplex Architecture for Hybrid Systems Using Barrier Certificates". In: *Computer Safety, Reliability, and Security*. Edited by S. Tonetta, E. Schoitsch, and F. Bitsch. Volume 10488. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pages 117–131. ISBN: 978-3-319-66265-7 978-3-319-66266-4. DOI: 10.1007/978-3-319-66266-4_8. URL: http://link.springer.com/10.1007/978-3-319-66266-4_8 (visited on Nov. 27, 2023).

[42] J. Yao, X. Liu, G. Zhu, and L. Sha. "NetSimplex: Controller Fault Tolerance Architecture in Networked Control Systems". In: *IEEE Transactions on Industrial Informatics* 9.1 (Feb. 2013). 41 citations (Semantic Scholar/DOI) [2023-11-27], pages 346–356. ISSN: 1551-3203, 1941-0050. DOI: 10.1109/TII.2012.2219060. URL: http://ieeexplore.ieee.org/document/6303908/ (visited on Nov. 27, 2023).

A Appendix

Related Academic Conference Publications

This thesis is related to two conference submissions. One of these is directly based on the contents of this thesis, the other references the use case discussed in this thesis and shares a section with it.

A Safety-Critical Object Controller With Inherited Trust

This paper was written by Clemens Tiedt, Katja Assaf, Robert Schmid and Andreas Polze.

A condensed version of this thesis has been submitted to SafeComp 2024. The paper focuses on the concept of inheriting trust across interface updates and provides an example of this using the axle counting prototype developed in this thesis.

> Updating safety-critical applications, such as adapting to a changed interface specification, is expensive due to the safety assurances required. In the railway sector, a shift toward standardized interfaces is happening to enhance interoperability between different vendors' components. The main European initiative is EULYNX, whose specifications are under active development. Due to the long lifetimes of railway systems, vendors are already building EULYNX-compatible products. We describe a concept based on the Simplex architecture to develop a controller that implements an updated interface with consistent safety properties. We postulate that safety can be inherited from an existing trusted controller at the cost of reducing the overall system's availability. We show the applicability of our approach and test our hypothesis by providing a case study. In the case study, we build a EULYNX-compatible object controller for axle counter modules, a safety-critical controller for train detection that relies on the proven safety of a NeuPro object controller.

Dependable Dynamic Real-Time Systems: Examples For The Applicability Of The Simplex Architecture

This paper was written by Katja Assaf, Robert Schmid, Clemens Tiedt, Frederic Reiter, Dirk Friedenberger and Andreas Polze.

subsection 2.1.2 is quoted from the paper "Dependable Dynamic Real-Time Systems: Examples For The Applicability Of The Simplex Architecture" which has been submitted to ISORC 2024. The section was written by the author of this thesis. The axle counting use case discussed in this thesis also appears as an example application of the Simplex architecture in the thesis.

With the increasing reach and applicability of software systems, the demand for their complexity increase. Complexity can also stem from continuous changes in an application's operating environment, scope, functional requirements, or the need to address security incidents.

Therefore, it becomes harder to ensure the functional safety of a complex application before putting it into operation (*offline assurance*). Monitoring the safe operation continuously during operation could be an escape from this situation (*online assurance*).

The *simplex* architecture is one approach to enable online monitoring, upgrading, and the use of deep neural networks or COTS hardware in dependable systems. This paper presents different architecture variants and their applications. We provide a classification for these applications according to their goals, domain and architectural decisions. Concluding from the available literature, we identify gaps in the research landscape and explore additional fields for application.

Zusammenfassung

In der Eisenbahnindustrie findet aktuell ein Wandel in Richtung von standartisierten Schnittstellen zwischen Komponenten der Leit- und Sicherungstechnik statt. Die primäre europäische Intitative zur Entwicklung solcher Schnittstellen ist EULYNX, dessen Spezifikationen aktiv weiter entwickelt werden. Diese Schnittstellen erhöhen die Interoperabilität und entkoppeln die Lebenszyklen von Komponenten. Allerdings werden duch Aktualisierungen von Schnittstellen auch Aktualisierungen der Komponenten, die diese Schnittstellen verwenden, notwendig. Da funktionale Sicherheit von höchster Relevanz für die Eisenbahn ist, müssen Komponenten vor ihrer Verwendung in der Praxis einen aufwändigen Zulassungsprozess durchlaufen. Aktualisierungen von sicherheitskritischen Komponenten sind mit einer Neuzertifizierung verbunden. Allerdings ändern sich die grundlegenden Aufgaben einer Komponente selten durch Änderungen an Schnittstellen. Die meisten Aktualisierungen von Schnittstellen fügen nur Informationen (beispielsweise Diagnose-Informationen) zu Nachrichten hinzu, ändern aber nicht deren existierende Inhalte.

Diese Arbeit stellt ein Konzept basierend auf der Simplex-Architektur vor, um eine Komponente zu entwickeln, die eine neuere Schnittstelle implementiert, aber deren Sicherheitseigenschaften konsistent mit einer existierenden, vertrauenswürdigen Komponente sind. Wir stellen die These auf, dass es möglich ist, die Sicherheitseigenschaften einer existierenden Komponente auf Kosten der Verfügbarkeit zu vererben.

Wir zeigen mittels eines Prototypen für einen Achszähl-Object-Controller die Anwendbarkeit des Konzepts. Der Prototyp nutzt einen existierenden, zugelassenene Object Controller, der die ältere NeuPro-Schnittstelle implementiert, um die Sicherheit eines Object Controllers, der die neuere EULYNX-Schnittstelle implementiert, zu gewährleisten. In drei Testfällen, die auf dem EULYNX-Standard basieren, zeigen wir, dass der Prototyp sowohl korrektes als auch inkorrektes Verhalten des nicht vertrauenswürdigen Object Controllers toleriert und auf Kosten einer reduzierten Verfügbarkeit sicher bleibt. Eine Quellcode-Analyse des Prototypen mit Hilfe der zyklomatischen und kognitiven Komplexitätsmetriken zeigt, dass die Implementierung seiner zentralen Sicherheitskomponente, des Entscheidungsmoduls, eine geringe Komplexität erfordert. Der Vergleich unseres Simplex-basierten Ansatzes mit einer EULYNX-NeuPro-Übersetzungskomponente ergibt, dass unser Ansatz bei Änderungen an der Schnittstelle nicht aktualisiert werden muss und eine weniger komplexe Implementierung erfordert.

Um den Prototypen zulassen zu können, sind zwei große Änderungen erforderlich. Zum einen läuft der Prototype auf einem Standard-Linux-Rechner. Für die Zulassung müsste er auf eine sichere Plattform wie ARINC 653 portiert werden. Zum anderen nutzt der Prototyp eine Ereignisschleifenarchitektur, welche nicht echtzeitfähig ist. Sie müsste für die Zulassung in eine periodische Architektur überführt werden.

Eidesstattliche Erklärung

Hiermit versichere ich, dass meine Masterarbeit "The Simplex Architecture in Practice – Runtime Assurance for Safety-Critical Railway Systems" ("Die Simplex-Architektur in der Praxis – Konsistenzprüfung zur Laufzeit für sicherheitskritische Eisenbahnsysteme") selbständig verfasst wurde und dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt wurden. Diese Aussage trifft auch für alle Implementierungen und Dokumentationen im Rahmen dieses Projektes zu.

Potsdam, den 6. März 2024,

____

(Clemens Tiedt)