Platform-independent Safety:
Building a Portable Railway Interlocking Software

Ansgar Richter, Florian Stuhrberg, Judith Herrmann, Laura Rost, and Luke Ortlam

Professorship for Operating Systems and Middleware
Hasso Plattner Institute for Digital Engineering

The field of railway interlockings is currently experiencing a shift from specialized hardware with built-in dependability guarantees to cloud-based systems running on commercial off-the-shelf (COTS) hardware. These safety-critical guarantees now have to be ensured through software. Our project aims to identify which dependability properties are necessary for such systems and to develop a platform-independent interlocking providing these properties on COTS systems. To address this challenge, we implement an interlocking compliant with ARINC 653, a standard for safety-critical applications developed for the aerospace industry. Our approach for hardware independence involves using a middleware layer that emulates an ARINC 653 compliant system on COTS hardware. For that, we compare the open-source hypervisors POK, Xen and a653rs-linux. We find that for this project a653rs-linux is the most suitable. Using this hypervisor, we implement a working prototype, which can send and receive messages compliant with EULYNX. However, using a653rs-linux, we cannot make definite statements about safety guarantees of the system.

1 Introduction

Interlockings are safety-critical systems that control the movement of trains in a railway network. To make this possible, they are connected to field elements such as light signals, points and train detection systems. All software and hardware components used in an interlocking have to be certified for their safety guarantees. This certification process is expensive and time-consuming.

The first generation of computer-based interlockings, the electronic interlocking (ESTW), used vendor-specific interfaces for connecting to field elements, which often relied on conventional electrical switching technology and long copper cables [28].

In contrast to this, current digital interlockings (DSTW) communicate with field elements using Ethernet network technologies and open IP-based interfaces according to the EULYNX standard [10, 28]. This design provides numerous benefits. Most importantly, replacement parts can be installed much more easily without being bound to a specific manufacturer [7].

First projects are trying to expand on this with the goal of installing interlockings as virtual machines on commercial off-the-shelf (COTS) hardware, making an

"interlocking in the cloud" possible [28, 29]. However, an important question in this regard is how the dependability properties of classical interlockings can be achieved on COTS hardware.

In our project, we aim to build a portable interlocking software, where *portable* means that we have a hardware-independent layer capable of running the interlocking logic. To achieve this, one first has to understand how an interlocking operates and which dependability and safety guarantees it provides. For the hardware independent middleware, we chose to examine a standard of the aerospace industry, namely ARINC 653. We compared different implementations of this standard and decided for one to run our own interlocking logic. Both software components (the middleware and the interlocking logic) have to provide safety, availability and dependability guarantees in order to make a certification possible. It is crucial that they can both be analysed individually, which simplifies the certification process.

In section 2 the report first gives relevant background information about interlockings, EULYNX, ARINC 653 and SCORPOS. Our main part in section 3 features details about the implemented interlocking. We compare different ARINC implementations and provide reasons why we decided to use a653rs-linux as our middleware in this project in section 4. Section 5 describes two different proof of concept setups that we developed during our project. This is followed by a small outlook in section 6 and a conclusion in section 7.

2 Background

This chapter first gives relevant information about railway specific topics, such as Interlocking (subsection 2.1) and EULYNX (subsection 2.2), followed by details about the ARINC 653 standard (subsection 2.3) and one implementation of it (subsection 2.4).

2.1 Interlocking

Railway interlocking systems are crucial for the safety and efficiency of train movements [33]. An interlocking system prevents conflicts between rail vehicles by controlling their track routes. The core components of interlocking systems include the interlocking station as the main control centre and the object controllers, which in turn manage various field elements such as axle counters, points, and signalling components [33].

To track the states of these components, interlocking systems employ state machines, where each element has a defined set of states. The most common elements in these systems include:

Route is a technically secured path for train transits or train movements [4]. A route consists of one or more zones.

Zone is a section of railway tracks [5].

Transit represents a directed movement over a specific zone.

Point enables trains to switch from one track to another without stopping [6].

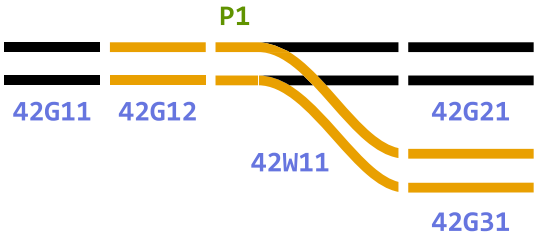

Figure 1: Example of a track section. The blue identifiers each mark a zone. There is one point marked in green. For example, if you want to set the orange route, the point P1 has to be in right position. For releasing the route again, all zones have to be free (no train inside).

Interlocking systems operate based on strict safety rules. For example, Figure 1 shows a small track section. This section includes one point (P1 marked in green) and five zones (marked in purple). Suppose you now want to set the yellow route. To do so, P1 needs to be in right position. If it is in left position, the route cannot be set. Once the point is in the correct position for this route, the route is set. For releasing the route again, all zones covered by the route have to be free. These and other predefined safety conditions ensure that the system operates with deterministic behaviour, which is crucial for quickly detecting erroneous and undefined states. To meet the high required levels of performance and safety, most interlocking systems utilize real-time operating systems (RTOS) such as PikeOS. These systems provide fast and efficient computing while ensuring compliance with stringent safety and security standards [32].

However, as technology advances, railway interlocking systems face new challenges. More complex requirements and new potential risks need to be addressed to achieve continuous improvements. A modularization approach is becoming essential for better maintenance, allowing easier updates to new safety regulations and technological inventions [33].

2.2 EULYNX

EULYNX is an initiative by 15 European infrastructure managers aimed at standardizing railway signalling systems. The primary goal is to establish a common standard for signalling interfaces between various components, such as interlocking systems and object controllers. EULYNX defines a modular architecture ensuring that standardized interfaces and message structures are used between different submodules (see also Figure 2) [12]. This allows multiple vendors to develop and

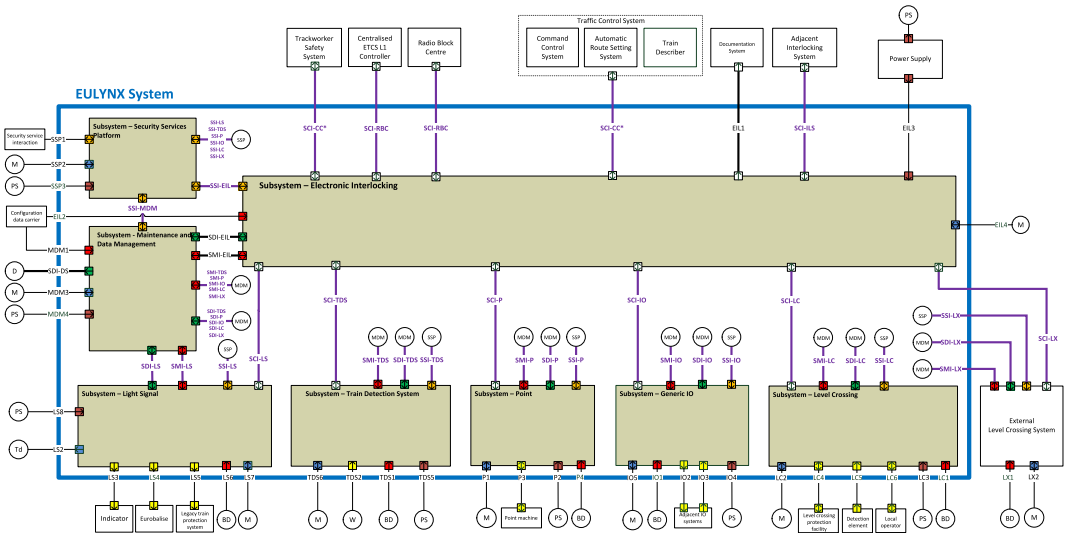

Figure 2: The EULYNX System Architecture visualizes the modular structure of the standard, including the submodules and communication interfaces between them. [11]

implement their solutions while maintaining full interoperability between different hardware and software components.

In practice, this standardization is achieved through a set of well-defined messages exchanged between the interlocking and other subsystems, i.e. the Train Detection System (TDS). These messages serve various functions, including route setting, status reporting, and safety validation. Some messages sent between the interlocking subsystem and other subsystems are:

OCCUPANCY_STATUS is sent by the TDS to the interlocking. It can be either `OCCUPIED`, `VACANT` or `DISTURBED`. This sets the status of the targeted zone accordingly [24].

REQUEST_ROUTE impacts the route and has to be sent to the interlocking when a route is requested manually. The route then switches to the state `PREPARING` or `SET`, depending if all points on the route are available and set in the correct direction. If two routes cover the same zone, only one can be set at a time, the other one will be in `PREPARING` state. To activate the second route, the first one has to be released [22].

RELEASE_ROUTE impacts the route and has to be sent when a route is manually released. The route will only be released when all zones it covers are free [22].

POINT_POSITION is a message emitted by a point subsystem to the interlocking. It contains information about the actual position of the point in the field [23].

MOVE_POINT is sent by the interlocking to the targeted point subsystem. It indicates that the specific point should be moved to the commanded position [23].

To deliver the EULYNX messages between interlocking components and object controllers safely and efficiently, the Rail Safe Transport Application (RaSTA) protocol is used. RaSTA is a network protocol specifically designed for safety-critical systems, making it particularly suitable for railway applications. RaSTA provides several key features that enhance communication reliability. Firstly, it provides reliable packet transmission to prevent data loss and corruption. This guarantees that commands and status updates always reach their intended destination. Additionally, RaSTA uses redundant communication channels to improve fault tolerance and system reliability [25].

2.3 ARINC 653

ARINC 653 is a software standard originally developed for aviation and space travel to meet the necessary safety and dependability requirements of these industries. The standard is designed to ensure high predictability in system behaviour, making it suitable for safety-critical applications where failures must be avoided at all costs [38].

A fundamental principle of ARINC 653 is the execution of programs in separation-kernel-like partitions, where they run isolated from each other. Each partition operates within its own dedicated memory space and execution time frame, preventing interference between them. The strict separation of applications enables fault tolerance, ensuring that failures in one partition do not affect others [38].

Partitions have multiple possibilities to communicate with each other. The most relevant of these are:

Queuing Ports where messages are stored in a fixed size buffer until they are retrieved [2]

Sampling Ports where only the latest message is available at any time [2]

Each partition operates within a predefined time frame. If the application finishes execution within its window, it enters a waiting state until the next cycle. If it fails to complete within the time frame, the system triggers an error-handling routine to maintain system stability [38].

Every partition has to follow a structured execution routine, consisting of two primary functions:

Appli_Init(): an initialization function, executed at the start of the partition's lifecycle [9].

Appli_Cyclic(): a function that is repeatedly called during the partition's execution time frame [9].

All system calls and function definitions, which are available in ARINC 653 compliant systems, are specified by the APEX (Application Executive) and made available in a C header file. This allow applications to interact with the system. One of the implementations of this APEX interface is SCORPOS.

2.4 SCORPOS

SCORPOS is a real-time operating system developed by Aviotech GmbH [1], specifically designed for safety-critical applications and compatible with the ARINC 653 standard. Therefore, one of SCORPOS' core features is its strict partitioning, which provides complete isolation of applications in memory and execution time. As a result, faults within one partition neither affect other partitions nor other parts of the system [13].

One notable feature of SCORPOS is the definition of the whole system ahead of time. This means the configuration is separated from the operating system. In some cases, the configuration can be updated at runtime. Additionally, SCORPOS runs on specialized hardware instead of COTS hardware [1]. SCORPOS also incorporates health monitoring at multiple levels, namely on the module level, which covers the device's hardware, the partition/application level, and the process levels. In case of faults, the system ensures secure fault handling, including the program's termination, logging, and activating a safe mode [13].

To enhance fault tolerance, SCORPOS employs a dual-lane execution system with time synchronization (see also Figure 3). This is a two out of two implementation, where the program is executed in two parallel but isolated hardware parts, so-called lanes. These lanes are constantly synchronized and compared to help detect faults, for example one instance running longer than expected, one throwing an error while the other does not, or both producing different results. This architecture improves system reliability and fault resilience [13].

3 Interlocking

In this section, we take a look at our specific interlocking implementation (subsection 3.1) based on previous work by Robert Schmid. We focus on an ETCS (European Train Control System) compliant interlocking, which does not require any signals. Instead, trains are commanded using radio [8]. In subsection 3.2, we describe how a specific interlocking can be instantiated from a formal XML description.

3.1 Concepts

Our interlocking consists of four main concepts (see Figure 4): route, zone, transit and point (see also subsection 2.1).

Each concept is formally described by one or more state machines. The state machines interact with each other by adapting to state changes of dependent components. For the route concept, we have two state machines, one modelling requesting routes and the other one modelling the automatic release of routes. The

[1] https://aviotech.de/scorpos/

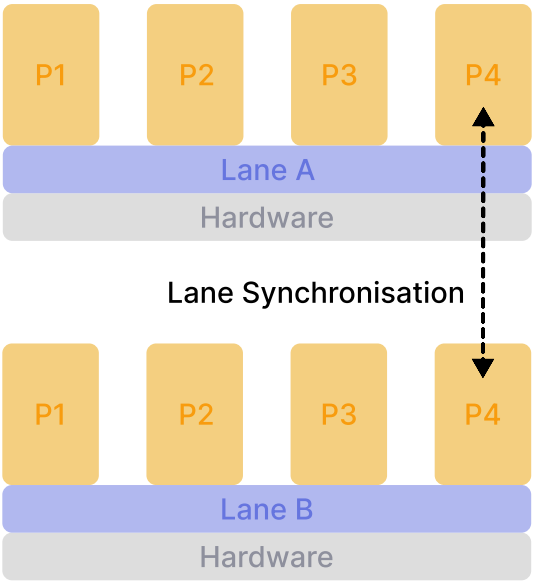

Figure 3: Illustration of the dual-lane concept employed by SCORPOS. Partitions are executed twice in isolated hardware lanes. Their results are constantly compared to and synchronized with each other to detect faults.

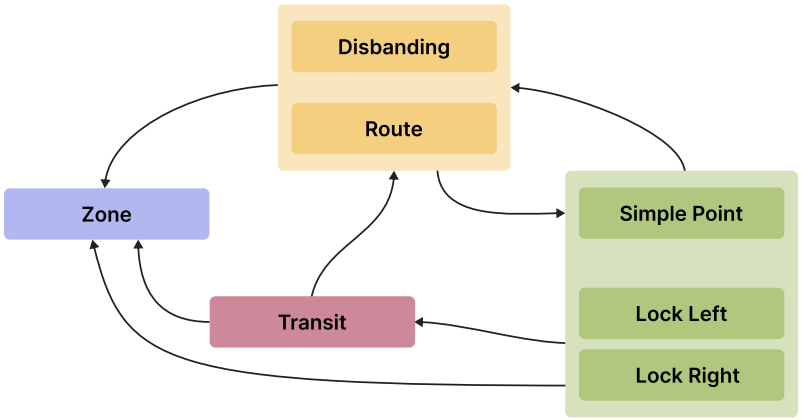

Figure 4: Main concepts and dependencies of our interlocking

zone and transit concepts are both based on one state machine. The point concept uses three state machines, one for modelling the position of the point and two more to describe whether the point can be moved from one position to the other or if it is locked in one direction.

The interlocking does not only rely on internal states, but also requires input from the outside world, i.e. the actual position of the point in the field. To incorporate this into our design, we modelled the messages according to EULYNX [10]. Our interlocking understands four messages (see also Figure 5 and subsection 2.2).

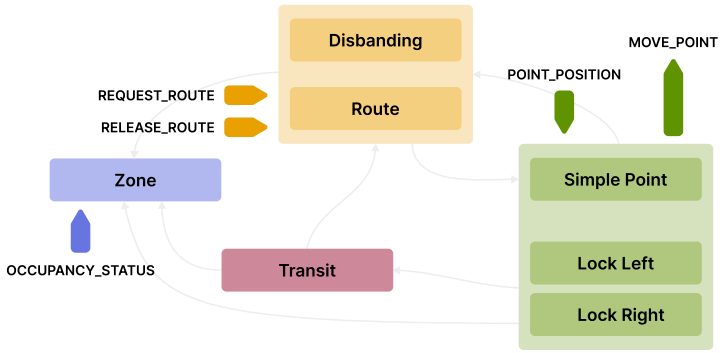

Figure 5: Messages sent and received by the interlocking components.

All state machines and the communication between them are implemented as a library in C [17]. The interlocking receives the messages detailed in Figure 5 from other components of the system. Their payload is written into a matrix `state_register`, sized appropriately for all interlocking components. During processing of the state machines, they can access these received messages, and for one complete cycle this data does not change, so all state machines access the same `state_register`. Additionally, all state machines are always triggered in the same sequence, which is: zones, route disbanding, route, transit, point lock left, point lock right, point. This is necessary for the intended determinism.

3.2 Code Generation

For instantiating a specific interlocking, i.e. a train station, we can build a fully functional ARINC partition using our `safety-generator` [19]. For that, a formal description of the interlocking in XML format is needed, defining all dependencies within the interlocking, i.e. which zones belong to which route. Our generator takes this formal description and generates necessary links between the different already implemented state machines and thus creates new instances of these state machines. The result is a C file describing all dependencies, which is compatible with our interlocking library. Additionally, the generated source file is responsible for stepping all the state machines. The generator itself is written in Rust.

4 ARINC Implementations

In this chapter, we take a closer look at different existing implementations of the ARINC 653 standard and reason why we have decided to use a653rs-linux. We ranked the different hypervisors by their completeness. The SCORPOS stub is an outlier in this regard as it was provided and recommended by Matthias Lehmann from Aviotech / University of Stuttgart.

4.1 SCORPOS Stub

In our project, we were provided with a hardware module that runs SCORPOS, but due to missing software to flash programs onto the hardware, we could not use it. Additionally, since SCORPOS is closed-source, we were not able to obtain a simulator to program against. To support the deployment of our code on the SCORPOS platform, we were provided with a simple but incomplete stub, which we used for compiling our code and checking if it could work in theory. Unfortunately, since the stub was incomplete, we could not run our interlocking software with it.

Due to these technical limitations, we were unable to use the SCORPOS system as intended. As a result, we had to search for an alternative method to run our digital interlocking system outside the SCORPOS environment in an ARINC-compatible environment.

4.2 POK

POK (a Partitioned Operating System) is an open-source real time operating system with micro-kernel architecture [37]. It is the most complete of our options in terms of the number of implemented APEX calls. POK is developed by the Télécom Paris.

While trying to install POK and running their examples, we ran into missing dependencies as documented in their GitHub issue tracker [31]. Even the proposed fix in the issue did not solve the problem of the missing Ocarina dependency. This was due to the fact that the download page was neither reachable nor archived.

4.3 Xen scheduler

Xen is a Linux Foundation project intended to create an open-source virtualization framework. Part of this framework is the Xen scheduler for ARINC 653 [30]. This scheduler comes in the form of a kernel module.

While trying to install this scheduler, the machine we installed it onto became unstable. Additionally, the second part of the installation (after installing the scheduler) turned out to be badly documented. These additional requirements and the instability made us abandon this hypervisor.

4.4 a653rs-linux

This hypervisor has been developed and is actively maintained by the DLR (Deutsches Zentrum für Luft- und Raumfahrt). It is written in Rust and almost compliant with the ARINC 653 specification, meaning that it implements all system calls relevant to us. For implementing the partitioning, it makes heavy use of linux namespaces and cgroups [9].

`a653rs-linux` only requires a linux machine and no lengthy installation. Also, examples can be run directly after cloning the repository. The declaration of a system works by specifying the layout with a yaml file. Due to easy setup and prompt support by the DLR staff, we chose this hypervisor despite the difference in programming language.

a653rs extension

While developing the partitions, we had multiple occasions where we had to debug a partition. During these sessions, we discovered multiple problems with debugging:

1. Normal (runtime) debuggers don't work because the hypervisor adds another indirection.
2. Not every log message gets printed to the screen.
3. The log messages get mixed up. This means, that if you i.e. have two partitions, log messages from partition one's first cycle end up behind the log messages of partition two's second cycle.

To help us with these problems, we extended our partitions with debugging capabilities.

Firstly, to allow for tracing of the program state, we implemented a function that could print messages to the log. This allows for basic *printf debugging* which helped us find some of the bugs. Sadly, the longer the partition ran, the more log messages got lost. We suspect that somewhere in the hypervisor some sort of buffer is overflowing because too many messages arrive. To make matters even worse, the log was not running in real time, but instead got farther behind the longer the hypervisor ran.

To fix these problems, we used the hypervisor capability of connecting to remote TCP ports. This allowed us to stream the debug messages directly to another application, which created a complete log file. Additionally, when opening the file with a tool like Visual Studio Code, the file contents were updated in real time.

Because most of our logic was captured in several different state machines, we decided to add a special debug tool for those as well: We added a message that would encode a state machine transition from one state to the next. To help us visualize these transitions, we developed a browser-based frontend (see Figure 6) which receives its updates using a websocket connection. This frontend uses the state transition messages emitted by the state machines and displays the currently active state (for more implementation details, see subsection 5.2). This helped us

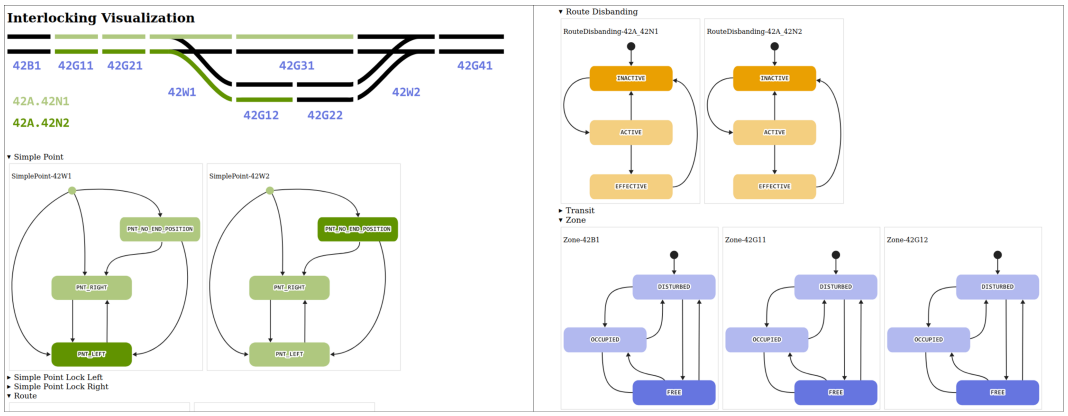

Figure 6: Two screenshots of our browser based visualization for the interlocking. All state machines for this station can be examined and they update their state in real time.

by pinpointing a problem in our specific station definition which had inconsistent switch connections.

5 Setup

We chose a653rs-linux as the hypervisor to run our partitions, for which we have two different setups: development and "production". Our development setup (subsection 5.2) is useful for debugging the interlocking logic, as it includes a graphical interface that displays all state machines and their respective states. This visualization significantly simplifies the process of identifying errors in the implementation. To enable us to trigger an actual point, we chose to integrate the EULYNX Point Server, which allows us to control a point through a REST API. The "production" setup (subsection 5.3) on the other hand is more realistic. It uses EULYNX-conform messages transmitted via RaSTA, which enables communication with other control stations, object controllers or similar.

5.1 Interlocking Partition

This partition runs our interlocking logic within the a653rs-linux hypervisor [16]. The interlocking logic is written in C. However, the hypervisor implementing the system calls uses a Rust API. This means that we need an adaptor between the Rust hypervisor and the interlocking logic. In the hypervisor, we need to call the `Appli_Init` and `Appli_Cyclic` methods of our C library. Our C Library uses APEX calls, which are implemented in Rust. Consequently, we need a bidirectional adaptor (see Figure 7). We designed the interface of this adaptor in such a way that running the partition on a regular SCORPOS system should be possible without

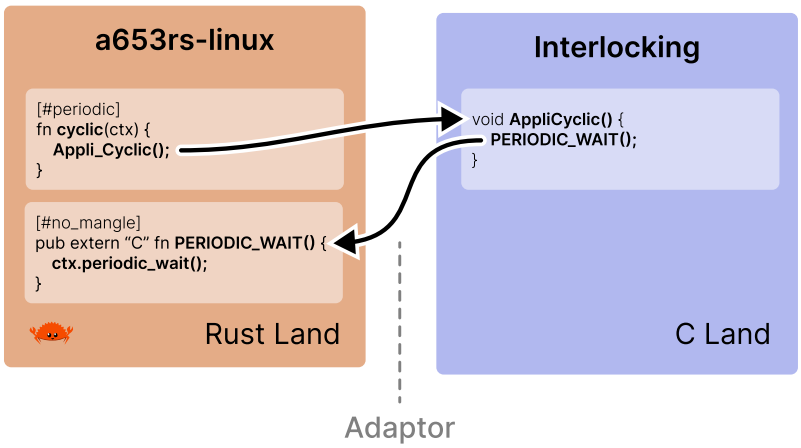

Figure 7: The hypervisor needs to call `Appli_Init` and `Appli_Cyclic` in our C partition and the partition does system calls, e.g. `PERIODIC_WAIT`, that are implemented in Rust.

changes to the C code. This is possible as the rust side of the adaptor mirrors the interface available on the SCORPOS platform.

The interlocking partition uses queueing ports to communicate with other partitions running the send and receive logic. We have one port for receiving incoming messages and one port for sending outgoing messages. These ports buffer a few messages in case the interlocking or sending logic does not work fast enough.

5.2 Development deployment

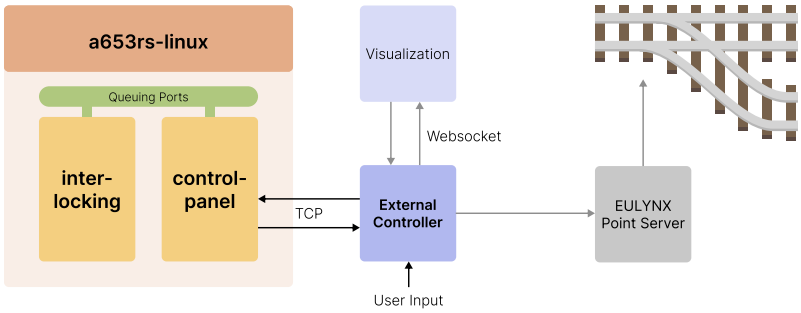

Figure 8: The development setup: We have two partitions running in the hypervisor, which communicate using queueing ports. The control input gets transmitted from an external controller to the control panel partition, which relays the information to the interlocking partition. Responses return using a channel in the opposite direction. The control commands are then sent to the EULYNX Point Server, which controls the point. Additionally, the current states of the interlocking state machines are visualized using a webpage.

The development setup uses the interlocking and the control panel partition [15] in the hypervisor. Additionally, there is the Control Panel Command Line Interface [18] (external controller in Figure 8) for taking the input from the user, relaying the messages to the actual point and visualizing the internal states of the state machines in the interlocking.

The control panel partition works as a message dispatcher between the interlocking partition and the messages from the outside world. This partition binds to a running TCP socket and receives messages (see also Figure 5) for our interlocking. The received messages are then forwarded to the interlocking partition using a queueing port. Messages from the interlocking are received via a queueing port, and forwarded to the TCP socket.

The Control Panel Command Line Interface fulfils three main tasks: command line interface, command forwarder, and debug receiver. Additionally, it provides a graphical webpage enabling the user to inspect the state of the interlocking.

command line interface accepts user input via the command line. These inputs are formatted correctly and relayed to the control panel partition via an established TCP connection.

command forwarder receives `MOVE_POINT` messages from the interlocking partition and relays them to the EULYNX Point Server [3] via a REST API.

debug receiver receives debug messages over the established TCP connection, writes them into a log file and transfers them into a websocket for display on the webpage.

The actual point commanding is executed by the EULYNX Point Server [3], which receives a GET request to a REST API containing the direction the point should be set to.

5.3 "Production" deployment

As an alternative to our control panel, EULYNX and RaSTA can be used directly to manage the interlocking software while keeping the entire protocol stack on the machine, closer to what a realistic setup could look like. Therefore, all interlocking-related communication between the host and external hardware would be via standardized EULYNX packets over RaSTA. Since our interlocking system does not require all information from the respective EULYNX messages, we created a separate partition that transforms the messages from the interlocking partition into complete EULYNX messages and back, adding and abstracting from metadata while sending and receiving as necessary [20, 26]. Examples include header identifiers and specific fields irrelevant to our scenario. EULYNX SCI-P, SCI-TDS, and SCI-CC messages will then be exchanged with the RaSTA partition [21], which handles the transport of these messages to and from other hardware. In each timeframe, we try to handle all available messages, if possible.

The RaSTA partition receives the EULYNX packets and relays them to an external RaSTA stack, which finally sends them to the object controllers (and vice

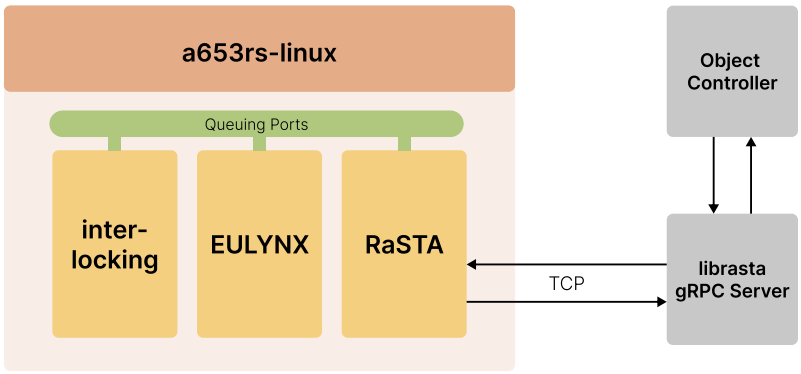

Figure 9: The "Production" setup: We have the interlocking, EULYNX and RaSTA partition, communicating using queuing ports. RaSTA messages are received by the RaSTA partition and relayed to the EULYNX partition, which passes relevant information to the interlocking partition. When receiving a message from the interlocking, the EULYNX partition packs this into a EULYNX compliant message format. The RaSTA partition then passes this message via TCP to an external RaSTA stack, which uses librasta with a gRPC bridge to control object controllers.

versa). The current implementation of this side uses the TCP socket functionality of a653rs-linux to communicate directly to a RaSTA endpoint that will encapsulate these packets for RaSTA communication. This socket server relay currently is implemented in Python [27], while the actual RaSTA transformation is accomplished using librasta [34].

Another way to handle the RaSTA layer would be to implement the RaSTA packet assembly and disassembly on the partition side and sending fully assembled packets over TCP. This is an approach followed by the SBB (Schweizerische Bundesbahnen AG) in their implementation of an object controller in the ARINC 653 environment.

Their RaSTA logic itself is open source. However, to our knowledge, there is no free implementation of this logic or an adaptor for ARINC 653. Towards the end of our project phase, the SBB and their contractor CSA supplied us with their RaSTA implementation for SCORPOS (see OCPoC). However, when trying to reproduce their setup, we encountered two major challenges.

The automated linux environment that builds all necessary artifacts and deploys them to the SCORPOS hardware platform is essential for the system to function. This automated deployment process could not be executed without access to this linux pc, even with the provided software package.

All programs on SCORPOS need to be compiled using the Wind River compiler, which provides specific compiler flags. These flags are not supported by alternative and free-to-use compilers such as GCC, meaning that switching to another compiler was not a viable option for us.

OCPoC
The RaSTA implementation of the SBB came in the form of 24 repositories called the OCPoC project. Unfortunately, the time we had left for looking into this project was very limited. Nevertheless, we want to give a brief overview of the project. The main `aviotech` repository contains the general setup for programming in the SCORPOS environment on a Windows machine, including the debugger and the ability of flashing the software to the hardware component. A submodule of `aviotech` is `sbb-poc-pil`. This contains an ARINC 653-compliant RaSTA implementation. It features the RaSTA safety retransmission and RaSTA redundancy layers. The implementation uses multiple partitions, where reusable parts are split into different submodules. Some of these submodules implement low level features, like a ring buffer or crypto primitives.

In the end, we decided not to use the SBB RaSTA implementation, but to use the previously mentioned RaSTA bridge instead.

6 Future Work

Our interlocking software prototype still leaves a wide variety of options for future work. First, our implementation can be adjusted to the *Computing Module* hardware provided to us by the SBB and Aviotech. This requires a careful analysis of the provided development environment and dependencies. Most importantly, our codebase would need to be integrated with the RaSTA implementation provided by the SBB, whose interface differs noticeably from our current RaSTA implementation.

Future projects could also focus on the topic of providing dependability properties on COTS hardware. Although this was touched on in our project, it was limited to the dependability properties provided by a standard ARINC 653 execution environment, such as time and space partitioning. Future work could investigate further approaches, for example a dual lane concept (see subsection 2.4). An implementation could be built on our work and use the a653rs-linux hypervisor, or could investigate different real-time operating systems and microkernels, such as *seL4* [14], *ThreadX* [35] or *Zephyr* [36].

7 Conclusion

In this project, we developed a prototype for a portable interlocking, running on COTS hardware using an open-source ARINC hypervisor as the hardware-independent middleware.

We chose a653rs-linux over other options due to its ease of use, excellent support and good documentation. To provide maximum system portability, our interlocking is implemented in C, targeting the ARINC 653 API. To use this partition with the hypervisor, we developed an adaptor, showing that our code is flexible enough to

be integrated into different hypervisors. Another possible execution environment could be SCORPOS, although we only targeted the SCORPOS stub.

Our chosen hypervisor provides the time and space partitioning required by ARINC using linux namespaces and cgroups. Further evaluation is needed to determine if this is enough for safety-critical systems. Additionally, we did not look into the possibility of a dual lane setup or similar.

Our interlocking uses state machines derived from EULYNX to allow only predefined states and transitions. Furthermore, we added debugging capabilities to our partitions allowing real-time debugging of the interlocking state machines. With our tooling and code-generation, we were able to find issues in our interlocking and in the given railway station XML definitions.

Additionally, we added partitions to this interlocking logic to handle the communication with field elements and control stations using the open standards EULYNX and RaSTA. With a development setup that used external implementations of these standards, it was possible to operate a real point using our interlocking in a laboratory environment.

Our solution provides a good baseline for future projects wanting to investigate additional dependability properties. Further approaches for implementing the interlocking logic in other simulation environments or the SCORPOS platform can be pursued. These approaches will facilitate the use of COTS hardware for interlocking systems, eventually making the "interlocking in the cloud" a reality.

References

[1] Aviotech. *SCORPOS*. https://aviotech.de/scorpos/. last accessed: 10.03.2025.

[2] A. Biondi. *ARINC 653*. https://retis.sssup.it/~giorgio/slides/cbsd/Biondi3-ARINC.pdf. last accessed: 25.02.2025.

[3] A. Boockmeyer. *Eulynx Point Server*. https://gitlab.hpi.de/osm/eulynx-point-server. last accessed: 25.02.2025.

[4] DB InfraGO AG. *Definition 'Fahrstraße' im Bahnbetrieb, Richtlinie 482.0009*. https://www.fahrdienstleiter.net/fdl/definition-fahrstrasse-bahnbetrieb. last accessed: 25.02.2025.

[5] DB InfraGO AG. *Definition 'Gleisabschnitt' im Bahnbetrieb, Richtlinie 482.0009*. https://www.fahrdienstleiter.net/fdl/definition-gleisabschnitt-bahnbetrieb. last accessed: 25.02.2025.

[6] DB InfraGO AG. *Definition 'Weiche' im Bahnbetrieb, Richtlinie 482.0009*. https://www.fahrdienstleiter.net/fdl/definition-weiche-w-bahnbetrieb. last accessed: 25.02.2025.

[7] Deutsche Bahn AG. *Digitale Stellwerke (DSTW)*. https://digitale-schiene-deutschland.de/de/technologien/DSTW. last accessed: 25.02.2025.

[8] Deutsche Bahn AG. *European Train Control System (ETCS)*. https://digitale-schiene-deutschland.de/en/technologies/ETCS. last accessed: 10.03.2025.

[9] DLR Institute of Flight Systems. *ARINC 653 Hypervisor for Linux*. https://github.com/DLR-FT/a653rs-linux. last accessed: 03.03.2025.

[10] EULYNX consortium. *About Us - EULYNX*. https://eulynx.eu/about-us/. last accessed: 25.02.2025.

[11] EULYNX consortium. *EULYNX System architecture*. https://eulynx.eu/storage/simple-file-list/General-Documents/Baseline-set-4-Release-3/20240618-EULYNX-System-Definition-Appendix-A1-Eu_Doc_7_A1-v4_2-2_A.pdf. last accessed: 25.02.2025.

[12] EULYNX consortium. *What is EULYNX?* https://eulynx.eu/about-us/. last accessed: 12.03.2025.

[13] M. Lehmann. *Scorpos User Guide*. last accessed: 25.02.2025.

[14] LF Projects, LLC. *The seL4 Microkernel*. https://sel4.systems/. last accessed: 04.03.2025.

[15] L. Ortlam and J. Herrmann. *ARINC-rs partitions (control-panel)*. https://gitlab.hpi.de/osm/mp2024/arinc-facade/-/tree/control-panel. last accessed: 25.02.2025.

[16] L. Ortlam and J. Herrmann. *ARINC-rs partitions (interlocking)*. https://gitlab.hpi.de/osm/mp2024/arinc-facade. last accessed: 25.02.2025.

[17] L. Ortlam and J. Herrmann. *interlocking*. https://gitlab.hpi.de/osm/mp2024/interlocking. last accessed: 10.03.2025.

[18] L. Ortlam and J. Herrmann. *Rail Control Panel*. https://gitlab.hpi.de/osm/mp2024/simple-rail-control-panel. last accessed: 25.02.2025.

[19] L. Ortlam and J. Herrmann. *safety-generator*. https://gitlab.hpi.de/osm/mp2024/safety-generator. last accessed: 25.02.2025.

[20] L. Ortlam, J. Herrmann, A. Richter, L. Rost, and F. Stuhrberg. *ARINC-rs partitions (eulynx)*. https://gitlab.hpi.de/osm/mp2024/arinc-facade/-/tree/eulynx. last accessed: 12.03.2025.

[21] L. Ortlam, J. Herrmann, A. Richter, L. Rost, and F. Stuhrberg. *ARINC-rs partitions (rasta)*. https://gitlab.hpi.de/osm/mp2024/arinc-facade/-/tree/rasta. last accessed: 12.03.2025.

[22] Platform 21. *Interface specification SCI-CC*. https://platform21.app/specifications/eulynx/baseline/EulynxBaseline4R3/144. last accessed: 25.02.2025.

[23] Platform 21. *Interface specification SCI-P*. https://platform21.app/specifications/eulynx/baseline/EulynxBaseline4R3/164. last accessed: 25.02.2025.

[24] Platform 21. *Interface specification SCI-TDS*. https://platform21.app/specifications/eulynx/baseline/EulynxBaseline4R3/141. last accessed: 25.02.2025.

[25] relesoft. *Notes on implementing RaSTA protocol*. https://relesoft.io/2024/08/notes-on-implementing-rasta-protocol/. last accessed: 25.02.2025.

[26] A. Richter, L. Rost, and F. Stuhrberg. *eulynx-impl*. https://gitlab.hpi.de/osm/mp2024/eulynx-impl. last accessed: 12.03.2025.

[27] L. Rost. *TCP to RaSTA gRPC*. https://gitlab.hpi.de/osm/mp2024/tcp-to-rasta-grpc. last accessed: 12.03.2025.

[28] Siemens Mobility GmbH. *Digitales Stellwerk Warnemünde*. https://www.mobility.siemens.com/global/de/portfolio/referenzen/digitales-stellwerk-warnemuende.html. last accessed: 25.02.2025.

[29] S. Steffens and W. Valvoda. "The development of the new DS3 safety platform – from the research project to commissioning". In: *Signal + Draht* 6/2021 (113), pages 52–59.

[30] N. Studer. *What is the ARINC653 Scheduler?* https://xenproject.org/blog/what-is-the-arinc653-scheduler/. last accessed: 25.02.2025.

[31] syrba4eva. *Can't run examples*. https://github.com/pok-kernel/pok/issues/24. last accessed: 25.02.2025.

[32] SYSGO. *ARINC 653 on PikeOS*. https://www.sysgo.com/arinc-653. last accessed: 12.03.2025.

[33] SYSGO. *Interlocking in the Railway Sector*. https://www.sysgo.com/blog/article/interlocking-in-the-railway-sector. last accessed: 12.03.2025.

[34] Systems Lab 21. *librasta: A C implementation of the RaSTA protocol stack*. https://github.com/eulynx-live/librasta. last accessed: 11.03.2025.

[35] The Eclipse Foundation. *Eclipse ThreadX*. https://threadx.io/. last accessed: 04.03.2025.

[36] The Linux Foundation. *Zephyr Project*. https://www.zephyrproject.org/. last accessed: 04.03.2025.

[37] The pok developers. *POK, a real-time kernel for secure embedded systems*. https://pok-kernel.github.io/. last accessed: 03.03.2025.

[38] windriver. *What Are ARINC 653–Compliant Safety-Critical Applications?* https://www.windriver.com/solutions/learning/arinc-653-compliant-safety-critical-applications. last accessed: 25.02.2025.