Journal of the Brazilian Computer Society

RESEARCH Open Access

Development and validation of a safe communication protocol compliant to railway standards

Duccio Bertieri[1*], Andrea Ceccarelli[1], Tommaso Zoppi[1], Innocenzo Mungiello[2], Mario Barbareschi[2] and Andrea Bondavalli[1]

* Correspondence: duccio.bertieri@unifi.it
[1]Department of Mathematics and Informatics, University of Florence, Viale Morgagni 65, 50134 Florence, Italy
Full list of author information is available at the end of the article

Abstract

Railway systems are composed of a multitude of subsystems, sensors, and actuators that exchange datagrams through safety-critical communication protocols. However, the vast majority of these protocols rely on ad hoc interlacing mechanisms and safety codes which raise the heterogeneity and complexity of the overarching railway system. Therefore, Rete Ferroviaria Italiana, the company who is in charge of managing the Italian railway network, coordinated the definition of the Protocollo Vitale Standard (Standard Vital Protocol). This protocol is inspired to, and compliant with, the communication protocols adopted for the European Train Control System (ETCS) (SUBSET, UNISIG, 037, Euroradio FIS, version 2.3. 0; SUBSET, UNISIG, 098, RBC-RBC safe communication interface, 2007), and it is meant to become the standard layer to enable safe communication between components of the Italian railway system. This paper reports our experience in the design, implementation, verification, and validation of the Protocollo Vitale Standard in compliance with the European safety standards for railway systems. We first defined a safety plan and a verification and validation plan, which guide the design, development, verification, and validation activities as required by safety standards. Guidelines of such plans have been followed strictly until completion of the work, which concludes with the provision of a safety case where all safety evidences are summarized. Noticeably, we (i) selected appropriate safety mechanisms, (ii) verified the software design, (iii) implemented the software in compliance with code metrics and coding rules, (iv) conducted tests to validate the protocol against its functional and performance requirements, and ultimately (v) devised all relevant documentation and a safety case which summarizes the evidences needed for certification.

Keywords: Railway, Communication protocol, Safety, Security, Safety integrity level, Verification and validation, CENELEC

Introduction

Safety-critical systems must adhere to appropriate guidelines to ensure that safety requirements are met. In fact, following the definition of safety, i.e., avoidance of catastrophic failures [1, 2], any misbehavior shall not lead to fatalities, severe injuries, or

Springer Open

© The Author(s). 2021 **Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

major damages to the environment [3, 4]. In other words, a safety-critical system must be able to mitigate and manage potential catastrophic failures. In addition, security breaches or vulnerabilities could also lead to unsafe behaviors; therefore, safety-critical systems may also be required to guarantee security, often with a particular accent on *integrity* of information. Safety standards typically define a *safety integrity level (SIL)* [5, 6], that sets qualitative and quantitative constraints that must be met to ensure the safe behavior of a target component or system. Matching the requirements of a desired safety integrity levels mean that a specific set of processes and techniques have been applied through the lifecycle of the system, and such application is documented by a specific list of work products.

Safety standards for the railway domain

In the railway domain, different devices, actuators, or control systems need to cooperate with each other to provide safe functionalities. In order to be actually approved and deployed in Europe, railway systems and protocols must be designed, implemented, verified, and validated in compliance with the *CENELEC* standards. Noteworthy, safety of communication for electronic and electrotechnical equipment in the railway is regulated by the CENELEC EN-50159 [7]. In addition, standards CENELEC EN-50126 [6], EN-50128 [8], and EN-50129 [9] describe the required processes and techniques for the entire lifecycle of system, software, and hardware.

Interlacing and communications

When these devices are placed in different locations, they must communicate remotely through appropriate transmission systems such as *LAN*, *WAN*, or *GSM-R* [10]. The data sent over the transmission system is often critical, containing information that will be used to perform safety-related activities e.g., changing the state of a rail switch. Consequently, such components or subsystems cannot rely on basic protocols developed for the exchange of non-critical information as *TCP/IP* [11], as they do not offer sufficient safety guarantees. Protocols to be exercised in a safety-related environment should comply with the standards that are defined for the specific domain. Typical protocols that take part to the deployment of a railway system are *Euroradio* [12], used for the GSM-R transmission system in the European Train Control System (*ETCS*) level 1 [13], and the protocol which manages communications between *Radio Block Centres* (RBCs) [14], servers in charge of continuously transmitting to trains their speed limit and the movement authority.

Our contribution

This paper documents our experience in the design, implementation, verification, and validation of software for the railway domain. We show how to follow applicable standards, motivate key choices, and show several outcomes of V&V activities. The overall process is applied to the *Protocollo Vitale Standard* (PVS), a communication protocol that provides a safe interface between railway actuators, sensors, and control systems. The protocol was specified by Rete Ferroviaria Italiana (RFI, the company who manages the Italian railways) as a standard communication protocol built upon existing—albeit heterogeneous—protocols as [12, 14]. Our work is based on [6, 7] and follows the lifecycle described in the *CENELEC* standards, starting from the Software Requirements phase up to the step 6.B–Software Validation, but does not account for further steps, which are currently managed by the owner of the case study. In order to be applied into railway systems, PVS should conform to SIL4, which is the highest SIL defined by CENELEC standards. We devised a verification, validation (V&V) and safety plan that describes, among others, techniques for software architecture, design specification, implementation, and test specification and execution. This paper extends the work in [15], where we provided a preliminary explanation of activities performed during the project's lifecycle. Instead, this work depicts the whole process and set of techniques and methodologies applied to develop the protocol. Relevant additions include (i) a revised architecture specification, (ii) a detailed dynamic analysis including the resulting coverage analysis and related discussion, (iii) conclusive results on static analysis and related discussion.

Paper structure

This paper is structured as follows: "Control systems and railway standards" section describes relevant systems in the railway domain, along with applicable standards. "Specification of Protocollo Vitale Standard" section describes the main design patterns, strengths, weaknesses, and range of applicability of the PVS. The "Verification, validation, and safety plan" section expands the V&V plan and the safety plan that describe the documents to be produced and the techniques selected for each phase. The application of such techniques is detailed in "Architecture specification" and "Design specification" sections, letting "Verification and validation activities" section report on code metrics and testing. Performance analyses are expanded in "Performance analysis of the protocol" section, while "Concluding remarks" section concludes the paper.

Control systems and railway standards

In this section, we report on control systems that are relevant for European railways, alongside with a summary of the CENELEC standards which provide prescriptive guidelines.

Railway systems

The realization of the European Railway Traffic Management System (ERTMS, 0) requires the deployment and interaction of heterogeneous devices, sensors, actuators and embedded systems, often with human interactions, e.g., the train driver. Among the different components, the Station Control Systems (SCSs) provide *computer-based interlocking* to (i) manage the interactions between trains and physical devices, e.g., semaphores, railway switches, as well as (ii) connecting the train with peripheral or central control systems. To avoid collisions, trains periodically ask the *movement authority*, which grants or denies authorization to move forward through their route. The Radio Block Centers (RBCs) instead are trackside units that interact with central offices and SCSs to provide feedbacks through the GSM-R network [14].

Currently, two ERTMS functional levels implement different communication strategy between trains and control systems. In *ERTMS Level 1*, trains communicate with the *Balise Transmission System* (BTM) [16], which relies on rail beacons called *Eurobalises* that measure speed and location and provide them as telegrams to trains as soon as they pass by. As it can be seen in Fig. 1, ERTMS level 2 is instead a digital radio-based system, where data are handled by the RBCs. With this configuration, eurobalises are considered reference points for correcting distance measurement errors. Since radio transmission systems are susceptible to a number of problems related to security aspects and integrity of data, ERTMS level 2 forces devices and subsystems to adopt the *Euroradio FIS* [12] (Euroradio Functional Interface Specification). This enables communications between critical components to be carried out in accordance to the safety standards specified by the *Category 3* (open networks) of EN50159 [7].

However, while Euroradio FIS is a requirement for Over-The-Air communication, there are no unified standards to protect communications between other entities e.g., between actuators and interlocking. Therefore, each manufacturer devises its own protocol and deploys it wherever it is needed. The lack of a unique safe communication protocol was the pilot reason for the definition of the Protocollo Vitale Standard (PVS) specification, which generalizes the Euroradio protocols and is intended to be applied in all the communications in the Italian railway system. For example, PVS can act as safe and secure channel for the communication between RBCs and the SCS, or more in general to interlace various control/signalling systems produced by different manufacturers.

Applicable standards

Safety-critical systems need to be designed, developed, and deployed in compliance with standards, which provide appropriate guidelines. Focusing on the railway domain, the CENELEC EN50126 [6] standard defines the safety integrity levels (SILs) and the corresponding lifecycle for hardware-software systems. The standard defines activities to be performed through the lifecycle, ranging from System Definition to Operation and Maintenance. As a result, several documents are produced, such as (i) the hazard log, where all the possible situations that might lead to damage to the health of people, properties, or the environment (*hazards*) are listed alongside with countermeasures and mitigations; (ii) the safety requirements document, which reports actions to reduce the

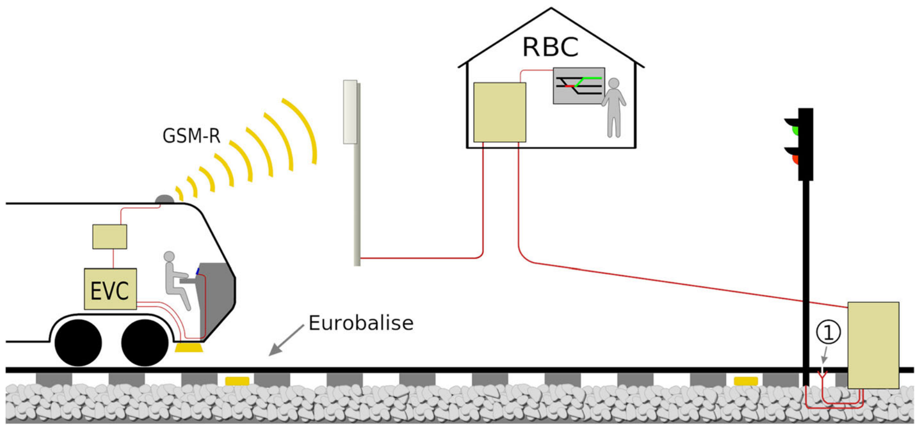


Fig. 1 ERTMS level 2. https://www.rfi.it/it/Sicurezza-e-tecnologie/tecnologie/ccs/ertms.html

severity and the probability of occurrence (*risk*, [5]) associated to each hazards; (iii) the safety plan, which guides resources management, the activities, and the responsibilities involved to reach a given SIL, and (iv) the safety case, which provides evidence that actions required in the safety plan were correctly executed and safety measures are effective and in place.

When the system contains (or is) software, compliance should also be met with the EN50128 [8], which focuses on the development of *software for railway control and protection systems*. Among others, the standard drives the definition of a verification and validation plan (V&V Plan), which specifies all the activities prior, during and after the design and development of the system. EN50128 expands on a wide set of techniques to be applied during the development lifecycle, which can be selected according to the desired SIL.

In addition, also interactions and communications between system components should be strictly regulated to avoid unsafe situations. To such extent, EN50159 [7] describes how to manage and protect safety-related communications depending on the underlying transmission system and its characteristics. EN50159 identifies accidental and malicious *threats* to *safety-related* communications and recommend possible countermeasures.

Specification of Protocollo Vitale Standard

The Protocollo Vitale Standard (PVS) is a communication protocol that lays over the ISO/OSI Transport and implements the Session and Presentation ISO/OSI levels [17]. Both TCP and UDP transport mechanisms are supported by this protocol, which defines slight variations of its structure to perfectly suit either of the two. Overall, PVS is a light protocol that can wrap communications between (i) Station Control Systems (SCSs), (ii) SCSs and RBCs, and (iii) SCSs and Graphical Elaboration Assembly (GEA). While protocols [12, 14] are tailored for their respective domains, they may not be portable to a slightly different context. As an additional merit, PVS can deal with both *Category 1* (closed transmission system, e.g., private LAN) and *Category 3* (open transmission system, e.g., public Wi-Fi) transmission systems as specified in [7].

PVS layers

The reference architecture of the PVS (see Fig. 2) is designed [18] as a stack composed by three mandatory layers, plus an optional one used to provide additional cryptographic functionalities. In this way, the PVS can deal with both Category 1 (closed network) and 3 (open network) transmission systems, providing defences against (i) data *corruption*, (ii) *resequencing*, *repetition*, *insertion*, *delay*, and *deletion of packets*, and (iii) *masquerade* in compliance with EN50159 [7]. Each layer has different responsibilities:

- Safe application interface layer (SAI): manages the data flow between the application and other PVS layers by applying several defense mechanisms, namely (i) *sequence number*, that protects against resequencing, repetition, insertion, and deletion, and (ii) *execution cycle*, that ensures “freshness” to shield against delay attacks.

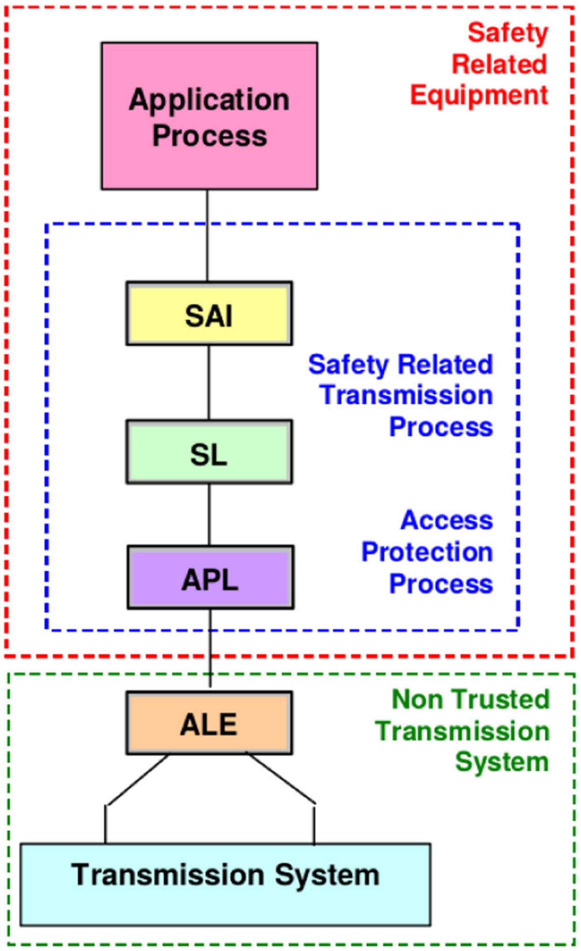

Safety Related Equipment
Application Process
SAI
Safety Related Transmission Process
SL
Access Protection Process
APL
Non Trusted Transmission System
ALE
Transmission System

Fig. 2 Reference architecture of PVS [18]

- Safety layer (SL): placed between SAI and APL layers, it ensures message integrity through *non-cryptographic safety code* and *authenticity* by means of a node identifier, called *nSAcePID*.
- Adaptation and redundancy management layer entity (ALE): manages redundancy (for *availability* purposes) by operating on two physical communication links and guarantees correct interfacing with the underlying transport mechanism, either TCP or UDP.
- Access protection layer (APL–optional): required only when dealing with Category 3 (open) transmission systems; it provides *access protection* through *AES* [19] and *AES-CMAC* [20] cryptographic algorithms.

The communication is *cyclic* with configurable period, which is set independently for each device: typically, railway devices range from 250 to 500 ms. Each *producer* must periodically send the information according to its own cycle time, which will be processed by the *consumer*. Symmetrically, each consumer must acquire cyclically, according to a period independent of the producer's cycle, the information sent by the producer. When an *application* wants to send *user_data*, the bytes flow goes through the protocol stack in Fig. 3 where each layer wraps the message by adding its header/

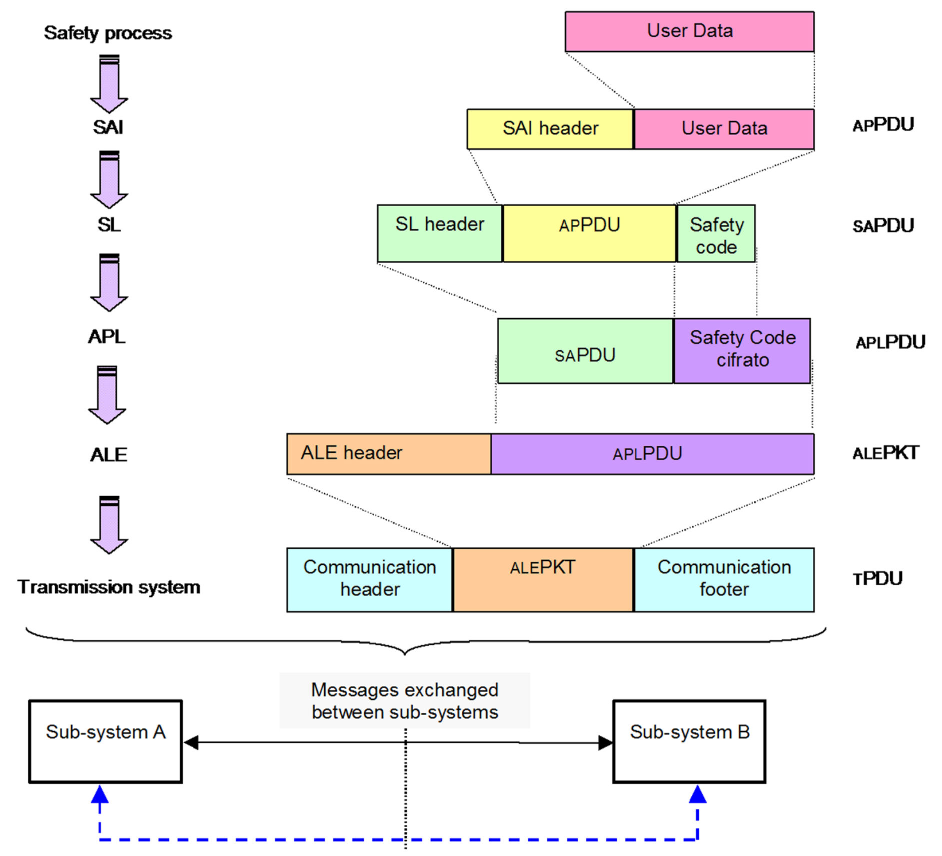


Fig. 3 Information flow through the protocol stack during messages send [18]

footer information. When receiving, each layer in reverse order decapsulates the message by removing either header/footer and then forwards the unwrapped data to the upper layer.

When the communication does not need cryptography, APL can be omitted. Moreover, in Fig. 2 ALE is depicted externally of the "safety related equipment" area meaning that this layer can be deployed on a remote (non-safe) device because it does not have safety requirements. For such reason, a means other than *shared memory* must be provided to realize the communication between ALE and the other layers of the protocol stack.

Connection establishment

In terms of handling of the connection, the most important role is played by the *safety layer* entity: when establishing a connection, the safety layer activates entity authentication, a handshake where two devices exchange (i) a nonce, (ii) the initial sequence number, (iii) execution cycle, and (iv) device identifier. In Fig. 4, the initial state of the safety layer is marked as IDLE: here, the safety layer is waiting for *connection establishment* requests. When a request arrives and the handshake is performed correctly, the safety layer goes through *WFAU3* and *WFRESP* in case of an incoming connection establishment request, through *WFTC*, *WFAR* whenever an outgoing connection request arrives. In both cases, a correct handshake leads the *safety layer* entity in the *DATA* state, where the two devices can finally exchange *safety-related* messages. The

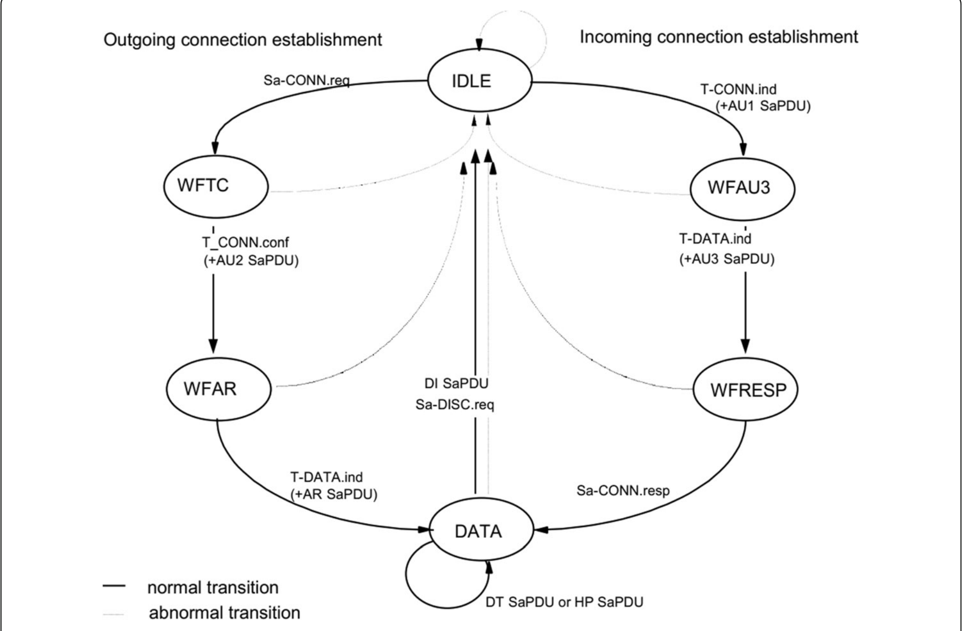


Fig. 4 States taken by the safety layer [12] during *connection establishment*

occurrence of one or more errors during the handshake or the exchange of data messages cause a transition to the *IDLE* state, which is therefore the *safe state* of the PVS.

Verification, validation, and safety plan

The first steps for the realization of a safe system according to the railway standards are the definition of a V&V and safety plan. The plan describes the steps to be fulfilled during the realization of the PVS to comply with applicable standards, to ultimately ensure a proper planning of activities which are adequate to reach the target SIL. Since the PVS is essentially software, we refer to the lifecycle of the CENELEC EN-50128 [8]. We explored such standard to define the documentation plan, which enlists the various documents to be produced at different stages of the design, implementation, validation, and deployment of a component or a system. Table 1 reports the documentation plan tailored to PVS, highlighting the document name and number, as well as tracing the document with respect to the involved EN50128 activities. It is worth remarking that our list of documents is a subset of all the lifecycle documents described in [8] (page 66, Table A.1), as our activity focuses only on the software requirement specification, architecture specification, design and implementation and verification and validation, while requirements definition, integration, deployment, and decommissioning are left out.

In compliance with [8] and according to the documentation plan shown in Table 1, the activities performed during the lifecycle's phases and reported in the V&V and safety plan are the following:

- Planning: description of the documentation plan, scheduling of the verification and validation activities and selection of the techniques or methodologies that must be adopted to obtain the target SIL during each of the development phases.

Table 1 Documentation plan for PVS

Document number	*Document name*	*CENELEC EN-50128 phase*
4	Software verification plan	*Planning*
5	Software validation plan	*Planning*
6	Software requirements specification	*Software requirements*
7	Overall software test specification	*Software requirements*
9	Software architecture specification	*Architecture and design*
10	Software design specification	*Architecture and design*
11	Software interface specification	*Architecture and design*
14	Software architecture and design verification report	*Architecture and design*
15	Software component design specification	*Component design*
16	Software component test specification	*Component design*
17	Software component design verification report	*Component design*
18	Software source code and supporting documentation	*Component implementation and testing*
19	Software component test report	*Component implementation and testing*
20	Software source code verification report	*Component implementation and testing*
24	Overall software test report	*Overall software testing/final validation*
25	Software validation report	*Overall software testing/final validation*
27	Release note	*Overall software testing/final validation*

- Software requirements: description of the software system to be developed considering both *functional* and *non-functional* requirements, development of the test plan to be followed during the verification phase.
- Architecture and design: definition of the software architecture to specify the software elements composing the system, the relations among them, their properties and their tasks. Definition of the software design to clarify the implementation activity to the development team with respect to what defined in the software architecture specification document, definition of the software interface for the interaction with the system and verification of the system architecture and design with respect to the verification validation and safety plan.
- Component design: description of the design of each architecture's component, specification of the tests for each component and verification of the component design.
- Component implementation and testing: implementation of the components, definition of the source code documentation, verification of the tests performed for each component, and of the source code produced.
- Overall software testing/final validation: reporting on the overall test performed over the software system and final validation of all the activities performed during the system development.

For each of the phases above, except for the planning phase which requires the drafting of the safety and V&V plan itself, in the safety and V&V plan we defined the techniques adopted to manage the safety and reach the target SIL, described the methodologies to be followed during the phase execution and scheduled the activities needed to release the final product. In the following section, we will report on the details about each of those project's phases, which, for simplicity, will be divided on "Architecture specification" section, "Design specification" section, and "Verification and validation activities" section.

Architecture specification

The standard [8] suggests methodologies and techniques to be applied for architecture specification, depending on the target SIL. PVS needs to comply with SIL4, therefore, according to Table A.1 of [8] we chose to adopt defensive programming, error detecting codes, fully defined interfaces, structured methodology, and modeling, which are summarized in Table 2. Software architecture was specified through the ISO/IEC 10746-1 Open Distributed Processing (*ODP-RM*) [17], a *structured methodology* which requires the use of five different *viewpoints*, each of them covering different aspects such as the definition of the general SW architecture, the data-flow between the SW modules, the specification of the Protocol Data Unit(s) (PDUs), and interfaces.

Modeling techniques allow describing ODP-RM through (i) *structure diagrams* for the description of the SW architecture, (ii) a *state-transition diagram* to describe the behavior of the protocol, and (iii) *sequence diagrams* to describe the sequence of actions taken during the communication between two devices. We will expand briefly each of the ODP-RM points of view in "Enterprise viewpoint" section to "Technology viewpoint" section. In this way, we will also debate on the items of Table 2 that are yet to be discussed: defensive programming, error detecting codes, and fully defined interfaces.

Enterprise viewpoint

Enterprise viewpoint specifies the role and responsibilities of (i) actors involved in the system and (ii) the external environment. It explains the relations between the system's component and the internal and external interfaces. We hereby expand the general PVS architecture depicted in Fig. 5, which builds on the following entities:

Table 2 Techniques [8] selected for the development of software architecture

Technique	***Description***	***Implementation***
Structured methodology	Use of precise and intuitive notation to promote the quality of software development by focusing attention on the early parts of the lifecycle	*ODP reference model* [17]
Modeling	Use of precise and field-specific graphical formalisms to provide a complete description of the system and its parts	*Sequence, state-transition, and structure diagrams*
Fully defined interfaces	Complete definition of both internal and external interfaces of each SW module	*Detailed description of interfaces for each SW module*
Defensive programming	Detect anomalous control flow, data flow, or data values	*Acceptance/credibility checks, control flow monitoring*
Error detecting codes	Detect errors in sensitive information by, e.g., Hamming, cyclic, or polynomial codes	*Cyclic redundancy code*

The three columns report respectively the technique name, a brief description of the technique, and the actual technique implementation

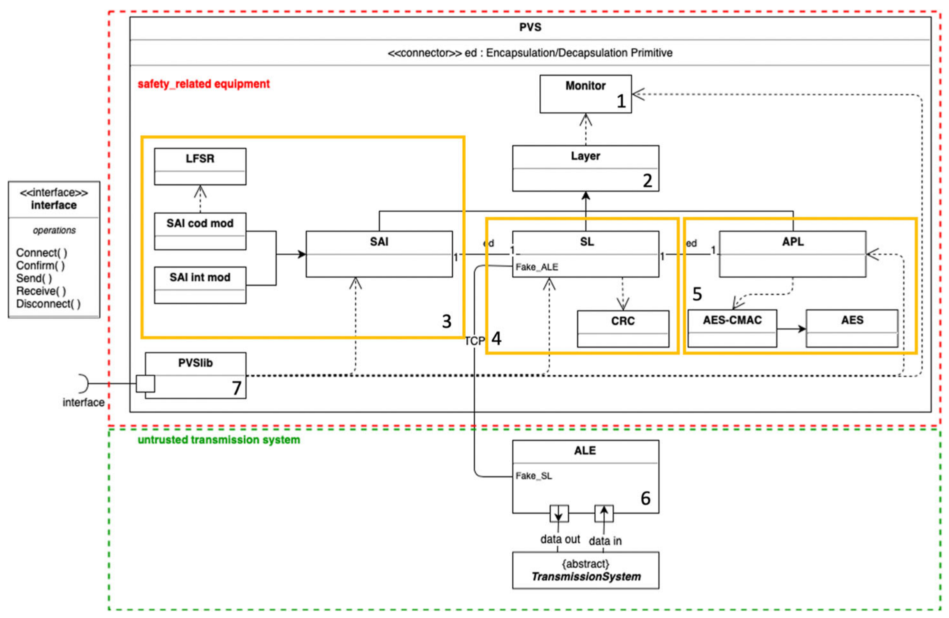


Fig. 5 Software Architecture of the PVS [18]. The numbers identify entities: (1) *monitor*; (2) *layer*; (3) *SAI*; (4) *SL*; (5) *APL*; (6) *ALE*; (7) *PVSlib*

1) *Monitor*: according to the Software Architecture/Design Specification (documents 9 and 10 in Table 1) this module checks the correct evolution of the control flow to provide *defensive programming* by implementing *control flow monitoring*, as it can be seen in Table 2;
2) *Layer*: the module that implements features shared amongst layers;
3) *Safe application intermediate layer:* the module that implements all the functionalities of SAI. *SAI int mod* is relevant for the closed network configuration, while open networks add the *SAI cod mod* and *LFSR* (linear shift register, produce pseudo-random numbers series) blocks to the picture;
4) *Safety layer:* implements all the functionalities of *SL*, using *CRC* module to perform *cyclic redundancy check* calculations (*error detecting codes: cyclic redundancy check*, from Table 2).
5) *Access protection layer:* implements all the functionalities of APL through the modules *AES-CMAC* [20] and *AES* which provide implementation of the homonymous algorithms.
6) *Adaptation layer management entity*: implement all the functionalities of ALE
7) *PVSlib:* provides *external interfaces* to simplify the interactions of the application layer with the protocol.

Figure 5 highlights connections between the layers SAI, SL, and APL with the *"ed"* lines, realized through shared memory. This notation points out how layers SAI, SL, and APL communicate each other using the interfaces identified as "encapsulation/decapsulation primitive." More precisely, each of the SAI, SL, and APL layers define (i) an *encapsulation* primitive, which will be invoked from the upper layer when sending messages and adds the layer-specific header information, and (ii) a *decapsulation* primitive, which lower layers invoke when receiving messages to verify the correctness of the layer-specific header dataframes. Layers *SL* and *ALE* are instead connected by a "*TCP*" line between their attributes *Fake_ALE* and *Fake_SL*. The dashed green rectangle shows how layer *ALE* could be placed on a remote machine: as a consequence, the communication between ALE and SL is realized through a TCP connection, set up during the initialization of the protocol stack and managed by entities *Fake_ALE* and *Fake_SL*.

Information viewpoint

Information viewpoint elaborates upon the structure and dataflow exchanged between components. To such extent, we specified format, structure, and size of the different PDUs to be exchanged between the PVS layers in the different protocol phases identified by the SL state in Fig. 4. When sending a message through PVS, the User Data crosses the layers SAI, SL, and APL using the encapsulation primitive. After APL encapsulation, the packet is sent by the SL::Fake_ALE and received by the ALE::Fake_SL via TCP through the connection established between the two entities during handshake. When ALE has added its header information, the message can be sent to transport OSI layer.

Computational viewpoint

The system can be seen as an ensemble of objects which exchange data through the interface functions and specification of actions taken by these objects during system's operative phase. This description includes—but not limits to—generation and verification of safety code, execution cycle and sequence numbers, and more generally sending/receiving algorithms for each layer of the protocol stack. The description of such sending/receiving algorithms causes layer interfaces to be *fully defined*.

Engineering viewpoint

The engineering viewpoint constitutes the most detailed perspective over the system, considering support functions intended to guarantee non-functional properties e.g., security or safety. These aspects are addressed through *acceptance/credibility checks* (*defensive programming*: *acceptance/credibility checks*, from Table 2), *control flow monitoring*, and *cycle redundancy check* (CRC).

Acceptance/credibility checks (left part of Fig. 6) have been implemented by checking inputs, intermediate values, and outputs of all the functions through conditional constructs inside each of the SW modules identified during architecture specification.

Control flow monitoring (right part of Fig. 6) includes the software entity *monitor*, which observes the execution flow with respect to some predefined paths. In a nutshell, the monitoring of the control flow works as follows. The monitor holds *primitives*, each corresponding to a specific *protocol phase* (connection establishment, sending and receiving messages, or disconnection) and points to the initial primitive of the current phase. When a layer calls a *primitive*, it notifies the execution of such primitive to the monitor through the *signature transfer* function. Then, the monitor verifies the correctness of the evolution of the control flow through the *signature checking* process, checking if the signature of the primitive transferred by the process conforms to expectations. If either of the checks above fail, they are *notified to the application*

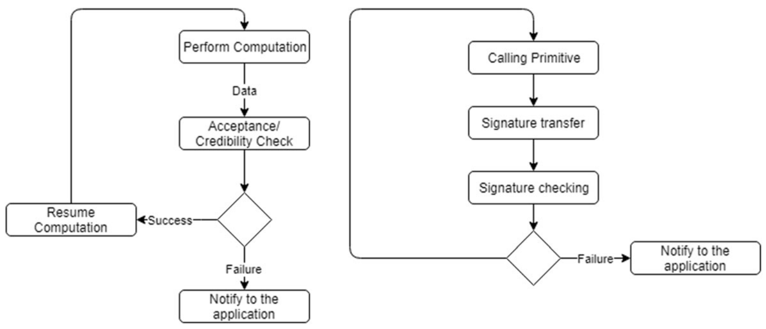


Fig. 6 Acceptance/Credibility Check and Control Flow Monitoring Diagrams

causing connection to close and letting the safety layer entity to return to its initial *IDLE* (*safe*) *state.*

The 64-bit CRC instead ensures integrity of the messages. The choice of CRC with respect to other *error detection codes* was carried out as CRC is considered proven-in-use for the railway domain [12, 14]. Moreover, as described in [21, 22], the usage of a 64-bit CRC with an adequate generator polynomial (e.g., Jones [23]) lowers the probability of collisions i.e., same CRC assigned to different dataframes. Polynomial selection is guided by the *Hamming distance* (HD) [23] that is defined as the number of bits two dataframes differ. Knowing the minimum HD for specifics dataframes lengths allow us to estimate the *probability of undetected errors* (Pub) using the following formula [22]:

$$P_{ud(\epsilon)} \approx \sum_{i=d_{min}}^{d_{max}} \frac{\binom{n}{k}}{2^r} * (1-\epsilon)^{n-1} (\epsilon)^i \tag{1}$$

where

- ϵ : *bit error rate of the physical mean* (e.g., optic fiber, Wi-Fi);
- n : *codeword size*;
- r : *code size*;
- $d_{\min}$: *minimum HD for selected polynomial and dataword size*

The formula (1) allows to estimate the probability of undetected errors for a CRC that uses the selected polynomial. In our case, considering as physical mean a plastic optic fiber with 10 GB/s speed and a supposed BER of 10^{-10} [24], a maximal dataframes size of 65,000 byte and a minimum HD of 6, we get

$$\epsilon = 10^{-10}, n = 65008, r = 8 \text{ and } d_{min} = 6 \text{ and we obtain } P_{ud(\epsilon)} \approx 10^{-31.575}.$$

Multiplying this value by the maximal supposed throughput of messages and comparing the result obtained against the target tolerable hazard rate per hour (THR), we obtain $P_{ud(\epsilon)} \approx 10^{-27.581} * h^{-1}$. Considering that SIL4 requires the failure rate to be less than 10^{-9}, the selected CRC is plenty sufficient under all expected operating conditions, also in case different channels are used and consequently different parameters values are set.

Technology viewpoint

Lastly, the technology viewpoint considers the physical characteristic and assumptions of use (AoU) of the target platform where PVS will be deployed. Our three AoU are mandatory requirements for all the operating systems on which the PVS software will execute: (i) target platform must support multithreading: this is required because our implementation of the protocol requires the execution of multiple thread needed for example to handle the execution cycle or to trigger event-specific handling actions; (ii) target platform must run a POSIX-compliant OS: PVS may execute on multiple platform, and consequently POSIX compliance is a natural choice to improve portability. For this reason, we implemented different layers of the Protocol Stack by using POSIX features like thread and time source management; (iii) target platform must provide TCP/IP networking support: since it is the de-facto standard for communications, we lay PVS on top of the TCP/IP stack.

Design specification

The concepts and the methodologies defined in the previous section have been detailed during *design specification* phase, which for each SW module specifies: the *scope* of the module, the *signature* of the functions, the *interfaces* and the *data structures*. The techniques selected from [8] for this phase are listed in Table 3.

As pointed out by Table 3, during design specification, we adopted a *modular approach* in order to implement the architecture's *components* depicted in Fig. 7. This technique requires to decompose the software in small parts easy to understand and analyze. Following such design technique, we were able to *fully define* each software *component* in terms of *interfaces* and related *parameters*. Each of these components has been specified using different *modeling* techniques, such as the *structure diagram* of Fig. 7, where all the software components and their relations are identified.

We selected and applied one of the widespread *coding standards* for safety-critical applications: *MISRA C 2012* [25]. Such coding standard specifies a number of guidelines categorized as *rules*, requirements that are precisely defined and can be enforced solely by analysis of the source code and *directives*, that instead may not be precisely

Table 3 Techniques selected for design specification [13]. The three columns report respectively the technique name, a brief description of the technique, and its implementation

Technique	***Description***	***Implementation***
Modular approach	Decomposition of a software into small comprehensible parts in order to limit the complexity of the software	*Definition of different SW modules depending on functionalities needed.*
Components	Well-defined interfaces and behaviour with respect to the software architecture and design	*Fully defined interface, parameter number limit*
Modelling	Use of precise and field-specific graphical formalisms to provide a complete description of the system and its parts.	*Control flow, state-transition and structure diagrams*
Design and coding standard	To ensure a uniform layout of the design documents and the produced code, enforce consistent programming and to enforce a standard design method which avoids errors.	*MISRA C 2012 coding standard* [25], *adequate coding styles*
Strongly typed programming language	Reduce the probability of faults by using a language which permits a high level of checking by the compiler	*Subset of the standard C99: MISRA C 2012 coding standard* [25]

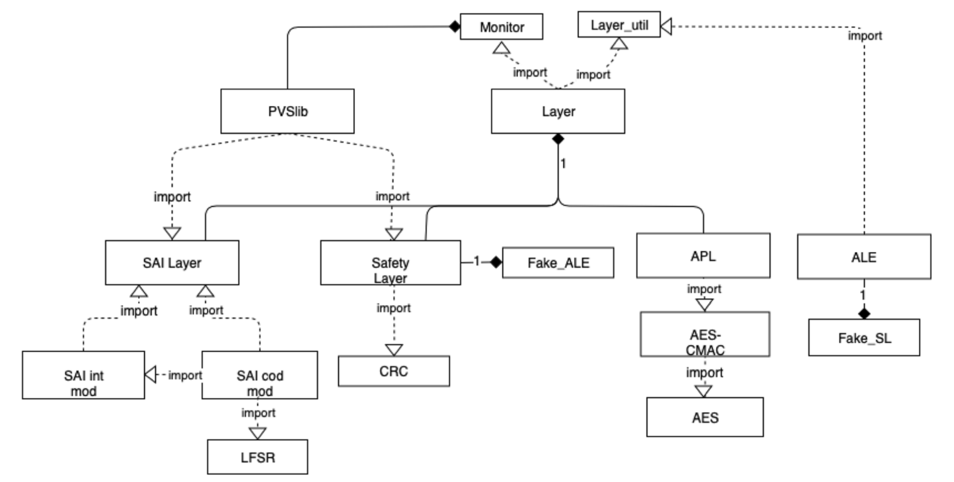


Fig. 7 Structure diagram representing the software components specified during design specification

defined and may require reference to documentation, functional requirements or subjective judgement to check the compliance of the source code.

The compliance of the source code with the MISRA coding standard helps the programmer to prevent undefined behaviors which can arise from an incorrect use of the programming language. Moreover, it ensures the production of a high-quality code which will be easier to understand, to debug and to test.

In general, *MISRA C 2012* requires the use of a subset of the standard C99 along with adequate coding styles [26] (e.g., avoiding libraries such as *"stdio.h"* to prevent the large number of unspecified, undefined and implementation-defined behaviors associated with streams and file I/O). This contributes to a *strongly typed programming language*, as MISRA C 2012 [25] requires, among other rules, to explicitly re-define types, e.g., 4, 8, 16-bit integers (Directive 4.6: *typedefs* that indicate size and signedness should be used in place of the basic numerical types) to be sure of the storage size needed by arithmetic variables on different architectures.

In some cases, the programmer could *intentionally* violate one or more rules or directive. For example, it could allow the value of a *composite expression* to be assigned to an object of wider *essential type* to avoid sub-optimal compiler code generation (violation of Rule 10.6) and obtaining a better performance. In such cases, a *deviation* must be provided, defining:

1. The *guideline(s)* being *violated*;
2. The circumstances in which the *violation* is acceptable;
3. The reason why the *deviation* is required;
4. Background information to explain the context and the language issues;
5. A set of requirements to include any risk assessment procedures and precautions which must be observed.

In addition to the *MISRA* coding standard, we selected code metrics and corresponding thresholds that should be checked while developing the C source code of PVS. We choose some of the most acknowledged code metrics in software engineering based on documents [27, 28] and on oral interviews with domain experts. We ended up selecting the metrics *lines of code* (LoC), *cyclomatic complexity* (CC), and *number of parameters* (NoP), while thresholds were extracted from both the experience of practitioners of RFI and on white papers available in the literature [28]. As a result, PVS functions should not exceed 100 uncommented lines (LoC), while the *cyclomatic complexity* (CC) of each single function should be lower or equal than 10. Finally, the *Number of parameters* (NoP) should not be greater than 5. Together, such metrics and the respect of their constraints favor a better software quality.

Verification and validation activities

To adhere with CENELEC standards, also verification and validation techniques and activities (shown in Table 4) have been selected and performed in compliance with [8]. It is worth remarking that the software/hardware integration phase is out of scope in our work as it will be privately managed by the owner of the case study. We detail instead each one of techniques applied in Table 4 and in the following subsections that expand on test coverage and static analysis.

Dynamic analysis, testing, and traceability

EN50128 [8] states that activities to be applied for dynamic analysis and testing are: (i) *test cases boundary value analysis (*to detect the boundary values of the inputs of functions), and (ii) *equivalence classes and input partition testing*, which selects the correct inputs depending on correlations between input and outputs of each function. Those activities allow deriving *unit tests* executed as *white-box* test, i.e., assuming knowledge of the internal structure of functions, and *functional tests* to be instead executed as *black-box* tests, i.e., without assuming any knowledge of the internals of functions.

Another white-box testing activity has been the *structure-based testing*, a testing technique that aims at exercising the largest number of sub-routines in the *call graph* through an accurate selection of input values. The overall *test suite* is composed 63 test cases, partitioned as 19 functional test cases, 37 unit test cases and 7 structure-based test cases. Each of these tests include also robustness tests defined by calling functions with unexpected parameters, such as *NULL* pointers, or numerical overflows like

Table 4 Techniques selected for generic V&V Activities [8]

Technique	*Description*	*Implementation*
Dynamic analysis and testing	Verification of the software through execution and instrumentation of the software elements	*Test cases boundary value analysis, unit testing, performance modelling, equivalence classes and input partition testing, structure-based testing*
Functional/ black-box testing	To verify that the functional requirements are satisfied	*Functional tests, boundary value analysis, equivalence classes and input partition testing*
Traceability	Ensuring that all requirements are properly met and that no untraceable material has been introduced.	*Traceability matrix*
Test coverage	Verification of code coverage reached using different criteria	*Statement and compound condition*
Static analysis	Verification of the software through manual/ automated analysis of the Software structure	*Boundary value analysis, control flow analysis and walkthroughs/design reviews*

The three columns report respectively the technique name, a brief description of the technique, and the actual technique implementation passing 16-bit integers as *actual parameters* (used during the function call) where the *formal parameters* (defined in the function signature) were 8-bit integers (see the example test case in Table 5).

For each test case, the approach is the following:

1. The test case is prepared. In addition to the general data such as the name of the test, requirement and date, the necessary steps to perform the test are identified. The functions called and the relationships between data and expected results are specified.
2. The data sets to be provided as input to the test are identified.
3. The test script is written following the test case.

Table 5 An example test case showing the unit test of the CRC generation function

Test case ID	crc_ch1		
Priority	Medium		
Description	CRC generation function		
Module	Safety layer		
Prepared by		**Date prepared**	
Reviewed/ updated		**Date reviewed**	
Tested by		**Date tested**	
Software version			
Test activities			
Sl. no.	**Step description**	**Expected results**	
1	Definition of the input data		
2	if the input data are plausibles: • Computation of the CRC using an external tool and comparation with the result obtained using the internal function otherwise: • Verification of the correct error handling and report	The code obtained using the external tool and the internal function must be equals	
Test data sets			
Data type	**Data set 1**	**Data set 2**	**Data set 3**
uint8_t input[];	input ={0,0,0,0,0,0,0,0}	input = {0xffff, 0xffff, 0xffff, 0xffff}	input = {‘C’,’o’, ’n’,’t’,’r’,’o’,’l’ }
uint16_t size;	size = 8	size = 4	size = 0
Actual results			
Data type	**Data set 4**	**Data set 5**	**Data set 6**
uint8_t input[];	input = NULL	input = {0,0,0,0,0,0, 0,0}	
uint16_t size;	size=0	size = − 1	
Actual results			
Test case result			

Data sets 2 and 4 are robustness tests that invoke the function with values bigger than expected (using 16-bit unsigned integers in place of the 8-bit requested) and with NULL parameters

4. The test is executed. For each data set, the test result is marked as passed or failed based on the correspondence between expected and actual results. When all data sets produce the expected output, the test case is considered passed.

Some tests require a slightly complex procedure: both layers SAI and SL need, to comply with their functional requirements, a *pseudo-random number generator*. Random number generation has been unit tested as follows: (i) generation of a *statistic sample* composed by a relevant quantity of random numbers; (ii) use of the *ent* [29] library to process the *statistic sample*, and (iii) evaluation of the *statistic measures* produced by *ent*, namely *entropy*, *chi-square mean*, *arithmetic mean*, *monte carlo value for* P_i, and *serial correlation coefficient*. The analysis of the random number generation did not reveal any weakness on the target platform, but it is recommended to repeat this test when installing PVS on different platforms.

Another important aspect of dynamic analysis relative to functional testing is given by *traceability* [8] that is usually satisfied by filling a *traceability matrix*, where each *functional requirement* reported in [18] is linked to "called functions." The traceability matrix allows the identification of test cases which needs to be updated in case of change in requirements and to track the overall test execution status.

Finally, the evaluation of test quality should be performed through *code coverage*, considering two different coverage criteria. The two criteria applied in this project are described below.

- *Statement coverage*: computes the fraction of the executed lines of code with respect to their total. It is a relatively weak criterion since it does not account for logical operators and control structures.
- *Compound condition coverage*: requires that each single condition in a decision should be tested by assigning both the *false* and the *true* values

Test coverage

Test coverage reports on the amount of source code that was tested. Coverage depends on specific criteria that can be set depending on the needs of the user and on the characteristic of the system or software. Often, for each selected criteria a certain coverage threshold is set beyond which the test campaign can be considered concluded, that is, the test stop condition. However, the stop condition for our testing activities does not rely on coverage values. Instead, tests should be performed until the exhaustion of the inputs identified through the boundary value analysis and the equivalence classes and input partition testing performed in both dynamic and static analysis. Despite this, it is very useful to analyse coverage, because it gives an indication about how many behaviors have been examined.

Figure 8 reports on test coverage according to the selected criteria, detailing the results for each of the modules depicted in Fig. 5 (apart from the additional sub-module *layer_util*, added to better modularize the entity *layer* during the design specification phase).

Coverage is calculated through the tools *Gcovr* [30] and *Lcov* [31], which profile the code and show code coverage using different criteria, ultimately providing human-readable coverage reports in HTML format.

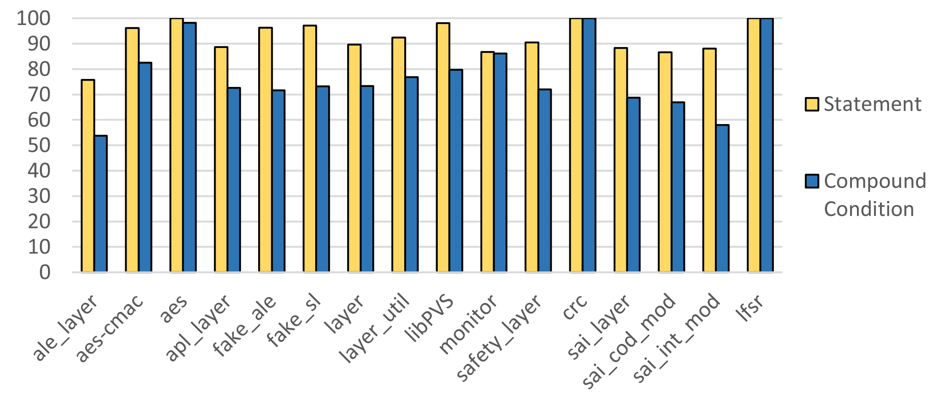


Fig. 8 Statement and compound condition coverage computed with *Gcovr*

Noticeably, single coverage values in Fig. 8 rarely reach 100%, with an overall of 89% for statement coverage and 69.8% for compound condition coverage. Such values are determined by the high presence of defensive code: code branches produced by conditional statements needed to check inputs and outputs for acceptance and credibility (acceptance/credibility check) or to verify the evolution of the control flow (control flow monitoring), eventually triggering error handling routines. Given the complexity of designing test capable of cover such branches, we decided to apply manual inspection to verify all the code that was not covered by dynamic analysis.

Static analysis

Static analysis requires the semi-automated analysis of the source code using software tools or manual inspections without executing the code. Compliance with SIL4 requires [8]: (i) *control flow analysis*, to check the evolution of the control flow, (ii) *walkthrough/design reviews*, to determine useless or buggy code through manual inspections; (iii) verification of satisfaction of coding rules and metrics, done through the use of the static analysis tool Polyspace [32]. We will report about the *control flow analysis* and *walkthrough* performed in the "Control flow analysis and walkthrough" section, reporting also about the result of the evaluation of code metrics and coding rules in "Code metrics" and "Coding rules" sections.

Control flow analysis and walkthrough

Since the *monitor* object already implements the dynamic control flow monitoring of the whole PVS, we directly step into *walkthrough/design reviews* (as verification of the control flow is actually a review of the behavior of the Monitor). We asked a colleague, with no knowledge on the project, to act as auditor. He reviewed the flow of execution of the PVS when used in a typical procedure consisting of configuration, connection, sending, receiving and disconnection. This allowed reviewing all the functions implemented (tracked with a checklist). The auditor filled the "Comment" column of Table 6 which address MISRA *Directives* and Table 7, which points to good practices of

Table 6 Extract of the walkthrough report on MISRA directive. Table's columns contain in order: directive number, directive class, description of the rule and auditor's comment about the rule

Directive	Class	Description	Comment
1.1	Mandatory	Any implementation-defined behavior on which the output of the program depends shall be documented and understood	The description of the implementations Followed a pattern that made it easier to understand its content and goals
2.1	Mandatory	All source files shall compile without any compilation errors	No errors or warnings during compilation.
3.1	Mandatory	All code shall be traceable to documented requirements	Most of the requirements names are reused in the code; it is easy to trace everything to its original requirements.
4.4	Advisory	Sections of code should not be "commented out"	No commented code found.
4.5	Advisory	Identifiers in the same name space with overlapping visibility should be typographically unambiguous	Identifiers are self-explanatories, no ambiguity found.
4.6	Advisory	Typedefs that indicate size and signedness should be used in place of the basic numerical types	Functions use raw integers to indicate a state/type.
4.7	Mandatory	If a function returns error information, then that error information shall be tested	There is error handling in all the functions related to the protocol, haven't seen any tests to prove their efficiency.
4.8	Advisory	If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden	Pointers have their object implementation hidden.
4.9	Advisory	A function should be used in preference to a function-like macro where they are interchangeable	Function-like macros are rare in the code
4.12	Mandatory	Dynamic memory allocation shall not be used	Not found.

Table 7 Extract of the walkthrough report on selected coding/design rules

Rule	Comment
A module/component shall have a single well-defined task or function to fulfill;	*They all are straightforward.*
Connections between modules/components shall be limited and strictly defined	*Ok.*
Collections of subprograms shall be built providing several levels of modules/components;	*Done.*
subprograms shall have a single entry and a single exit only;	*Done.*
Modules/components shall communicate with other modules/ components via their interfaces. Where global or common variables are used, they shall be well structured, access shall be controlled and their use shall be justified in each instance;	*Most of them share structured variables, and all the interfaces are used accordingly.*
All module/component interfaces shall be fully documented	*They are well described and documented*

First column contains the rule description while the second contains auditor's comment software design/development [25], to eventually report violations of the described rules. Tables 6 and 7 only report rules and practices that were deemed applicable for the walkthrough. As can be seen from the "Comment" columns of Tables 6 and 7, the analysis did not find any violation.

Code metrics

The values of code metrics selected for the implementation of PVS, extracted by using *Polyspace* [32], are reported in Table 8. The table reports metric values highlighting in yellow the values that violate the constraints set by the V&V&S plan described in "Design specification" section.

As can be seen from the table above, some of the source files exceed the predefined constraints. The file *PVSlib.c* violates cyclomatic complexity threshold, while constraints on lines of code are violated in files *monitor.c* and *aes.c*. For number of parameters, we have violations in the files *PVSlib.c*, *sai_int_mod.c*, *sai_cod_mod.c*, *sai_layer.c*, and *safety_layer.c*. Those violations have been addressed during the finals V&V stages of the project to check whether was possible or necessary to operate some changes on the modules design or implementation. Briefly, the considerations done on each of those violations are:

- Cyclomatic complexity: the values could be reduced defining sub-procedures that execute some of the instructions executed by the functions under exam. However, it was not considered necessary to make changes as the two functions were appropriately overhauled and tested; furthermore, their modification would not have brought significant improvements in terms of simplicity or maintainability of the code.
- Number of parameters: we did not make any change because during the design specification phase, it was deemed appropriate to keep them unchanged for reasons of logical consistency between the inputs and outputs produced by them.
- Lines of code: it has been observed that the tool [32] calculates the metric as the difference between the brace that determines the start of the function and the brace

Table 8 Maximum value per file of the metrics extracted by *Polyspace*

File	CC	LoC	NoP
lfsr.c	5	21	2
layer.c	2	13	5
crc.c	5	27	2
aes-cmac.c	10	49	5
layer_util.c	6	31	5
monitor.c	7	107	3
apl_layer.c	9	44	4
PVSlib.c	11	47	8
fake_ale.c	9	80	4
sai_int_mod.c	8	58	7
aes.c	10	163	4
sai_cod_mod.c	10	70	9
sai_layer.c	10	63	7
safety_layer.c	10	92	7

First column contains the file name, the others report respectively on cyclomatic complexity (CC), lines of code (LoC), and number of parameters (NoP). Violations with respects to defined thresholds are in yellow that determines its end, also counting the empty lines necessary to improve readability of the code. Therefore, the actual metric value for the functions of the monitor.c file was less than the threshold set. The same does not apply to the aes.c file. However, we did not modify its functions to avoid the insertion of bugs and cause malfunctions of the algorithm.

Coding rules

Polyspace [32] finds violations to MISRA C:2012 coding rules, showing some *false positive*, that need to be manually dropped, alongside with violations that we do not consider as *defects*. The final result of running Polyspace on the PVS source code is in Table 9.

After attentive analysis of the violations identified and, on the opportunity to solve them, we opted for motivating *deviations*. Each deviation describes: (i) the rule; (ii) the use case that triggers the violation; (iii) the root cause for rule violation; (iv) the possible risks given by rule violation; (v) any possible alternative approach, and (vi) a verification code to be used to check risks on different architectures. While aspects (ii) and (v) cannot be shared due to confidentiality, Tables 10 and 11 show deviations respectively for violations of rules 11.3 and 18.8. We do not report about each of the violations identified in Table 9 as deviations are in any case consistent with what shown in Tables 10 and 11.

Performance analysis of the protocol

We conducted a performance analysis targeting throughput in terms of Kilobytes sent and received per second between two remote machine and using three different setups: (i) a normal *C99 socket* transmission, (ii) PVS closed network, and (iii) PVS open network.

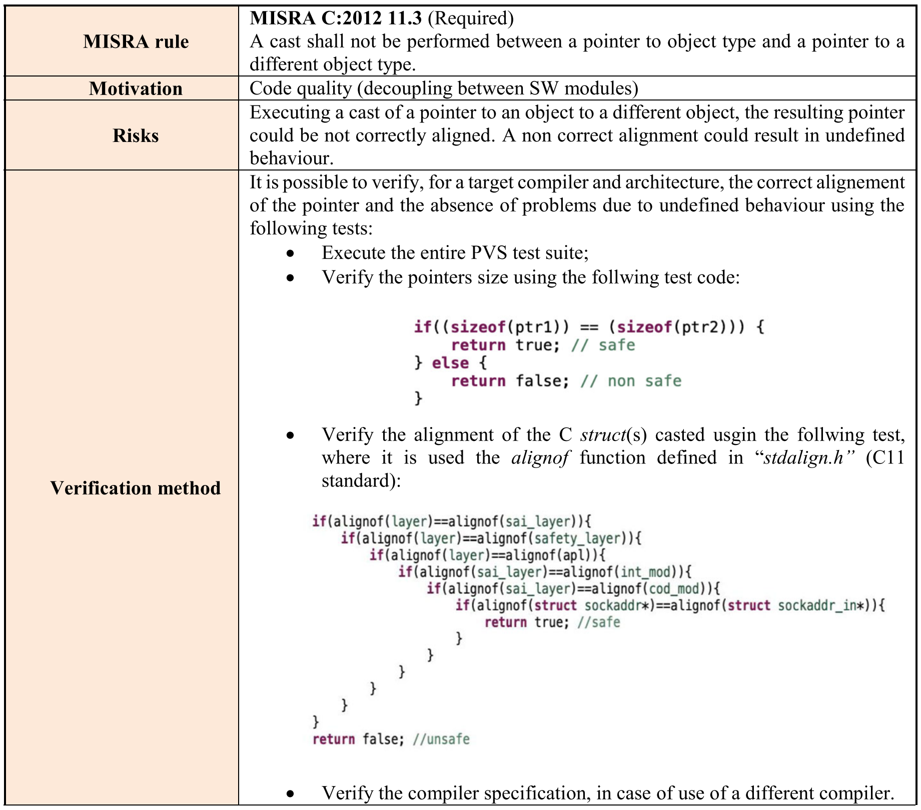
Experiments execution

We created *bash scripts* that exercise PVS by executing the sending and the receiving of 100 packets of different payload sizes, varying from 5 kB to 65 kB, that is the maximum payload allowed by the PVS. The underlying *WLAN* has a nominal bandwidth of 100 Mbps. Figures 9 and 10 have been generated by averaging the data collected during

Table 9 MISRA C:2012 violations purged by false positive and false defect. The table reports on rule description and Number of violations inside the code

MISRA C:2012 rule	N° of violations
MISRA C:2012 11.3 (Required) *A cast shall not be performed between a pointer to object type and a pointer to a different object type.*	**88**
MISRA C:2012 11.5 (Advisory) *A conversion should not be performed from pointer to void into pointer to object.*	**24**
MISRA C:2012 11.6 (Required) *A cast shall not be performed between pointer to void and an arithmetic type.*	**4**
MISRA C:2012 15.4 (Advisory)*There should be no more than one break or go-to statement used to terminate any iteration statement.*	**2**
MISRA C:2012 18.8 (Required) *Variable-length array types shall not be used.*	**33**
Tot	**151**

Table 10 Deviation for MISRA rule 11.3



MISRA rule	**MISRA C:2012 11.3** (Required) A cast shall not be performed between a pointer to object type and a pointer to a different object type.
Motivation	Code quality (decoupling between SW modules)
Risks	Executing a cast of a pointer to an object to a different object, the resulting pointer could be not correctly aligned. A non correct alignment could result in undefined behaviour.
Verification method	It is possible to verify, for a target compiler and architecture, the correct alignement of the pointer and the absence of problems due to undefined behaviour using the following tests:

- Execute the entire PVS test suite;
- Verify the pointers size using the follwing test code:

```
if((sizeof(ptr1)) == (sizeof(ptr2))) {
    return true; // safe
} else {
    return false; // non safe
}
```

- Verify the alignment of the C *struct*(s) casted usgin the follwing test, where it is used the *alignof* function defined in "*stdalign.h*" (C11 standard):

```
if(alignof(layer)==alignof(sai_layer)){
    if(alignof(layer)==alignof(safety_layer)){
        if(alignof(layer)==alignof(apl)){
            if(alignof(sai_layer)==alignof(int_mod)){
                if(alignof(sai_layer)==alignof(cod_mod)){
                    if(alignof(struct sockaddr*)==alignof(struct sockaddr_in*)){
                        return true; //safe
                    }
                }
            }
        }
    }
}
return false; //unsafe
```

- Verify the compiler specification, in case of use of a different compiler.

the execution of the 100 experimental runs for each of the 2 operational modes of PVS and for a basic TCP socket. As expected, the open networks mode of PVS (Category 3 of [7]) has the lowest throughput. This can be explained considering that this operational mode of PVS embeds AES encryption and decryption, in addition to building or checking a wider packet header. However, the throughput of approximately 300 kB/s is appropriate considering the usual dimension of data exchanged between nodes in the railway network.

Table 11 Deviation for MISRA rule 18.8

MISRA rule	**MISRA C:2012 18.8 (required)** **Variable-length array types shall not be used.**
Motivation	Minimization of stack memory, used for the messages handling and the ease in the definition of array sizes used to store the messages.
Risks	Variable-length array are implemented as variable size objects stored on the stack. Using variable-length array it is very difficult to determine the stack memory required. Moreover, if the size variable was negative or zero undefined behavior can occur. If a variable length array must be compatible with another array type, the array sizes must be identical; otherwise, an undefined behavior could occur.
Verification method	The maximum stack memory used by the PVS during exchange of messages is calculable as payload + header and therefore it is sufficient to make sure that the system is able to manage this memory size.

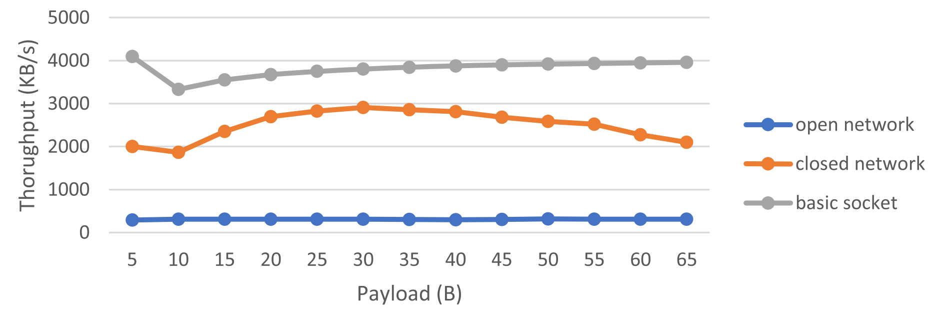


Fig. 9 Receive throughput by using different operational modes of PVS and a simple TCP socket

Concluding remarks

In this paper we reported our experience in the implementation of software for safety-critical systems. More in detail, we applied the CENELEC railway standards to deploy the Protocollo Vitale Standard, a safe and secure communication protocol to interlace various railway devices. We described the Protocollo Vitale Standard [18], examining its structure, and then describing and motivating the techniques used for the design, implementation and V&V processes.

The overall process complies with applicable *CENELEC* standards [6–8], which specify the software lifecycle and the V&V activities to be carried out before certification and decommissioning. While standards EN50126 [6] and EN50159 [7] define the process needed to specify the safety functions to be allocated to the Software, standard EN50128 [8] focuses on technical requirements for the development of software, that needs to adhere with a given safety integrity level (SIL).

For the development of our safety-critical communication protocol, we started from the structured methodology *ODP-RM* [17] to define and describe the system architecture, where we applied various modeling formalism like structure diagrams, sequence diagrams, block diagrams, and state machines to describe the system and selected techniques needed to improve the system safety such error detecting codes (CRC), control flow monitoring, and acceptance/credibility checks. In compliance with [8] and the verification, validation and safety Plan, we implemented our software according to (i) the *MISRA C 2012* coding standard [25] and (ii) a set of *software metrics* to be respected [27, 28].

The V&V activities consisted then in a thorough revision of each activity done in the previous phases, in the drafting and execution of *unit*, *functional*, and *structure-based*

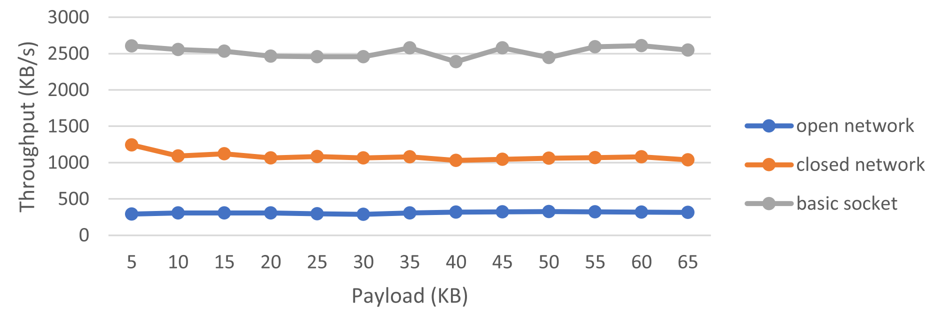


Fig. 10 Send throughput by using different operational modes of PVS and a simple TCP socket

tests, and in the verification of compliance to coding standards and software metrics. Ultimately, we conducted *performance analyses* to verify the sending/receiving throughput of packets with both *open/closed network* operational modes of PVS.

As a last remark, we strongly believe that our experience in preparing, implementing, verifying, and validating a software to be installed in safety-critical systems could be used as reference for those who have to deal with safety requirements either for railways or for other domains (e.g., automotive, avionics), which share most of the V&V items with respect to railway standards.

Acknowledgements
Not applicable.

Authors' contributions
Entity of contribution and of involvement in technical research matches authors' order. The author(s) read and approved the final manuscript.

Funding
This work has been funded by RFI Rete Ferroviaria Italiana S.p.A.

Availability of data and materials
Data is not available due to contractual IPRs.

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Department of Mathematics and Informatics, University of Florence, Viale Morgagni 65, 50134 Florence, Italy.
[2]Research and Development, Rete Ferroviaria Italiana, Via Curzio Malaparte 8, 50145 Florence, Italy.

Received: 17 June 2020 Accepted: 24 January 2021
Published online: 02 March 2021

References
1. Avizienis A et al (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Dependable Secure Comput 1(1):11–33
2. Ceccarelli A, Zoppi T, Vasenev A, Mori M, Ionita D, Montoya L, & Bondavalli A (2018) Threat analysis in systems-of-systems: an emergence-oriented approach. ACM Transactions on Cyber-Physical Systems 3(2):1–24. USA.
3. Bhatti ZE, Roop PS, Sinha R (2016) Unified functional safety assessment of industrial automation systems. IEEE Trans Industr Inform 13(1):17–26
4. Xie G et al (2017) Hardware cost design optimization for functional safety-critical parallel applications on heterogeneous distributed embedded systems. IEEE Trans Industr Inform 14.6:2418–2431
5. IEC, IEC61508 (2010) 61508 functional safety of electrical/electronic/programmable electronic safety-related systems. International Electrotechnical Commission
6. CEI EN 50126. Railway applications - the specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS), 2008.
7. CENELEC, EN50159 (2010) Railway applications-communication, signalling and processing systems - safety-related communication in transmission systems [Report]: Standard.-[sl]. European Committee for Electro-Technical Standardization
8. CENELEC, EN50128. Railway applications-communication, signaling and processing systems-software for railway control and protection systems, 2011
9. CEI EN 50129 (2004) Railway applications - communication, signalling and processing systems - Safety related electronic systems for signalling
10. Sniady A, Soler J (2012) An overview of GSM-R technology and its shortcomings. In: 2012 12th International Conference on ITS Telecommunications. USA: IEEE; pp 626–629
11. Fall KR (2011) TCP/IP Illustrated, Volume 1: The Protocols, Addison-Wesley Professional, USA, ISBN: 9780321336316
12. UNISIG (2015) SUBSET-037 EuroRadio FIS. Version 3.2.0
13. Winter P et al (2009) Compendium on ERTMS. Eurail Press
14. SUBSET, UNISIG. 098, RBC-RBC safe communication interface, 2007.
15. Bertieri D, Zoppi T, Mungiello I, Ceccarelli A, Barbareschi M, Bondavalli A (2019) Implementation, verification and validation of a safe and secure communication protocol for the railway domain. In: LADC 2019 9th American symposium on dependable computing
16. Wang R, Zhao H-b, Wang S-m (2008) Research on uplink-signal simulator used for BTM test in balise system. J China Railway Soc 30(6):46–50
17. Technical Committee ISO/IEC JTC 1/SC 7 (1998) ISO/IEC 10746-1:2009 Information technology — Open distributed processing — Reference model: Overview — Part 1
18. RFI (2017) Relazione Tecnica RFI DTCDNSSS RT IS 05 021 F. Protocollo Vitale Standard. Internal report (in Italian). Italy
19. PUB, NIST FIPS (2001) 197: advanced encryption standard (AES). Fed Inf Process Stand Publ 197(441):0311

20. Song, Junhyuk, et al. The aes-cmac algorithm. RFC 4493, 2006.
21. Wolf J, Michelson A, Levesque A (1982) On the probability of undetected error for linear block codes. IEEE Trans Commun 30(2):317–325
22. Franekova M, Rastocny K (2011) Modelling of disturbing effects within communication channel for safety-related communication system. Adv Electr Electron Eng 6(2):63–68
23. Koopman P, Chakravarty T (2004) Cyclic redundancy code (CRC) polynomial selection for embedded networks. In: International conference on dependable systems and networks, 2004. IEEE, pp 145–154
24. Loquai S et al (2012) 10-Gb/s pulse-amplitude modulated transmission over 1-mm large-core polymer optical fiber. IEEE Photon Technol Lett 24(10):851–853
25. MISRA C:2012s (2013) Guidelines for the use of the C language in critical systems. MIRA Limited, Warwickshire
26. Coding Styles. gnu.org/prep/standards/standards.html#Writing-C
27. Rosenberg LH, Hyatt LE (1997) Software quality metrics for object-oriented environments. Crosstalk J 10(4):1–6
28. Exida Consulting LLC. C/C++ coding standard recommendations for IEC 61508, version V1, revision R2 (2011)
29. John Walker, Ent - a pseudorandom number sequence test program. http://fourmilab.ch/random/, 2008.
30. GCovr Library. https://gcovr.com/en/stable/
31. LCov Library. https://github.com/linux-test-project/lcov
32. Polyspace tool, Matlab. https://it.mathworks.com/products/polyspace.html

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Submit your manuscript to a SpringerOpen journal and benefit from:

► Convenient online submission
► Rigorous peer review
► Open access: articles freely available online
► High visibility within the field
► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com