

A contribution to safety analysis of railway CBTC systems using Scola

Melissa Issad, Leila Koul, Antoine Rauzy

▶ **To cite this version:**

Melissa Issad, Leila Koul, Antoine Rauzy. A contribution to safety analysis of railway CBTC systems using Scola. Safety and Reliability of Complex Engineered Systems: ESREL 2015, Sep 2015, Zurich, Switzerland. 10.1201/b19094-64 . hal-01246161

HAL Id: hal-01246161
https://hal.science/hal-01246161

Submitted on 18 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright

A contribution to safety analysis of railway CBTC systems using Scola

M. Issad
LGI, Ecole Centrale Paris, France
Siemens SAS, France

L. Koul
PRiSM Laboratory
University of Versailles, France

A. Rauzy
Chaire Bleriot Fabre
LGI, Ecole Centrale Paris, France

ABSTRACT: Regarding their complexity, industrial systems are hard to design and even harder to validate and maintain. We try to address some particular issues of the railway systems conception.
Railway systems are characterized by their identified and limited number of failure accidents. Thus, safety analyses is mainly based on the research of failure scenarios that lead to these accidents. Those scenarios represent the misbehavior that must be avoided or corrected in the system. But, the specifications ambiguity makes it difficult to obtain a consistency and completeness in the analysis. At this point, the main issue is the systematic errors. They consist on the gaps on the system description that not only affect the comprehension but also the completeness of the analysis.
In this article, we propose an approach for system formalization and safety analyses. We define *ScOLA*, a modeling language built to understand and to formalize the specifications based on core concepts. We explain how, using a formal description and a stepwise simulation of the system, safety analysis can be performed easier and faster. The approach is applied to the Trainguard Mass Transit (the CBTC product of Siemens) CBTC(Communication Based Train Control) system of Siemens

Keywords : *Scola, Systems engineering, formal specification, safety analysis, CBTC.*

1 INTRODUCTION

Safety analysis in railway systems differs from the other industries. In that new systems are particular configurations of already existing ones. Therefore, to conduct a safety analysis, experts reuse existing ones. However, traditional methods do not resist to the growing complexity of systems. New features and functions have to be added. It is very difficult to develop, validate and maintain them. One of the reasons is the amount of implicit knowledge in the documents produced for existing systems. Exhaustiveness in safety and systems analysis became very difficult to prove.

Modeling is the most formal way for engineers to communicate, but accuracy is required. Multiple modeling languages, whether they are high or low level appeared and enhanced the way systems are done. Formal models have been designed and used to, in one hand, specify, design and develop systems such as the B-Method based on *B*(Abrial 2005), *Scade* (Abdulla et al. 2006) and more recently *AADL*(Feiler et al. 2006). Formal methods are also used for system safety and validation purposes such as *NuSMV*(Cimatti et al. 1999), *HipHops*(Papadopoulos et al. 2011), *Simulink* (Ong 1998) and *Altarica*(Arnold et al. 1999). Those methods are used in the industry, and are very powerful to specify and validate complex systems in an exhaustive way.

Still, the entry cost is high. In order to use formal models, it is required to have an accurate understanding of the system and an expertise in formal methods. In another hand, the so-called semi-formal methods are mainly graphical notations that represent the system at each stage in the development

life-cycle. An example of the semi-formal methods is the *SysML*(Friedenthal et al. 2011) language, defined by the OMG(Object Management Group). It is used to enlighten some ambiguities in systems and have a graphical view of the system. However, in order for an analysis to be accurate, it has to be exhaustive. Semi-formal methods are not designed for this purpose.

The issue today is how to build a bridge between the people designing the system and the people validating it. Making sure that a system where safety techniques are applied is correct, exhaustive and formal.

Brainstorming with the different system stakeholders, we noticed that the biggest issue is communication. The main reason is the lack of clarity in systems description, and the heterogeneity in the methods and tools used in the V-cycle. We built the ScOLA modeling language aiming at having a framework where system engineers can define and validate a system formally, starting from informal and textual descriptions of the system.

In this article, we present our on-going work on the formalization of railway systems: how can a formal description of system specifications allow a more efficient safety analysis of systems. Section 2 presents the process of system and safety engineering in the railway systems, section 3 introduces *ScOLA* and its core concepts, section 4 present the different techniques of safety analysis based on *ScOLA* models. Finally, section 5 is a concrete railway systems case study and the application of *ScOLA* and the different safety techniques identified on the Siemens CBTC product TGMT (Trainguard Mass Transit).

2 SYSTEMS ENGINEERING AND SAFETY ANALYSIS PROCESSES IN CBTC SYSTEMS

According to the IEEE 1474 standard(IEEE 2004), a CBTC system is a *continuous, automatic train control system using high-resolution train location determination, independent of track circuits; continuous, high-capacity, bidirectional train-to-wayside data communications; and trainborne and wayside processors capable of implementing Automatic Train Protection (ATP) functions, as well as optional Automatic Train Operation (ATO) and Automatic Train Supervision (ATS) functions.*(Pascoe and Eichorn 2009).

CBTC equipped trains determine independently their localization and forwards it to the track equipments.

The Trainguard© Mass Transit CBTC system is a *Siemens* solution for railway automation. It represents the operating system of a train. It is composed of two subsystems, the on-board and the wayside.

The on-board subsystem controls the train doors, the braking, the train position, its speed and the stop with the information to the passengers. The wayside mainly determines the trains movement authority according to their speed and position.

2.1 *Systems engineering process*

In order to define a modeling language that fits a system and its requirements, we need to understand how systems should be designed in theory, and what are the gaps with the existing methods.

The systems engineering (SE) process is the first phase of the system development in complex systems. According to (Leonard 1999), it integrates the inputs (customers requirements, requirements from prior developments, standards requirements), analyzes the requirements (functional, performance and design), realizes the functional and structural analysis (functions and components refinements) and finally a synthesis with the functions allocations.

On another hand, in the Cenelec standard EN50126(EN50126 1999), the system engineering process is simplified. It embeds a concept phase and a system definition and application conditions phase, in parallel with the system validation and the production of system requirements. The concept includes the definition of which components are going to be developed and why (given the customers' needs and the requirements collection). Followed by the definition in parallel with the validation of requirements. Figure 1 represents the v-cycle according to standard en50126. The system definition is basically a drawing defining the internal and external elements and interfaces of the system. The standard does not inform on a clear process of structuring the system, and what to consider as a part of the system or not. It results lengthy documents defining a general architecture of the system with its main components. Then, the system already known functions are described in detail; they are separately described with a refinement into sub-functions and requirements allocated to functions. These definitions involve a large collection of implicit concepts, such as the components involved in functions, the inputs and outputs of each function and the requirements allocated to each part of the system.

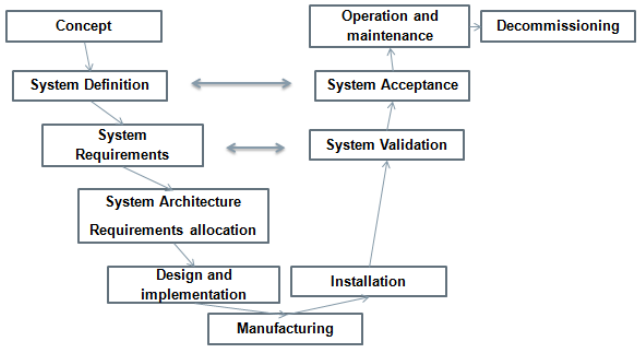


Figure 1: en50126 standard v-cycle

Moreover, functions often include sub-functions that are shared with other functions, thus multiple re-

dundancies occur that affects the documents size and comprehension.

The process has the advantage of using the natural language that expresses more than any model or drawing can do. Moreover, engineers have been using this process for decades and developed multiple systems. This process has been spread to the different entities in the rest of the life-cycle which adapted their activities given these inputs. However, systems becoming bigger and more complex, multi-domain and multi-cultural engineers were asked to work together, pointed out system specifications issues. Today, the system engineering process is complex and based on the knowledge of experts. It takes more and more time to develop new functions because multiple verifications have to be done in order to ensure all the interfaces have been taken into account within the existing system. In addition, the en50126 standard requires the system validation to be launched in parallel with the system engineering but it is rarely true because of the background differences between teams, the missing information and the complexity of the documents produced.

2.2 *Safety analysis process in railway systems*

According to the standard EN50126(EN50126 1999), the RAMS(Reliability Availability Maintainability Safety) activities have to be performed in parallel with the engineering and design phases.

As depicted in figure 1. RAMS activities focus on a review of achieved RAM and safety performances starting with a preliminary hazard analysis. Safety related and functional requirements are produced. Then, the requirements are refined and allocated to components and sub-components. Finally, the safety plan is produced and applied during the next development and production phases.

In the railway industry, and in particular CBTC systems, the safety analysis phase cannot process until late in the design phase. It is due to the lack of cooperation between teams in terms of common techniques and methods.

The safety analysis is divided into three successive parts: the PHA (Preliminary Hazard Analysis), the FMEA (Failure Modes and Effects Analysis) and finally the FSA (Functional Safety Analysis).

The PHA is used as a safety overview of the system to identify the sensitive parts that shall be analyzed. It results the identification of the hazardous events that may lead to an accident, the potential causes for each accidental event, the ranking of the identified events according to their severity, the system functions and equipments that are involved.

The FMEA is a systematic pro-active method for evaluating a process to identify where and how it might fail and to assess the relative impact of different failures. The main result is the qualification (safety relevant or not) of system requirements (functional and data flow).

Finally, the FSA is produced with the results from the previous phases, and looks for failure scenarios causing the hazardous events highlighted in the PHA. Then scenarios are checked whether they are covered by system requirements or if there is no chance for them to occur in the given configuration of the system. Otherwise, safety requirements are defined and correction are requested. FMEA is an inductive approach which evaluates the impact of every failure mode of a function and a data fault on the system, whereas FSA is a deductive approach. From a potential cause that occurs in a main function, we try to find backwards the errors and faults that can lead to it.

Standard EN50126 has been written by experts aware of the difficulties met in railway systems. Industrials can spend a decade conceptualizing, developing, validating and delivering a CBTC solution. Thus, early detection of errors and systematic techniques for system conception and also a rigorous management of the requirements is crucial. In the following, we introduce *ScOLA*, a scenario oriented modeling language that aims at reducing the gap between specifications used for the comprehension of the system and the models used for tool-based safety analysis.

3 SCOLA: A SCENARIO ORIENTED MODELING LANGUAGE FOR RAILWAY SYSTEMS

In order to describe a complex system and more specifically railways, two elements are important: the system architecture and its behavior.

In practice, multiple steps are discarded. The system architecture is defined in specifications with the hierarchical decomposition of the physical components and an informal description of the system functions. The behavior is implicit in the functions definition, and a list of operational scenarios are described.

With *ScOLA*, we focus on gathering the CBTC system concepts and formalizing their description. The systems description is based on the functional scenarios.

In this paper, we present an overview of the *ScOLA* language, we intentionally did not fully present the language because of the paper length limitation.

3.1 *Definition of ScOLA*

ScOLA is an extension of the formalism of SDLg(Issad et al. 2014). It introduces a novel approach for system specifications. It is based on numerous concepts that contain the system structure and its behavior. These concepts have been determined through the system architecture and its multiple views (functional, structural and event-based). In order to define those concepts, *ScOLA* relies on the informa-

tion provided by the different views of the system and its operational scenarios.

Definition 1. *$\mathcal{M}$ is a model in* ScOLA*, it is represented in the quadruplet $< \mathcal{S}, \mathcal{AC}, \mathcal{L}, \mathcal{O}p >$ where:*

- *$\mathcal{S}$ is the finite set of scenarios that describe the behavior of the system;*
- *$\mathcal{A}$ is the finite set of atomic actions the scenarios are built of ;*
- *$\mathcal{C}$ is the set of physical components that build the structural architecture of the system;*
- *$\mathcal{L}$ is the set of possible abstraction levels of the system;*
- *$\mathcal{O}p$ is the finite set of operators that describe the behavior of the previous concepts.*

1. **Concept of component:**
   Let $c \in \mathcal{C}$ be a system component. It is characterized using the triplet $< Id_C, \mathcal{A}(c), \mathcal{C}(c), L_c >$ where:
   - Id_C is the unique identifier of c;
   - $\mathcal{A}(c)$ is the set of actions allocated to the component c;
   - $\mathcal{C}(c)$ is the set of the components children if it applies, empty otherwise.
   - L_c is the level of abstraction where the component is defined.

   Given its complexity, c can either be:
   - basic: when c represents an atomic component that cannot be decomposed;
   - complex: when it still can be decomposed into multiple elements.
2. **Concept of scenario:**
   A scenario describes a step in the behavior of the system. It encapsulates multiple scenarios or atomic actions. Each scenario describes a partial view of the system behavior. A scenario $s \in \mathcal{S}$ is defined as $s = < Id_s, L_s, \mathcal{F}(s) >$, where:
   - Id_s: is the unique identifier of the scenario ;
   - L_s: is the s's corresponding abstraction level;
   - $\mathcal{F}(s)$ is the set of scenarios or actions encapsulated in s.
3. **Concept of action:**
   In *ScOLA*, we define actions in interaction as the behavior of the system. An action $a \in \mathcal{A}$ is defined as atomic, it cannot be decomposed.
   a is defined using the following triplet $a = < Id_a, \mathcal{C}(a), l_i, \mathcal{T}(a) >$ where:
   - Id_a: is the unique identifier of a;
   - $\mathcal{C}(a)$: is the components realizing a;
   - L_a: is the abstraction level of a, $i \in \mathbb{N}$;
   - $\mathcal{T}(a)$: is the corresponding type of the action.

   Moreover, as a may require input data and/or produce results, such an action when atomic may be one of the following types:
   - *Simple action* when it requires the resources of a single component to be completed. This type of action may require input data, that may be provided by one or several other actions. The input data, if there are any, are analyzed in order to generate an output result, after some process and calculation. Formally, let $\mathcal{A}_s$ be the set of *simple actions*.

     If $a \in \mathcal{A}_s$, then $\exists c \in \mathcal{C}$ such as $a \in \mathcal{A}(c)$.
   - *Transfer action* when an action is shared between two or more components. Such an action can be a data transmission between two components of the system, and thus requires the cooperation of both components. Let $\mathcal{A}_t$ be the set of *transfer actions*. If $a \in \mathcal{A}_t$ then $\exists c_1, c_2 \in \mathcal{C}$ such as $a \in \mathcal{A}(c_1) \cap \mathcal{A}(c_2)$.
   - *Question action* when it allows a allows the system to choose between two or more alternative behaviors. Typically, a question action can be a test on data in order to choose which action to proceed within the next step. Let $\mathcal{A}_q$ be the set of *question actions*. If $a \in \mathcal{A}_q$ then $\exists a_1, a_2, \ldots, a_n \in \mathcal{A}$ such as executing a leads to the execution of a_1 *or* a_2 *or* $\ldots$ *or* a_n.

   Assuming that each component in $\mathcal{C}$ has its own resource to realize an action, a transfer action a, $a \in \mathcal{A}$ ($\mathcal{A}$ being the set of all system actions), between two components c_1 and $c_2 \in \mathcal{C}$ is an action that requires the resources of both components to be completed, that is $a \in \mathcal{A}(c_1) \cap \mathcal{A}(c_2)$, where $\mathcal{A}(c_1)$ and $\mathcal{A}(c_2)$ are the set of actions executed by c_1 and c_2, respectively. We can say that c_1 and c_2 are in *cooperation* for a. However, if a requires the resource of component c_1 solely, then a is not a transfer action and c_1 is *assigned* to a.
4. **Concept of refinement**
   Because the different views of the system architecture may provide too detailed functions (functional view), components (organic view) and events (event-based view), it becomes necessary, during the system engineering process, to structure these information and introduce a certain hierarchy between them. Thus, we introduce the notion of *refinement* as one of our main language element.

3.2 *ScOLA operators*

ScOLA provides a small but effective set $\mathcal{O}p$ of operators. Operators are provided to describe the relationships between scenarios or actions. Their idiomatic textual and graphical representations are presented along with the case study in section 5.

- **Precedence** ($\rightarrow$) models the sequential completion of the actions or scenarios.

If a_1,$a_2 \in \mathcal{A}$, a_1 and a_2 follow a precedence order if $t_0(a_2) \geq t_1(a_1)$. $t_1(a_1)$ is the starting time of action a_1 and $t_0(a_2)$ is the end time of action a_1.

- **Parallelism** ($||$) models the independency in the actions or scenarios realization.

If $a_1, a_2 \in \mathcal{A}$, $\exists\ [t_1, t_2]$ such that $t_0(a_1), t_1(a_1) \in$

$[t_1, t_2]$, then $t_0(a_2)$, $t_1(a_2) \in [t_1, t_2]$. In *ScOLA*, parallelism represents a particular case of precedence where $t_0(a_2) \geq t_1(a_1)$ or $t_0(a_1) \geq t_1(a_2)$.

- **Preemption** (+) models the choice between two actions or scenarios of the system. If a,a_1,$a_2 \in \mathcal{A}$:

$$\begin{cases} a \text{ is followed by } a_1 & \text{if a is true} \\ a \text{ is followed by } a_2 & otherwise \end{cases}$$

- **Refinement** models the the refinement of an action a_1 of abstraction level l_n to an action a_2 of abstraction level l_{n+1}.

In the textual and graphical notation, we use the encapsulation.

Others are defined to model the relationship between actions/scenarios and components:

- **Assignment** models a component $c \in \mathcal{C}$ that realizes an action $a \in \mathcal{A}$ or a scenario $s \in \mathcal{S}$.
- **Cooperation** models a *transfer action* a from component c_1 to component c_2.
- **Access** models the access to sub-component c_1 of component c.

4 SAFETY ANALYSIS AND SCOLA

Safety analysis is defined in (Bahr 2014) as *the application of engineering and management principles, criteria, and techniques to achieve acceptable mishap risk, within the constraints of operational effectiveness and suitability, time, and cost, throughout all phases of the system life cycle* .

More recently, Model Based Safety Analysis (MBSA) approaches have been proposed((Arnold et al. 1999)(Papadopoulos et al. 2011)(Yakymets et al. 2013)). Their goal is to introduce mathematical artifacts in which system and safety engineers use the same system models.

Because of the systems and models complexity and differences, safety analysis cannot be automatically generated from the systems description models. It is even prohibited in certain domains like aeronautics where system and safety engineers must work separately to make the analysis viable. The most realistic objective is a contribution to safety analysis with complete and formal system models. This is the objective of *ScOLA* whose perspectives towards safety can be conducted starting from the language. We identify a technique that can help the synchronization between system and safety models. Being at the methodological phase, we roughly present them, and apply them on a concrete case study.

Failure scenarios creation: Safety analysis techniques in the railway industry are based on potential accidents that are refined until identifying the Failure scenarios that lead to the accident. The approach is deductive, it is railway oriented since the potential accidents in this field are not numerous and represent an input to the analysis. It is depicted in figure 2.

In this paper, we propose an inductive approach to

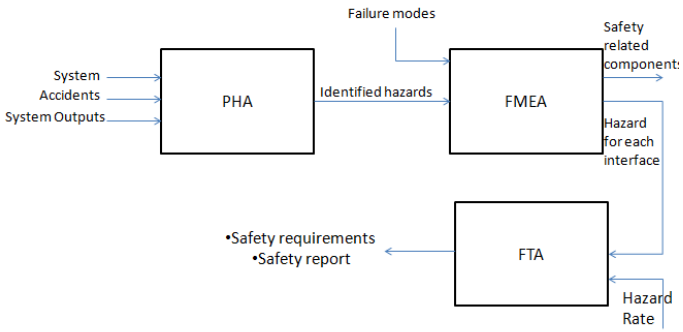


Figure 2: Functional Safety Analysis of a CBTC system

identify *Failure scenarios* starting from the functional ones in the specifications, taking into account the following inputs:

- The system specifications and requirements.
- The failure modes: Data corruption (Action failure, interface failure), non-transmitted data.

The methodology, which is similar to a fault propagation technique, can be summarized in the following steps:

- *Safety relevance of actions*: each requirement in the system specification is followed by a safety relevance information. Since requirements are associated with every step of a scenario, the information is spread over the actions.

Example: if we consider the $step_1$ in a scenario described in the specification, req_1 is its corresponding requirement. Req_1 is defined as being safety relevant, thus $step_1$ is safety relevant, as for the actions encapsulated in $step_1$.

- *Safety relevant data*: at this point of the analysis, we consider the different safety relevant inputs and outputs of the actions, and their respective failure modes. We process as for the actions.
- *Failure actions creation*: every action can have, given its type and inputs/outputs, corresponding failure:

– Simple action: typically, the action can fail. Thus, it can produce a corrupted or wrong data.
– Transfer action: components share an information, this information can simply not arrive because of an interface failure, or be corrupted.
– Test action: in this case, we can consider the wrong decision in the choice process.

- *Failure scenarios creation*: Using the actions created in the previous phase, engineers can choose to insert one or multiple wrong actions and thus create Failure scenarios.

The advantage with this technique is that, considering the system complete, all the hazardous actions can be discovered. Using the already-known failure modes, the safety relevance of actions and the functional scenarios, it is easy to place each action in its context. The use of *ScOLA* models for scenarios helps applying failure modes in a systematic modes since each action possesses a specific type. Solely, the amount of possible Failure scenarios might be impor-

tant and requires a decision process on how to effectively choose them.

In the next section, we present a CBTC system case study constructed using the system specifications of the TGMT product of Siemens.

5 CASE STUDY

In order to understand how we build a *ScOLA* model starting from the existing textual specifications. we present the scenario S = *the train doors supervision under driver's responsibility*. Then, starting from a *ScOLA* model, we will apply the safety technique introduced in the previous section.

S requires the train doors to be opened and closed under the drivers responsibility after the driver has enforced a door release. In the continued scenario, when the driver closes the doors, they are not reported as closed by the wayside. The driver overrides the door closed and locked supervision to continue the train run. In S, the initialization for closing and opening the doors from the train and the communication with the wayside components are optional and not shown in detail. The supervisions of the doors are equally described scenarios.

First, we define the physical components that intervene in the S. The system is composed of a Train, a Wayside and a driver. The train has three basic-components (OBCU, HMI, Train control) and the Wayside has two basic-components (WCU_ATP, WCU_TTS). The driver is already basic.

According to the specification, S is composed of the following steps :

- *Step 1:* The train detects standstill but train doors are not released by the on-board subsystem because the train has not stopped correctly. HMI does not display the door release.
- *Step 2:* The driver selects an enforced door release on the HMI for one side of the train and presses the acknowledge button to acknowledge his selection.
- *Step 3:* The enforced door release activates the door release on the selected side. The train transmits the release status to the wayside. The HMI displays the enforced door release status.
- *Step 4:* The driver initiates the door opening process for a specific side by pushing the open button.

If the train has recognized the stop at platform, the train now sets a door open authorization to the wayside and revokes propulsion release. The train continues to indicate an undefined platform stopping point because the train has not stopped correctly.

- *Step 5:* If the train recognizes stop at platform, the train sends a *doors open* command for the side selected by the driver to the wayside. With a delay, the train also opens the train doors on that side. The doors do not open because the wayside does not know at which stopping point the train has stopped. If the train has not recognized stop at platform, the train opens the train doors on the side selected by the driver.
- *Step 6:* The train doors open. The train becomes inactive. The train indicates the train doors opening to the HMI and to the wayside. The train indicates the speed as zero to the HMI.
- *Step 7:* If stop occurs at platform, the driver opens the doors manually. As soon as doors open, the wayside sets a restrictive RAUZ(Run AUthorization Zone) and sends this information to the wayside. The train receives the RAUZ and indicates the doors as open to the HMI.

S as it is described in the system specification, is a low level description reserved for experts. Thus, following the *ScOLA* semantics and syntax, the scenario is decomposed regarding the fact that a scenario represents a set of actions of scenarios executed by one or two components using multiple refinements. We use the following notation for actions: $a_{\alpha,\beta}$: where α is the corresponding abstraction level of a and β is the sequence number of the action in S, same for scenarios.

- $a_{0,1}$: The train detects standstill
- $s_{0,2}$: The driver selects the doors release side and acknowledges the selection to the train
- $s_{0,3}$: The train transmits the door release to the wayside
- $s_{0,4}$: The train displays the door release status
- $s_{0,5}$: The driver initiates the door opening process for the specific side
- $s_{0,6}$: If the train has recognized the stop at platform. $s_{0,6}$ being a test scenario, it is followed by the sequence ($s_{0,7a}$,$s_{0,8a}$, $s_{0,9a}$, $s_{0,10a}$ and $s_{0,11a}$) or ($s_{0,7b}$,$s_{0,8a}$, $s_{0,9a}$ and $s_{0,10a}$)
- $s_{0,7a}$: The train sets a door open authorization to the wayside.
- $s_{0,8a}$: The train revokes the propulsion release.
- $s_{0,9a}$: The train continues to indicate an undefined platform stopping point.
- $s_{0,10a}$: The train opens the train doors.
- $s_{0,11a}$: The wayside does not open the platforms.
- $s_{0,7b}$: The train opens the doors on the side selected by the driver.
- $s_{0,8b}$: The train indicates the doors opening to the passengers.
- $s_{0,9b}$: The train indicates the doors opening to the wayside.
- $s_{0,10b}$: The train revokes the propulsion release.
- $s_{0,12}$: If the train stops at platform
- $s_{0,13a}$: The driver opens the platforms manually.
- $s_{0,14a}$: The wayside sets a restrictive RAUZ.
- $s_{0,15a}$: The wayside transmits the information to the train.
- $s_{0,16a}$: The train informs the passengers of the platform opening.

Figure 3 depicts the graphical level l_0 description of the door supervision under driver responsibility scenario. It is characterized by the multiple tests.

Figure 4 represents the textual level l_0 scenario of the door supervision under driver responsibility. The system architecture is defined using the keyword **ar-**

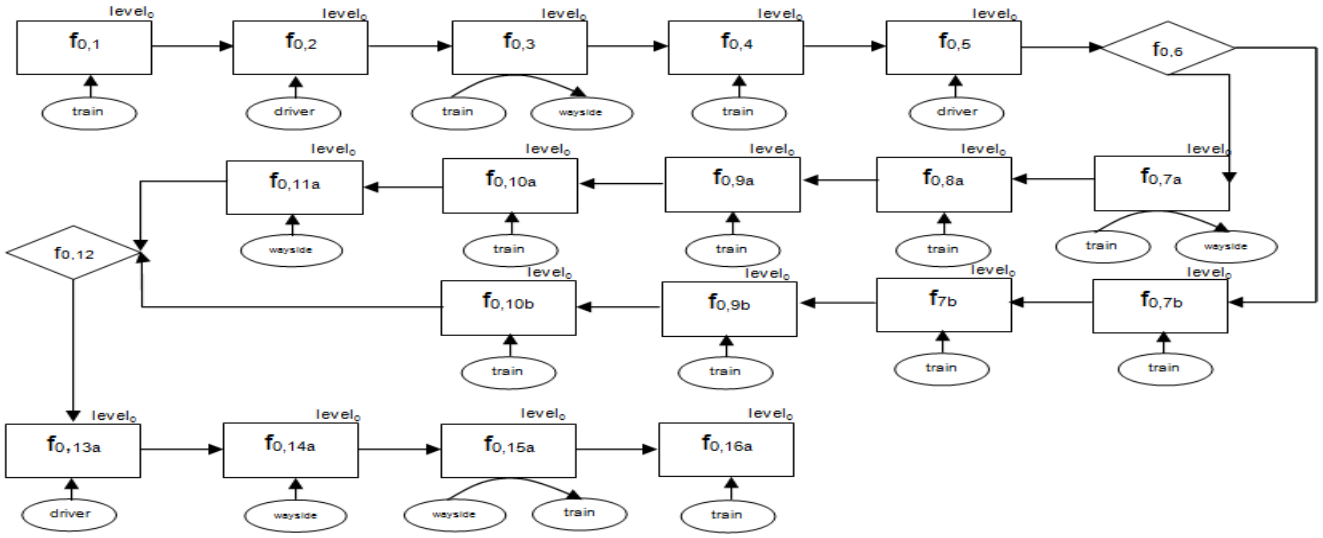


Figure 3: Door Supervision under driver responsibility $level_0$ scenario

chitecture. Instances of components are declared in **blocks** and finally the **scenarios** are described. Their order of completion is defined in the **script**If we decide to have a more concrete view of the system, the different scenarios must be refined.

As an example of the refinement process, we consider scenario $s_{0,2}$: At l_0, it involves solely the driver and the train. At l_0, the allowed components are the driver, the OBCU, the HMI and the Train Control. Thus, $s_{0,2}$ becomes:

- $a_{1,1}$: the driver selects *enforced door release* on the HMI
- $a_{1,2}$: the HMI transmits the *enforced door release* to the train
- $a_{1,3}$: the driver presses the *acknowledge button* on the HMI
- $a_{1,4}$: the HMI transmits the *acknowledgment* to the OBCU

Safety Analysis of S:

We consider the scenario $s_{0,2}$.

1. *Failure scenarios creation*:

Regarding its corresponding requirement in the system specifications, the scenario is safety related, thus it is fully part of the analysis.

The four actions of $s_{0,2}$ have basically two inputs:

- *HMI_I_Enforced_Door_Release*: it represents a signal sent to the hmi then the train. Its only failure mode is the *non-transmission*.
- *CAB_I_Acknowledge_Button*: it represents a signal sent to the hmi then the train. Its only failure mode is the *non-transmission*.

Figure 5 shows the interfaces created between basic components. In the scenarios definition, we can explicit the data transmission using the interfaces previously defined in the architecture and initialized in the blocks.

Regarding the actions themselves, they are typed as *transfer actions*. Therefore, their respective possible failure modes can be a components failure causing a non-transmission of the information or its corruption, or a failure in the interface between the components causing a non-arrival of the information. In the first case we consider that every following action realized by that specific component will fail, and in the second case, every similar interface will fail. In any case, the information will not arrive.

Considering the relationship between actions, $a_{1,1}$, $a_{1,2}$, $a_{1,3}$ and $a_{1,4}$ have a precedence relationship. Meaning that a single failure in one action causes the failure of the entire scenario because it cannot proceed.

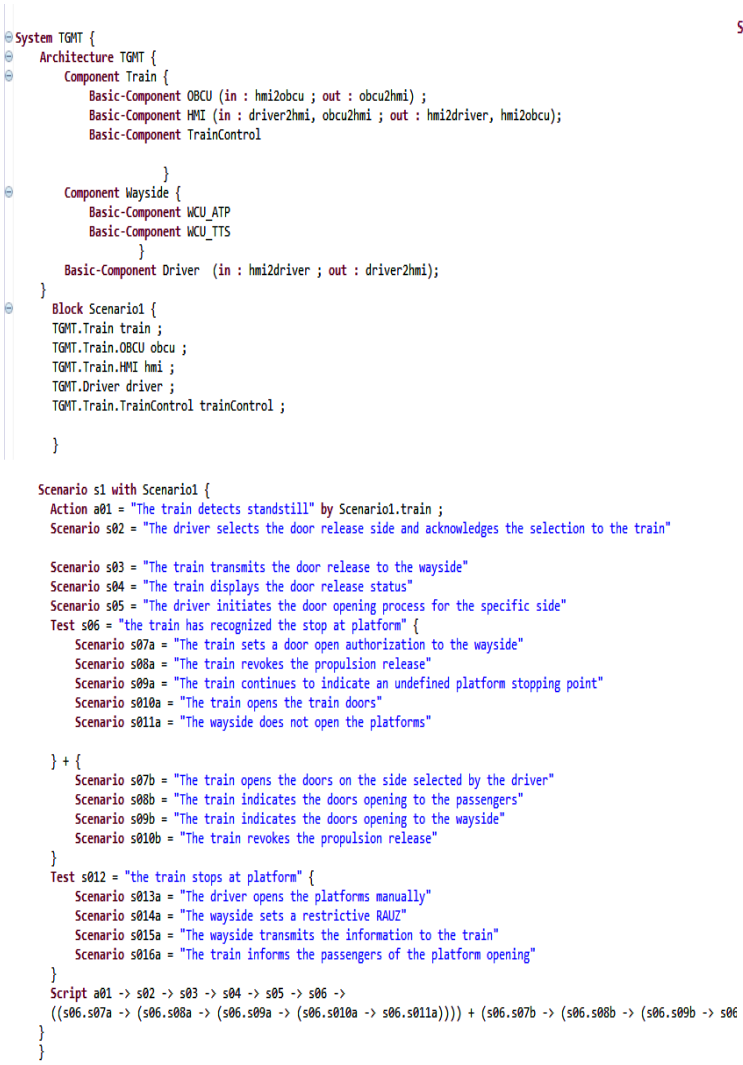
Using the previous information about the actions and their interfaces, we can identify the following Failure scenarios built on $s_{0,2}$:

(a)*Failure of* $a_{1,1}$ *because of an interface failure*: the driver fails at sending the *enforced door release* to the HMI, $a_{1,2}$: the HMI sends an outdated *enforced door release* to the OBCU. The driver presses the acknowledgment button to the HMI, the HMI transmits the acknowledgment to the train. The result of the scenario is the transmission of a wrong information to the train about the doors release which can cause an unexpected train doors opening for example.

(b)*Failure of* $a_{1,1}$ *because of a driver failure*: the driver fails at sending the *enforced door release* to the HMI, $a_{1,2}$: the HMI sends an outdated *enforced door release* to the OBCU. The driver fails at sending the acknowledgment button to the HMI, the HMI fails at transmitting the acknowledgment to the train. The result is an unexpected stop in the overall progress of the scenario.

6 CONCLUSION

In this paper, we address issues on system safety analysis in railway systems. In order to have system



```
System TGMT {
    Architecture TGMT {
        Component Train {
            Basic-Component OBCU (in : hmi2obcu ; out : obcu2hmi) ;
            Basic-Component HMI (in : driver2hmi, obcu2hmi ; out : hmi2driver, hmi2obcu);
            Basic-Component TrainControl

                        }
        Component Wayside {
            Basic-Component WCU_ATP
            Basic-Component WCU_TTS
                    }
        Basic-Component Driver  (in : hmi2driver ; out : driver2hmi);
    }
     Block Scenario1 {
     TGMT.Train train ;
     TGMT.Train.OBCU obcu ;
     TGMT.Train.HMI hmi ;
     TGMT.Driver driver ;
     TGMT.Train.TrainControl trainControl ;

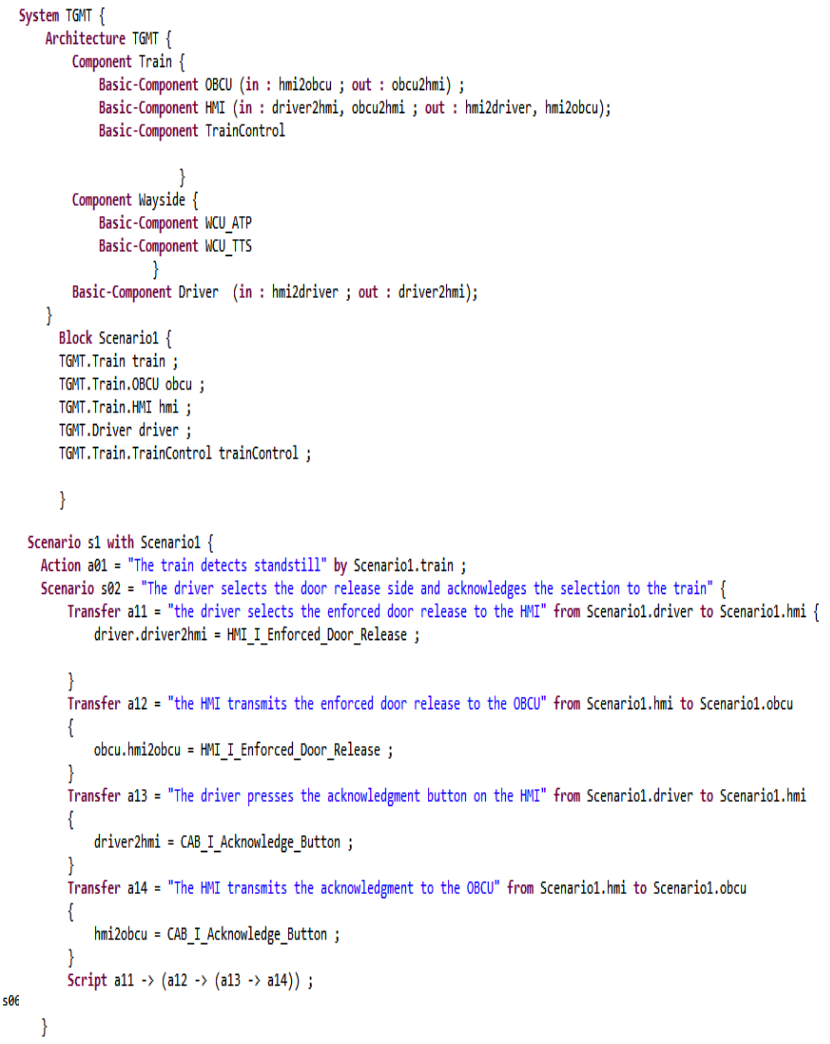
     }

Scenario s1 with Scenario1 {
  Action a01 = "The train detects standstill" by Scenario1.train ;
  Scenario s02 = "The driver selects the door release side and acknowledges the selection to the train"

  Scenario s03 = "The train transmits the door release to the wayside"
  Scenario s04 = "The train displays the door release status"
  Scenario s05 = "The driver initiates the door opening process for the specific side"
  Test s06 = "the train has recognized the stop at platform" {
      Scenario s07a = "The train sets a door open authorization to the wayside"
      Scenario s08a = "The train revokes the propulsion release"
      Scenario s09a = "The train continues to indicate an undefined platform stopping point"
      Scenario s010a = "The train opens the train doors"
      Scenario s011a = "The wayside does not open the platforms"

  } + {
      Scenario s07b = "The train opens the doors on the side selected by the driver"
      Scenario s08b = "The train indicates the doors opening to the passengers"
      Scenario s09b = "The train indicates the doors opening to the wayside"
      Scenario s010b = "The train revokes the propulsion release"
  }
  Test s012 = "the train stops at platform" {
      Scenario s013a = "The driver opens the platforms manually"
      Scenario s014a = "The wayside sets a restrictive RAUZ"
      Scenario s015a = "The wayside transmits the information to the train"
      Scenario s016a = "The train informs the passengers of the platform opening"
  }
  Script a01 -> s02 -> s03 -> s04 -> s05 -> s06 ->
  ((s06.s07a -> (s06.s08a -> (s06.s09a -> (s06.s010a -> s06.s011a)))) + (s06.s07b -> (s06.s08b -> (s06.s09b -> s06
}
}
```

Figure 4: Textual representation of the scenario



```
System TGMT {
    Architecture TGMT {
        Component Train {
            Basic-Component OBCU (in : hmi2obcu ; out : obcu2hmi) ;
            Basic-Component HMI (in : driver2hmi, obcu2hmi ; out : hmi2driver, hmi2obcu);
            Basic-Component TrainControl

                        }
        Component Wayside {
            Basic-Component WCU_ATP
            Basic-Component WCU_TTS
                    }
        Basic-Component Driver  (in : hmi2driver ; out : driver2hmi);
    }
     Block Scenario1 {
     TGMT.Train train ;
     TGMT.Train.OBCU obcu ;
     TGMT.Train.HMI hmi ;
     TGMT.Driver driver ;
     TGMT.Train.TrainControl trainControl ;

     }

Scenario s1 with Scenario1 {
  Action a01 = "The train detects standstill" by Scenario1.train ;
  Scenario s02 = "The driver selects the door release side and acknowledges the selection to the train" {
      Transfer a11 = "the driver selects the enforced door release to the HMI" from Scenario1.driver to Scenario1.hmi {
          driver.driver2hmi = HMI_I_Enforced_Door_Release ;

      }
      Transfer a12 = "the HMI transmits the enforced door release to the OBCU" from Scenario1.hmi to Scenario1.obcu
      {
          obcu.hmi2obcu = HMI_I_Enforced_Door_Release ;
      }
      Transfer a13 = "The driver presses the acknowledgment button on the HMI" from Scenario1.driver to Scenario1.hmi
      {
          driver2hmi = CAB_I_Acknowledge_Button ;
      }
      Transfer a14 = "The HMI transmits the acknowledgment to the OBCU" from Scenario1.hmi to Scenario1.obcu
      {
          hmi2obcu = CAB_I_Acknowledge_Button ;
      }
      Script a11 -> (a12 -> (a13 -> a14)) ;

}
```

Figure 5: Textual representation of the scenario with the data

and safety engineers to communicate, it is important to structure and formalize the way systems are designed. In this perspective, we designed *ScOLA*; a formal modeling language based on scenarios describing the system architecture and behavior. We explain how; starting from formal but explicit models of the system, safety analysis could be performed more efficiently. We introduced some identified safety techniques and their application to a cbtc railway case study.

We are currently implementing those techniques. We are also working on how *ScOLA* models could be used with existing model based safety analysis existing tools and languages.

REFERENCES

Abdulla, P. A., J. Deneux, G. Stålmarck, H. Ågren, & O. Åkerlund (2006). Designing safe, reliable systems using scade. In *Leveraging Applications of Formal Methods*, pp. 115–129. Springer.

Abrial, J.-R. (2005). *The B-Book: Assigning programs to meanings*. Cambridge University Press.

Arnold, A., G. Point, A. Griffault, & A. Rauzy (1999). The altarica formalism for describing concurrent systems. *Fundamenta Informaticae 40*(2), 109–124.

Bahr, N. J. (2014). *System safety engineering and risk assessment: a practical approach*. CRC Press.

Cimatti, A., E. Clarke, F. Giunchiglia, & M. Roveri (1999). Nusmv: A new symbolic model verifier. In *Computer Aided Verification*, pp. 495–499. Springer.

EN50126, C. (1999). Railway applications: The specification and demonstration of reliability. *Availability, Maintainability and Safety (RAMS)*.

Feiler, P. H., D. P. Gluch, & J. J. Hudak (2006). The architecture analysis & design language (aadl): An introduction. Technical report, DTIC Document.

Friedenthal, S., A. Moore, & R. Steiner (2011). *A practical guide to SysML: the systems modeling language*. Elsevier.

IEEE (2004). *Standard for Communications Based Train Control (CBTC) Performance and Functional Requirements. IEEE Std 1474.1-2004 (Revision of IEEE Std 1474.1-1999)*.

Issad, M., L. Kloul, & A. Rauzy (2014). A model-based methodology to formalize specifications of railway systems. In *Model-Based Safety and Assessment*, pp. 28–42. Springer.

Leonard, J. (1999). Systems engineering fundamentals. Technical report, DTIC Document.

Ong, C.-M. (1998). *Dynamic simulation of electric machinery: using MATLAB/SIMULINK*, Volume 5. Prentice Hall PTR Upper Saddle River, NJ.

Papadopoulos, Y., M. Walker, D. Parker, E. Rüde, R. Hamann, A. Uhlig, U. Grätz, & R. Lien (2011). Engineering failure analysis and design optimisation with hip-hops. *Engineering Failure Analysis 18*(2), 590–608.

Pascoe, R. & T. Eichorn (2009). What is communication-based train control? *IEEE Vehicular Technology Magazine 4*(4), 16–21.

Yakymets, N., H. Jaber, & A. Lanusse (2013). Model-based system engineering for fault tree generation and analysis. In

MODELSWARD, pp. 210–214.