Modeling the CBTC railway system with ScOLA

Melissa Issad, Leila Kloul, Antoine Rauzy, Karim Berkani

22nd ITS World Congress, Bordeaux, France, 5–9 October 2015

Paper number ITS-1930

Modeling the CBTC railway system with *ScOLA*

Melissa Issad[1,3*], Leila Kloul[2], Antoine Rauzy[1] and Karim Berkani[3]

1. LGI, Ecole Centrale Paris, France

2. PRiSM, University of Versailles, France

3. Siemens Mobility, France

Abstract

Considering their increasing complexity, industrial systems are, in general, specified in a natural language. Especially transportation systems where the design phase results an ambiguous and laborious system specification. The objective of this paper is to present *ScOLA*, a formal modeling language based on scenarios and built on railway system specifications. Its novelty is based on its restriction to the core concepts of the specification and its multiple representations (textual and graphical), and also on its formal semantics. The language offers means to understand what the system was supposed to do and to be as well as to support a dialog with experts so to be sure that we got everything correctly. The methodology is applied on the railway automation solution Trainguard MT CBTC of Siemens.

Keywords:

System engineering, *ScOLA*, railway systems.

Introduction

Siemens is one of the leaders in the railway automation solutions. Especially France who holds an international center of competences for driverless systems, it drives the development of fully automated trains of first and second generation (respectively Val and Cityval) and Airval for airports transportation. Compared to the systems with drivers, automation allows a higher commercial speed and reduced intervals between trains. It also increases the operational flexibility with systems running full time. These systems have been deployed over more than 30 lines like the line1 and 14 in Paris; line L in New York...etc.

The railway automation is based on the CBTC system which is based on principle that trains determine their positions themselves and transmit it to wayside equipments. CBTC assures that the space between trains is always safe [3]. Given the complexity of the system, their specifications spread over thousands of pages written in a natural language, which makes it

difficult for engineers to develop, validate and maintain.
In theory, regarding the V-cycle (Figure1) for complex systems, systems must be defined in specifications, analyzed seeking for system acceptance and then developed and integrated and finally validated. But in practice, the system analysis phase is often let at the very last steps of the design which brings a late detection of errors and ambiguities.

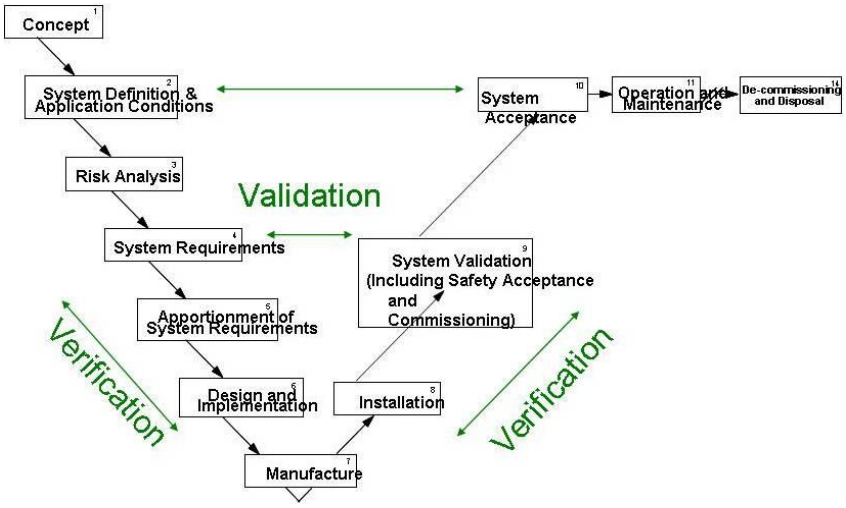

Figure 1 - Railway systems V-Cycle according to EN 50126

At INCOSE [1], they promote multiple model-based system engineering (MBSE) methodologies. Their goal is to widen the use of models instead of documents in the system engineering process. They define MBSE as the formalized application of modeling to support system requirements, design, analysis, verification and validation, beginning in the conceptual design phase and continuing throughout development and later life cycle phases. They also promote the fact that modeling should be used at multiple levels of the system (operational, system and component).
But often, modeling languages appeared and served as a graphical representation of the specification (UML, SysML[2],…). Mainly, two main approaches for system modeling appeared, the language centric, where focusing on a specific modeling language, users will use all the items provided by the language to model the system, the result was often redundant information. We also noticed multiple system centric methods where engineers will modify the language to fit the system. The result is a non-generic methodology. Then quickly, engineers expressed the need to give a meaning to the graphics. They wanted to reuse them for different purposes. But, given the fact than the languages did not clearly rely on a formalism made the operation difficult. With *ScOLA* we wanted to start by defining the adequate modeling formalism and then we created its graphical representation. In this article we present the language extracted from the formalism and its characteristics.

Presentation of the TGMT CBTC railway system

TGMT (Trainguard Mass Transit) is a Siemens customized CBTC (Communication Based

Train Control) system. A CBTC is railway signaling system that uses wired and wireless communications between the train and the track equipments for the traffic and infrastructure control. It significantly improved the way trains were localized.

Old systems used the track occupancy to determine the position of a train while CBTC equipped trains determine independently their localization and forwards it to the track equipments. According to the IEEE 1474 standard, 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.”[3].

The Trainguard MT CBTC [4] 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 (see Figure2).

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 train movement authority according to their speed and position.

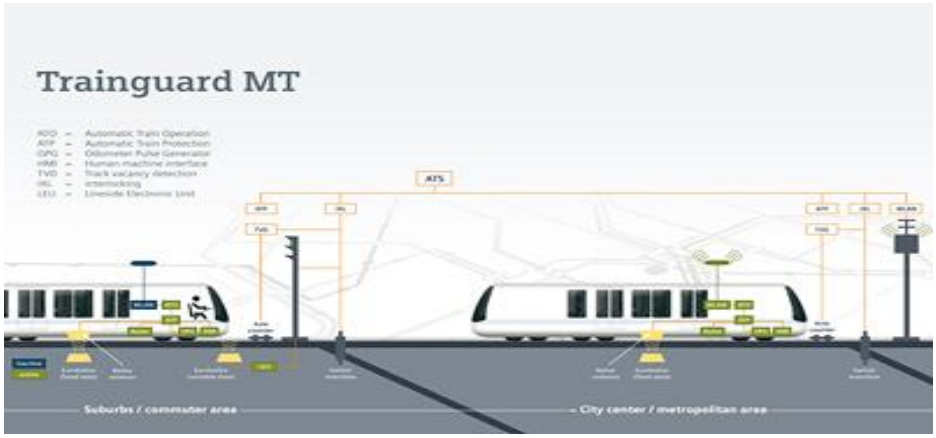

Figure 2 - Trainguard Mass Transit CBTC system

In the following sections, we present the modeling language based on the TGMT specification.

ScOLA, a SCenario Oriented LAnguage for railway systems

The approach we adopt to define *ScOLA* is the need for a system conceptualization, meaning that often models are created without any other goal than a graphical representation. Complex systems are composed of a description of its system architecture composed of the hierarchical decomposition of its components and functions, and a behavioral description.

System architecture provides a description of the different parts of the system. It aims at simulating their behaviors and the way they interact. In order to understand the behavior of a complex system, different views of its architecture have to be investigated. These are the

following:

- Functional view: the behavior of a complex system can be described through the actions or the functions the system has to realize in order to achieve its operational missions. The functional view aims at developing a breakdown structure of the system behavior in order to derive all its actions (main and subsidiaries) and all the relationships between them.
- Organic view: using this view, one can describe the system by defining its physical components. From the main component, which is the system itself, all its physical components are derived. These components are the ones realizing, individually or in cooperation, the actions or functions obtained using the functional view. Like the functional view, the organic view is a top-down approach which allows an internal and concrete analysis of the system.
- Event-based view: the relationships between the functions and the components of a system can be determined using event-based view. In general, the events are responsible for the data transmissions in the system. This representation is very useful for further system analysis because it allows differentiating between sensitive data and non-sensitive ones.

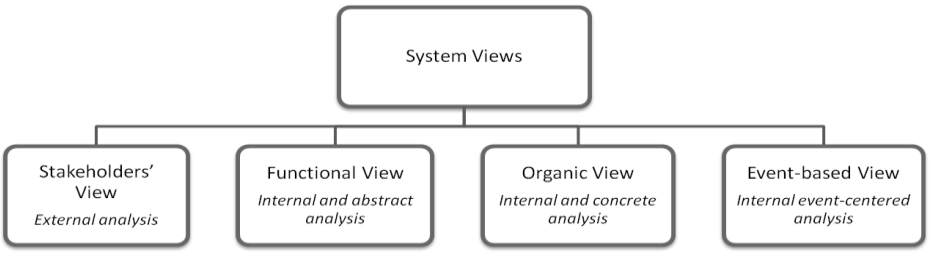

Figure 3 - The System Architecture Definition

Besides the three previous views, an identification of the stakeholders of the system under analysis is necessary. The system can be represented as a black box with different connections:

- The inputs: they are the variables that interfere in the system behavior;
- The outputs: they are the data and the results of the system's deployment;
- The supports: they are all the external stakeholders of the system.

ScOLA 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, organic and event-based). We can then define the system as a set of components *C* which execute functions, either individually or in cooperation. Let *F* be the set of all the functions in the system, and *F(C)* the set of functions in which component *c*, $c \in C$, is involved. Assuming that each component in *C* has its own resource to execute a function, a shared function *f*, $f \in F$, between two components *c1* and *c2* in the system is a function that requires the resources of both components to be completed, that is $f \in F(c1) \cap F(c2)$. However, if *f* requires the resource of component *c1* solely, then *f* is not a shared function.

Moreover, as a function can be defined as an action which may requires input data, which may produce results that may, or may not, be useful for other functions, we consider that a function may be of one of the following types:

- *S_action*: this is a simple action that 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.
- *T_action*: this is a shared action 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.
- *Q_action*: this type of action allows the system to choose between two or more alternative behaviors. Typically, a *Q_action* can be a test which has to be run on data in order to choose which action to proceed with in the next step.

Often the functions describing the behavior of a system are dependent, and thus have to respect a precedence order to receive adequate and coherent information. We consider three types of relationships between functions:

- *Precedence*: the functions have to be completed sequentially. This implies that a function has to wait the completion of another function, before starting.
- *Parallelism*: the execution order of two functions is not important. In this case, one can proceed before the other, independently. Clearly, such a relationship implies that the functions do not share the resources.
- *Preemption*: after a *Q_action*, a function among a set of two or more is chosen to proceed. In this case, the other functions are discarded.

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. Solely, we introduce the notion of abstraction level as one of our main language element.

Concept of Abstraction Level

This concept produces a refinement between actions and components. It helps us distinguish between high-level functions that can be useful. For example, in safety analysis, in order to define the top events a potential accident that can happen on the system, it is better to have a high level description of the system. Later on, we might need; after multiple refinements; more detailed functions of the system validation and the description of the hardware failures.

This notion allows distinguishing between a high-level function with just enough details to understand the part of the system behavior it represents, and an atomic function which results

from successive refinements of a high-level function into smaller functions of lower level. In order to progress from one level to another, *ScOLA* relies on the information provided by the different views of the system and its operational scenarios.

Modeling a CBTC scenario using *ScOLA*

We can formally define a complex system with all its parts. A scenario is a set of functions of a certain level. Functions are executed in the order defined earlier (precedence, parallelism, and test). A function of level l_n can also be a set of functions of lower levels (level l_{n+1}). A function is executed giving its type by one or many components. These components act individually or in cooperation with other ones. In case of a choice, a test determines which action is realized.

In this section, we will explain how, starting from an informal description of a CBTC scenario, we define a formal model using *ScOLA* and having both textual and graphical representations.

The Speed-Dependent Door Supervision Scenario

Here we present the scenario of Speed-Dependent Door Supervision. It is a scenario that involves the train and the platform. It consists of: When the train is fully berthed at a platform and stops, the trains doors are released and opened. Then, the train starts to roll away. The emergency brake is applied when the train exceeds a certain minimum speed. The physical components involved in the scenario are represented in a tree (Figure 4). Since the specification is a low level description, it gives the more detailed components. It is composed according to the specification of different steps:

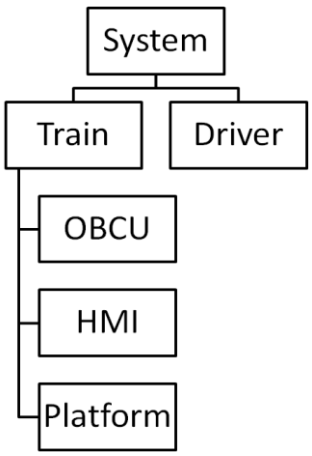

Figure 4 - Architecture of the components involved in the Scenario1

- Step 1: The train approaches the stopping point, it is already fully berthed. The on-board sub-system indicates this to the HMI.
- Step 2: The train comes to a standstill; the on-board subsystem releases the train doors at the correct side.

- Step 3: The driver initiates door opening. The on-board subsystem opens the train doors.
- Step 4: The doors open. This is reported to the on-board subsystem.
- Step 5: The on-board subsystem indicates the open doors to the HMI. It sets the recommended speed to zero.
- Step 6: The train starts to move, the configured minimum speed for the door supervision is not yet exceeded. The on-board subsystem reacts by revoking the door release.
- Step 7: While rolling, the train loses the fully berthed status. The on-board subsystem revokes the fully berthed indication to the HMI.
- Step 8: The train exceeds the configured minimum speed for the door supervision. The on-board subsystem applies an emergency brake.

This scenario is decomposed regarding the different events that happen in the system. We decompose the scenario in a way that highlights the different functions and components of the system. We start by a level l_0 representation:

- $f_{0,1}$: The train arrives at the stopping point.
- $f_{0,2}$: The train releases the train doors.
- $f_{0,3}$: The driver initiates door opening.
- $f_{0,4}$: The driver reports the door opening to the train.
- $f_{0,5}$: The train sets the speed to zero.
- $f_{0,6}$: The train starts to move without exceeding the minimum speed.
- $f_{0,7}$: The train loses the fully berthed status.
- $f_{0,8}$: If the train exceeds the configured minimum speed for the door supervision.
- $f_{0,9}$: The train applies an emergency brake.

Functions can be seen at multiple abstraction levels, if we consider $f_{0,1}$, it can be decomposed into atomic actions of level l_1 . In this particular case, level l_1 represents the last possible abstraction level, but as long as atomicity is not reached, functions can be decomposed. Hence, $f_{0,1}$ is described as:

$f_{0,1}$: The train arrives at the stopping point

- $f_{1,1}$: The on-board subsystem detects that the train is at the stopping point
- $f_{1,2}$: The on-board subsystem indicates the stopping point arrival at the HMI

Textual representation of the scenario

Figure 5 is a partial view of the textual representation of our case study. It is mainly composed of two aspects, the architecture and the scenarios. The architecture depicts the hierarchical decomposition of the system's physical parts. While scenarios using instantiations of

components, are an exhaustive description of the system behavior.

1. Architecture:

ScOLA is an object-oriented language; we have different types of classes. Component classes in order to be used in a scenario must be declared first. A component can be basic (atomic), or can also be a set of components or basic components. After the definitions of the hierarchy of components, we can describe the behavior of the system.

In the scenario, the system is composed of two components: train and driver. Train is also composed of multiple basic components. We start by declaring the component Train that is composed of three (3) basic components OBCU, HMI and Platform. Component Driver is also declared.

2. Scenario structure:

A scenario is an action realized by a component. Given the modularity of the language, we understand that a scenario can be decomposed until reaching an atomic action. Before describing the actions, an instantiation of the components involved in the scenario using the keyword Block, we note that a scenario can contain multiple instantiations of a component in case of redundancy or to represent a complex system involving multiple systems. After that, we can start describing the behavior part of the system that contains multiple scenarios of different abstraction levels, and then follows a script that explains the kind of link between actions. We introduce the following constraints:

- Simple actions are atomic actions that are realized by only one component. They are represented via the keywords **Action**, followed by its description and the component involved introduced by the keyword **By**. Example: **Action** $f_{1,1}$ **By** OBCU
- Transfer actions are also atomic, they are represented by the keyword **Transfer** followed by the issuer of the action introduced by the keyword **From** and the receiver with **To**. Example: **Transfer** $f_{1,2}$ **From** OBCU **To** HMI
- Test actions represent a step in the scenario where, regarding a condition, one among multiple actions will be realized. Its syntax starts with the keyword Test and followed by the actions

```
System TGMT {
    Architecture TGMT {
        Component Train {
            Basic-Component OBCU ;
            Basic-Component HMI ;
            Basic-Component Platform;
                        }
        Component Driver
    }
      Block Scenario0 {
      TGMT.Train train ;
      TGMT.Train.OBCU obcu ;
      TGMT.Train.HMI hmi ;
      TGMT.Driver driver ;
      TGMT.Train.Platform platform ;
      }

     Scenario S0 with Scenario0 {
        Scenario F01= "The train arrives at the stopping point" {
            Action F11 = "The on-board detects that the train is at the stopping point" by Scenario0.obcu ;
            Transfer F12 = "The on-board indicates the arrival to the HMI" from Scenario0.train to Scenario0.hmi ;
            Script F11 -> F12 ;
        }
        Scenario F02 = "The train releases the train doors" {
            Action F11 = "The on-board arrives at standstill" by Scenario0.obcu ;
            Action F12 = "The on-board releases the train doors at the correct side" by Scenario0.obcu ;
            Script F11 -> F12 ;
        }
        Scenario F03 = "The driver initiates door opening" {
            Action F11 = "The driver initiates the train doors opening" by Scenario0.driver ;
            Action F12 = "The on-board opens the train doors" by Scenario0.obcu ;
            Script F11 -> F12 ;
        }
        Script F01 || F02 -> F03 ;
     }
    }
```

Figure 5 - Textual representation of the Doors Supervision scenario

The Script describes the relationship between actions or scenarios, using the following symbols:

- →: Precedence order ($f_1 \rightarrow f_2$)
- ||: Parallelism ($f_1 \parallel f_2$)
- +: Choice ($f_1 + f_2$)

In the example, we model the first three scenarios. We start by declaring the required components (train, driver, obcu, hmi). After that, we describe the scenarios using the keyword Scenario, the simple actions using the keyword Action and the transfer actions with the keyword Transfer. At the end of each scenario, the script (Script) explains for example the first scenario $f_{0,1}$ is in parallel with $f_{0,2}$ followed by $f_{0,3}$ with the given notation: $f_{0,1} \parallel f_{0,2} \rightarrow f_{0,3}$

Graphical representation of the scenario

Figure 7 depicts the level l_0 description of the door supervision scenario. Functions $f_{0,1}$ and $f_{0,2}$ are starting in parallel. Function $f_{0,8}$ represents a test followed by $f_{0,9}$ if true, the scenario ends otherwise.

To graphically model the scenario starting either from its informal description of the textual representation, some rules and idiomatic representations must be followed (Figure 6)

- $\boxed{f}$: f is an atomic or complex function.
- $\textcircled{C}$: C is a component of the system.
- $\boxed{f}$ / $\textcircled{C}$: component C realizes function f.
- $\textcircled{C_1} \curvearrowright \textcircled{C_2}$ when C_1 and C_2 cooperate for a function f.
- $\diamond f$ when f is a Q_action.
- $\boxed{f_1} \rightarrow \boxed{f_2}$ meaning that f_1 and f_2 have a precedence order.
- $\boxed{f_1} \leftrightarrow \boxed{f_2}$ meaning that f_1 and f_2 are in parallel.
- $\boxed{f_1}$ / $\boxed{f_2}$ to represent the refinement, meaning that f_1 is an abstraction of f_2 and f_2 is the concretization of f_1.

Figure 6 - Graphical idiomatic representation of ScOLA models

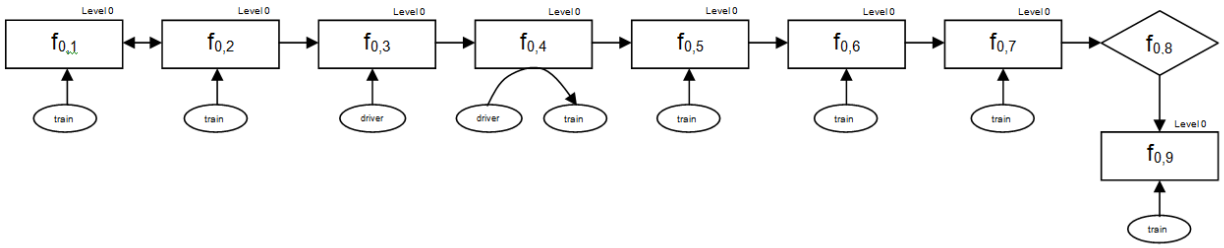

Figure 7 - Door Supervision Scenario at level l_0

Figure 8 represents the level l_1 description of the $f_{0,1}$ function.

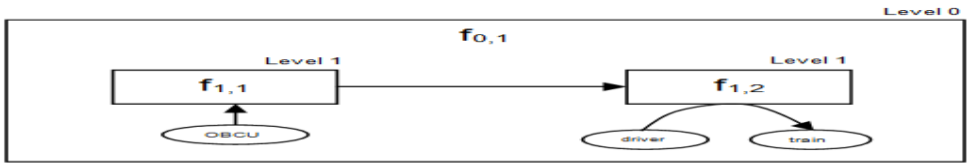

Figure 8 - Function *f0,1* level l_1 description

Conclusion

In this paper we presented *ScOLA*, a Scenario Oriented LAnguage, a domain specific language for railway systems. We tried to explain the importance on focusing on systems concepts in order to have a coherent and non-redundant model. We wanted the language to be

simple but efficient in a way that operators represent all the system's concepts. We also focused on the importance to have textual and graphical representations that is one of the perks of the formal languages. Instead of crossing by semi-formal languages to build a bridge between informal and formal models we decided to simplify a formal description that stays relied to a formal semantics.

Our next objectives are to define a methodology to perform safety analysis. In railway systems, safety is still hand-made and relies on the experience of experts. The need for formal methods is important to discover dysfunctional scenarios and find mismatches in the system specifications. We are confident in the fact that formal functional scenarios will considerably help understand the ambiguous specifications, and we also aim at using functional scenarios to help define the dysfunctional ones.

References

1. Sanford Friedenthal, Regina Griego, Mark Sampson INCOSE Model Based Systems Engineering (MBSE) Initiative INCOSE2007 June 24-27 San Diego
2. Sanford Friedenthal, Alan Moore, Rick Steiner, A Practical Guide to SysML, The Systems Modeling Language, MK/OMG Press, 2009, ISBN 978-0-12-378607-4
3. 1474.1-1999 - IEEE Standard for Communication Based Train Control Performance Requirements and Functional Requirements
4. Trainguard MT CBTC: The moving block communications based train control solutions, Siemens Transportation Systems.
5. Sunny A. Yaung Foundations of complex system theories in Cambridge University Press 1998
6. M.Stollberg, B.Elvester. A Customizable Methodology for the Model-driven Engineering of Service-based System Landscapes
7. Klaus Pohl Requirements Engineering: Fundamentals, Principles, and Techniques 1st Springer Publishing Company, Incorporated 2010 ISBN 978-3642125775
8. E.M. Clarke, J.M. Wing, et al. Formal methods : A state of the art ACM Computing Surveys, Vol. 28, No. 4, December 1996
9. David Harel and P. S. Thiagarajan Message Sequence Charts, In UML for Real: Design of Embedded Real-Time Systems
10. Glinz, M., Berner, S., & Joos, S An Object Oriented Modeling with ADORA Information Systems, 27(6), 425-444.
11. Adolph, S., Cockburn, A., & Bramble, P. Patterns for effective use cases Addison-Wesley Longman Publishing Co., Inc. (August 30, 2002)
12. Buede, D. M. The Engineering Design of Systems, Models and Methods (Vol. 55). John

Wiley & Sons.

13. Araujo, J., Whittle J., & Kim, D. K. Modeling and Composing Scenario- Based Requirements with Aspects In Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International (pp. 58-67). IEEE.
14. Issad, M., Kloul, L., & Rauzy, A. (2014) A Model-Based Methodology to Formalize Specifications of Railway Systems In Model-Based Safety and Assessment (pp. 28-42). Springer International Publishing.
15. France, R. B., Kim, D. K., Ghosh, S., & Song, E. (2004). A UML-based pattern specification technique. Software Engineering, IEEE Transactions on, 30(3), 193-206.