ETCS requirements specification and validation:
the methodology
Introduction .................................................................................................................. 4
1. Overview of the methodology ............................................................................. 5
2. Informal analysis phase ...................................................................................... 7
   2.1. Requirement Fragment isolation ................................................................. 7
   2.2. Requirement categorization ......................................................................... 7
      2.2.1. Examples of requirements classification .............................................. 8
   2.3. Requirements dependencies ....................................................................... 9
3. Formalization phase ............................................................................................. 10
   3.1. UML class diagrams ................................................................................... 10
      3.1.1. Classes .............................................................................................. 10
      3.1.2. Relationships ................................................................................... 11
      3.1.3. Examples of class diagrams in ETCS domain .................................. 12
   3.2. UML state machines .................................................................................. 14
      3.2.1. State .................................................................................................. 14
      3.2.2. Transition .......................................................................................... 15
      3.2.3. Examples .......................................................................................... 16
   3.3. UML Sequence diagrams ........................................................................... 18
      3.3.1. Basic Notation ................................................................................... 18
      3.3.2. Interaction Operators ....................................................................... 19
3.4. Controlled Natural Language .......................................................................... 25
   3.4.1. Semantics of the temporal operators in CNL ...................................... 27
   3.4.2. Examples of use of CNL in ETCS domain .......................................... 28
4. Formal Validation phase ....................................................................................... 30
   4.1. Validation .................................................................................................. 30
   4.2. An example of validation phase ................................................................. 34
   4.3. Example of problem definition .................................................................. 35
      4.3.1. Definition of main class ..................................................................... 35
      4.3.2. Definition of a generalized test scenario via CNL ......................... 36
5. An example of the formalization and problem definition process ...................... 38
   5.1. Classification of requirements .................................................................... 38
   5.2. Creation of the Class Diagram ................................................................. 39
   5.3. Creation of the constraints in CNL ............................................................ 41
5.4. Creation of the state machines .................................................................42
5.5. Creation of the sequence diagrams ........................................................44
5.6. Specification of the validation problem via the creation of a new class..........45
Appendix A: methodology summary ..............................................................47
References........................................................................................................48
Introduction

This document describes the methodology for the formalization of the set of ETCS specifications. The methodology is based on a six steps process: classification of requirements, formalization in UML, definition of the validation problem, translation of the problem into a formal language, check of the problem and analysis of the results. This process is supported by the ETCS tool that is made up of several components (in particular, Microsoft Word, RequisitePro, Rational Software Architect, some plug-ins based on RSA and the NuSMV model checker).

These steps will be detailed in the following sections in terms of the subprocesses, modelling concepts and artefacts to be exploited during the execution of the specification/validation process.
1. Overview of the methodology

The methodology is based on a set of activities, each one supported by a specific component of the ETCS tool. They are: **Classification and structuring of the Requirements** in the ETCS specification -supported by RequisitePro and Microsoft Word and a Rational Software Architect plug-in--; **for each classified requirement, formalize it** specifying the set of concepts and diagrams in UML language, and additional constraints in CNL -supported by Rational Software Architect and some plug-ins to support the specification model traceability services--; **verification problem definition** -supported by RSA plug-ins--; **automatic translation of the verification problem in a formal language** -supported by RSA plug-ins--; **use of formal analysis for validation purposes** -supported by NuSMV and RSA plug-ins--; **analysis of the results** of the check —supported by NuSMV and RSA plug-ins--.

These activities can be summarized in three major methodology steps described in Table 1.

**Table 1: The three phases of the methodology.**

<table>
<thead>
<tr>
<th>Methodology steps</th>
<th>Description of the step</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td><em>Informal analysis phase</em>: it consists of the categorization and structuring of the informal requirement fragment described in the requirements document to produce categorized requirement fragments</td>
</tr>
<tr>
<td>M2</td>
<td><em>Formalization phase</em>: the categorized requirement fragments are described through the set of concepts and diagrams in UML, and additional constraints in the defined CNL to produce formalized requirement fragments</td>
</tr>
<tr>
<td>M3</td>
<td><em>Formal validation phase</em>: it consists of the identification of a subset of the formalized requirement fragments (together with the definition of a series of validation problems) for an automatic validation analysis</td>
</tr>
</tbody>
</table>

In the next sections the steps will be detailed in terms of the subprocesses, artefacts and modelling concepts to be exploited during the execution of the specification/validation process.

In particular, the categories of requirements are detailed together with the steps for the analysis and structuring of the requirements. The set of UML constructs and diagrams considered in the project for the semi-formalization step of the methodology process are presented. The set of concepts described in this document represent a subset of the UML 2 concepts and diagrams described in the OMG UML 2 metamodel specification documents (that can be retrieved at http://www.omg.org/spec/UML/2.1.2/). The Controlled Natural language grammar is described together with some examples of constraints.
related to the ETCS domain. Finally, the steps related to the validation of the model via model checking techniques are presented.
2. Informal analysis phase

The first activity in the methodology is the informal analysis of the set of requirements (step M1 in Table 1). In this phase the requirements are first categorized on the basis of their characteristics; then some dependencies among them are established to structure the categorized requirement fragments. The steps for this informal categorization and analysis are:

M1.1 isolation of the fragments that identify a requirement unit of the requirements document;
M1.2 categorization of the informal requirement fragments;
M1.3 creation of the dependencies among the informal requirement fragments;
M1.4 analysis of the informal requirement fragments based on standard inspection-based software engineering in order to identify flaws such as, e.g., recursive definitions.

The final result of the informal analysis phase is a database of categorized requirement fragments. In the following we focus on the first three activities.

2.1. Requirement Fragment isolation

The requirements fragments isolation step (M1.1) can be decomposed in two substeps:

M1.1.1 Select the text describing the relevant requirements from the ETCS specification in Microsoft Word;
M1.1.2 Create the new requirement fragment in RequisitePro

These steps allow storing the set of requirements in the RequisitePro database, maintaining the link with the original specification document.

2.2. Requirement categorization

The requirements categorization (step M1.2) consists in choosing the category to classify the requirement from the taxonomy shown in Table 2 exploiting RequisitePro.
Table 2: requirements categories.

<table>
<thead>
<tr>
<th>Category</th>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glossary</td>
<td>GLS</td>
<td>The requirement defines a particular concept in ETCS domain</td>
</tr>
<tr>
<td>Architecture</td>
<td>ARC</td>
<td>The requirement introduces some system’s modules and describes how they interact</td>
</tr>
<tr>
<td>Functionality/behavioural</td>
<td>BEH</td>
<td>The requirement describes the steps a particular module perform or the states where the module can be</td>
</tr>
<tr>
<td>Communication</td>
<td>COM</td>
<td>The requirement describes the messages some modules exchange</td>
</tr>
<tr>
<td>Environmental</td>
<td>ENV</td>
<td>The requirement describes some constraints on the model</td>
</tr>
<tr>
<td>Scenario</td>
<td>SCE</td>
<td>The requirement describes a possible scenario of the ETCS</td>
</tr>
<tr>
<td>Property</td>
<td>PRP</td>
<td>The requirement describes an expected property of the ETCS</td>
</tr>
<tr>
<td>Annotation</td>
<td>NTE</td>
<td>Notes in the specifications</td>
</tr>
</tbody>
</table>

The categories are used to guide the formalization by suggesting the use of a particular language construct (i.e. UML elements or CNL constraints).

The result of this step is the set of categorized requirements in RequisitePro database.

2.2.1. Examples of requirements classification

In the following some examples of classification of ETCS specifications are given. In bold the parts of the text that can be categorized in a given category.

An example of glossary requirement fragments

3.1.1 Structure of a Movement Authority (MA)

3.1.1.2 For each section composing the MA the following information shall be given;

a) Length of the section

b) Section time-out value and distance from beginning of section to Section Time-out stop location

An example of annotation is given in the same subsection of the specifications

5.4.3.1 Note 2: The change of status of data in course of the procedure is shown in the table in section 5.4.4
2.3. **Requirements dependencies**

Three links have been identified to describe possible kinds of dependencies among the requirements fragments (step M1.3).

- **Strong Dependency** links: The requirement fragment “A” depends on the requirement fragment “B” if “A” cannot exist without “B”.

- **Weak Dependency** links: The requirement fragment “A” simply depends on the requirement fragment “B” but “A” can exist without “B”.

- **Refinement** links: The requirement fragment “A” refines the requirement fragment “B” if “A” redefines some notions of “B” at a lower level of abstraction.

RequisitePro that allows establishing the dependencies on the requirements supports this step.
3. Formalization phase

The next step is the requirements formalization (step M2 in Table 1) in the ETCS tool and in particular in the Rational Software Architect component.

The actions here are:

M2.1 Describe the requirements fragments exploiting UML and Controlled Natural Language formalizing them;

M2.2 Select a set of elements of the formalization;

M2.3 Link UML elements selected in M2.2 to the requirements fragments in RequisitePro; the link is used to trace the formalization, to ease the maintenance of the formalization after updates on the requisites, to select the requisite to check directly from the Word document.

Here in the following, we give a description of the fragment of UML and of the Controlled Natural Language used in the methodology. The constructs and concepts we use are:

- **Classes** and **class diagrams** are used to have a formalization of the requirements that have been classified as “glossary” requirements in the classification step of the methodology.
- **State machines** are used in the formalization of part of the “behavioural” constraints.
- **Sequence diagrams** to represent some kind of scenarios in the specifications.
- **Controlled Natural Language** (CNL) is used to specify the “environmental” requirements and the consistency constraints for the glossary and behavioural requirements.

The selection of UML diagrams and concepts has been performed on the basis of the expressive power of the UML concepts with respect to the needs of the ETCS specifications. In the rest of the section we describe the concepts of UML classes, state machines, sequence diagrams. Finally, the definition of the grammar of the Controlled Natural Language is presented.

### 3.1. UML class diagrams

Here we introduce the set of concepts, class and relationships to exploit in the specification of the Glossary requirements of the ETCS.

#### 3.1.1. Classes

A **Class** represents a concept in the ETCS domain:

- **Class attributes** (*attribute*) represent the set of characteristics of the concept;
• Class methods (*method*) represent an action the class can perform; a method accepts a set of parameters *par* in input and has a return parameter *par* ret.
  o A parameter *par* i has a type defined in the set \{integer, real, enumerate, class_type\}

The graphical representation of a class is given below

<table>
<thead>
<tr>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute: type</td>
</tr>
<tr>
<td>method(par i:type,par n:type): par ret type</td>
</tr>
</tbody>
</table>

### 3.1.2. Relationships

*Relationships* among classes represent the relation between concepts.

- **Association** is the basic relationship, it can be labelled via the roles of the two classes at its extremes and their cardinalities (the relative minimum and maximum numbers of instances of the two classes existing in the model); the relationship can also be named.

```
x..y role1 role2 l..m
    name
```

- **Aggregation**: an association in which one class belongs to a collection. An aggregation has a diamond end pointing to the class representing the whole;

```
   x..y l..m
  name
```

- **Generalization**: an inheritance link indicating one class is a superclass of the other. A generalization has a triangle pointing to the superclass;

```
  name
  superclass
```
• *Multiplicities* for the relationships represent constraints on the number of instances of the involved classes that can be created in the domain.

<table>
<thead>
<tr>
<th>Multiplicities</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..1</td>
<td>zero or one instance</td>
</tr>
<tr>
<td>0..* or *</td>
<td>no limit on the number of instances (including none)</td>
</tr>
<tr>
<td>1</td>
<td>exactly one instance</td>
</tr>
<tr>
<td>1..*</td>
<td>at least one instance</td>
</tr>
<tr>
<td>n..m</td>
<td>$n$ to $m$ instances</td>
</tr>
</tbody>
</table>

### 3.1.3. Examples of class diagrams in ETCS domain

In the following an example of formalization of a set of glossary requirements related to the on board subsystem and rbc via class diagrams.
Figure 1: an example of formalization of a set of glossary requirements.
3.2. **UML state machines**

A state machine is a graph in which the nodes represent states of a given system and the edges represent transitions between the states.

3.2.1. **State**

A state of the state machine is represented as a rectangle with smooth angles having:

- A **state name**
- An **entry condition** useful to specify an activity that can be executed once the flow enters in the state. The restrictions for the syntax of the activity are the same described in the next section for the activities related to the transitions.

<table>
<thead>
<tr>
<th>state name</th>
<th>entry/activity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Particular kinds of states are (see figure below):

- **initial state**, that is the first state of a state machine represented as a filled circle;
- **conditional state**, represented as a diamond, used to represent conditional branches in the state machine. The conditional state that allows splitting the transition in more than one branch, every one controlled by different conditions;
- **final state**, that is the last state of a state machine represented as a circle filled with a smaller one.
3.2.2. Transition

A transition of a state machine represents the passage between states and is represented as a labelled arrow. In figure below an excerpt of a two states state machine with two transitions is given. The source and the target states of a transition can be the same state as in the State_2 of the example.

![State Machine Diagram]

The label of the transition is structured as event[guard]/activity. The meaning of the label is that the transition is performed when the event occurs and the guard (that is a boolean predicate) is true; in this case the activity specified after the slash is performed.

A simple example of this label referred to the movement of a train can be:

- Event - pushed_button(); the event is the call of a class method; it will activate the transition
- Guard - [console.button=start]; it is a set of boolean predicates involving the variables in the model
- Activity - train.engineStart(); it is a method of the classes in the model or a set of simple assignments to the variables in the model

The flow between the two states of the example is: the flow is in the State_1. “When the event that labels the transition is activated and the guard is true, the transition is performed together with its associated activity and the flow enters in the State_2”.

Specification of activities

The activities specified in the transitions or in the entry clause internal to the state are not forced to terminate before the termination of the transition or the exiting from the state. In order to force the “wait for termination” behaviour, a special notation is used. In particular, in this case the method name has a “!” as a prefix. In this case a possible expression involving a method is “!method(par1,…parn)”. 
3.2.3. Examples

Example 1

In figure below is shown an example of how guards can be used to select the behaviour of the train. The train is in the state named wait. Once a button is pushed, an event, \((\text{pushed\_button()})\) and the train button that has been pushed is the stop button, \([\text{console.button=stop}]\) than nothing is performed, otherwise, if the train button that has been pushed is start than the engine start \((\text{train.engineStart()})\) and the flow moves to the state Run.

![State diagram](image)

Example 2

Consider the following example that would like to point out the use of a conditional state. The on-board shall reject \((\text{mess\_rejected})\) a message received \((\text{mess\_received})\) by radio and when variables in the messages have invalid values; it has to replay \((\text{reply})\) the message in the case it is valid.

In this state machine the conditional state (the diamond) is added to explicitly represent a condition.
decode()
[message.valid=false]
/message.display_reject()

mess_received

decode()
[message.valid=true]
/message.display_reply()
3.3. **UML Sequence diagrams**

UML sequence diagrams model the flow of logic, enabling to document and validate the interactions among objects, and are commonly used for both analysis and design purposes. Typically when designers produce interactions, the case is that the interactions do not tell the complete story. There are normally other legal and possible traces that are not contained within the described interactions.

In our context, we use the Sequence diagrams to model some “scenarios” requirements are requested to happen in the domain.

The most visible aspects of an interaction are the messages between the lifelines. The sequence of the messages is considered important for the understanding of the situation. The data that the messages convey and the lifelines store may also be very important, but the Interactions do not focus on the manipulation of data even though data can be used to decorate the diagrams.

3.3.1. *Basic Notation*

The basic notation for a sequence diagram is described in the following (see Figure 2). *Object* represents the instances of the classes involved in the interaction, *Lifeline* represents the object lifetime; in our context objects exist forever. *Execution specification* represents an interval in the lifeline in which the object is active. *Message* represents the call made by an object to a method of another object.

An example of a simple sequence diagram involving two objects is given in Figure 3; it describes a sequence of messages sent by an rbc through a channel of the On board subsystem.

![Figure 2: a generic sequence diagram.](image-url)
3.3.2. Interaction Operators

In the following some operators that can be used in the specification of a sequence diagram to specify particular configurations of messages are shown.

Negation operator

The operator neg designates that the represented traces are defined to be invalid.

The set of traces defined in a negative frame is equal to the set of traces given by its (sole) operand, but this set is a set of invalid rather than valid traces. All Interaction Fragments that are different from Negative are considered positive meaning that they describe traces that are valid and should be possible.

An example of the neg operand is given in the following diagram.
Figure 4: the negation of a behaviour.

In this case the occurrence of three “send message” in a strict sequence is forbidden via the constraint. This means that every trace having this configuration of messages have to be declared invalid.

**Alternative operator**

The Alternative (alt) operator is used to designate a mutually exclusive choice between two or more message sequences allowing the modelling of the classic "if then else" logic. The choices are specified via a set of guards.

An alt fragment element is drawn using a frame. The word "alt" is placed inside the frame's namebox. The larger rectangle is then divided into what UML 2 calls operands. A dashed line separates operands. Each operand is given a guard to test against, and this guard is placed towards the top left section of the operand on top of a lifeline. If an operand's guard equates to "true", then that operand is the operand to follow.
In Figure 5 an alternative operator is exploited. The sequence of messages starts with the first send. Then, if the guard in right parentheses is true a second message is send (message2), otherwise, the else guard, the first message is sent again.

**Option operator**

The option (opt) combination fragment is used to model a sequence that, given a certain condition, will occur; otherwise, the sequence does not occur. An option is used to model a simple "if then" statement. The condition is specified via a guard.
The option combination fragment notation is similar to the alternation combination fragment, except that it only has one operand and there never can be an "else" guard. To draw an option combination you draw a frame. The text "opt" is placed inside the frame's namebox, and in the frame's content area the option's guard is placed towards the top left corner on top of a lifeline. Then the option's sequence of messages is placed in the remainder of the frame's content area.

![Diagram of the optional operator](image)

**Figure 6: the optional operator.**

In Figure 6 the optional operator is used for choosing the occurrence of a message. If the guard is true, the send message in the optional frame is executed.

**Parallel operator**

The parallel operator (par) in Figure 7 designates a parallel merge between the behaviours of the operands. The messages of the different operands can be interleaved in any way as long as the ordering imposed by each operand as such is preserved. A parallel merge defines a set of traces that describes all the ways that messages in the operands may be interleaved without obstructing the order of the messages within the operand.
Figure 7: parallel operator in a sequence diagram.

Loop operator

The loop operand will be repeated a number of times specified via a guard.

The guard may include a lower and an upper number of iterations of the loop as well as a Boolean expression. The semantics is such that a loop will iterate minimum the 'minint' number of times (given by the iteration expression in the guard) and at most the 'maxint' number of times. After the minimum number of iterations has executed and the Boolean expression is false the loop will terminate. The loop construct represents a recursive
application of the seq operator where the loop operand is sequenced after the result of earlier iterations.

The textual syntax of the guard for the loop operand is:

\[
\text{loop}([\text{minint} \ [\text{maxint}]])
\]

\(<\text{minint}> ::= \text{non-negative natural}\)

\(<\text{maxint}> ::= \text{non-negative natural} \mid '*'

where:

\(<\text{minint}> \leq <\text{maxint}>

If only <minint> is present, this means that <minint> = <maxint> = \langle\text{integer}\rangle

'*' means infinity

If only loop, then this means a loop with infinity upper bound and with 0 as lower bound.

Figure 8: the loop operator; the send message is here repeated 3 times.

In Figure 8 the loop construct is exploited to produce the same behaviour of the SD in Figure 3. Also here three messages, as specified by the guard [1,3], are sent in sequence.
3.4. Controlled Natural Language

In the following the grammar for the Controlled Natural Language (CNL) for ETCS extracted from the definition of the general Controlled language grammar presented in [Ton07]. This language allows expressing constraints on the elements of the models.

In particular, the grammar defines 5 types of constructs: INVAR, defines a constraint that is always valid; INIT, defines a constraint that is valid at the initial stage; BEHAVIOR, constraints that can be used to express the admissible behaviour; SCENARIO, is not a constraint but a possible behaviour that we would like to have or to not have; PROPERTY, is a property that every possible admissible behaviour has to satisfy.

Constraints := `"INIT" basic_expr |
               `"INVAR" basic_expr |
               `"BEHAVIOR" temporal_expr |
               `"SCENARIO" temporal_expr |
               `"PROPERTY" temporal_expr ;

temporal_expr := basic_expr |
                 `"not" temporal_expr |
                 temporal_expr `"and" temporal_expr |
                 temporal_expr `"or" temporal_expr |
                 temporal_expr `"implies" temporal_expr |
                 `"always" temporal_expr |
                 `"never" temporal_expr |
                 `"in the future" temporal_expr |
                 temporal_expr `"until" temporal_expr |
                 temporal_expr `"infinitely many times" |
                 temporal_expr `"will eventually hold" |
                 `"every time" temporal_expr `"holds", temporal_expr |
                 `"sequence matching {" regular_expr "}" |
                 quantifier_prefix temporal_expr |
                 `"(" temporal_expr `")"
;

regular_expr := basic_expr |
                 `"emptyword" |
                 regular_expr `"," regular_expr |
                 `"{" regular_expr "}" or `{" regular_expr "}" |
                 `"{" regular_expr "}" and `{" regular_expr "}" |
                 `"{" regular_expr "}[*]" |
                 `"{" regular_expr "}[*" constant "]" |
                 `"{" regular_expr "}[*" constant ".." constant "]" |
                 `"{" temporal_expr "}" |

basic_expr := guard |
              `"if" guard `"then" actionguard |
              `"if" guard `"then" actionguard `"else" actionguard |
              `"not" basic_expr |
              basic_expr `"and" basic_expr |
              basic_expr `"or" basic_expr |
              basic_expr `"implies" basic_expr |
              quantifier_prefix basic_expr |
              `"(" basic_expr `")"
;

guard := term_expr `"<" term_expr |
         term_expr `">" term_expr |
         term_expr `"<=" term_expr |
         term_expr `">=" term_expr |
Formal Specifications of ETCS Functions

```
``der('' term_expr '')=' term_expr |
``der('' term_expr '')<'' term_expr |
``der('' term_expr '')>' term_expr |
``der('' term_expr '')<='' term_expr |
``der('' term_expr '')>'=' term_expr |
``not'' guard |
term_expr ``.' method `(){}` |
term_expr ```'' method ```' (`` parameter_list `')'' |
term_expr ```'' method ```' ('' returns'' term_expr |
``not'' guard |
guard ``` and'' guard |
guard ``` or'' guard |
guard ``` implies'' guard |
```' (`` guard``)'' ;
actionguard := action |
``term_expr ```='' term_expr |
term_expr ```<'' term_expr |
term_expr ```>'' term_expr |
term_expr ```<='' term_expr |
term_expr ```>='' term_expr |
``der('' term_expr '')='' term_expr |
``der('' term_expr '')<'' term_expr |
``der('' term_expr '')>' term_expr |
``der('' term_expr '')<='' term_expr |
``der('' term_expr '')>'=' term_expr |
``not'' guard |
actionguard ``` and'' actionguard |
actionguard ``` or'' actionguard |
actionguard ``` implies'' actionguard |
```' (`` actionguard``)'' ;
action := identifier_expr ```='' term_expr |
identifier_expr ```.' method `(){}` |
identifier_expr ```.' method ```' (`` parameter_list `')'' |
identifier_expr ```='' identifier_expr ```.' method `(){}` |
identifier_expr ```='' identifier_expr ```.' method ```' (`` parameter_list `')''

term_expr := identifier_expr |
constant |
term_expr + term_expr |
term_expr - term_expr |
constant * term_expr ;
parameter_list := identifier_expr |
parameter_list ```'' identifier_expr ;
quantifier_prefix := ```for all'' class_name variable |
```there exists'' class_name variable ```' such that' |
```for all'' variable ```' in'' identifier_expr |
```there exists'' variable ```' in'' range_expr ```' such that' |
```for all'' variable ```' in'' range_expr |
```there exists'' variable ```' in'' range_expr ```' such that' ;
identifier_expr := object |
variable |
identifier_expr ```.' attribute |
identifier_expr ```.' association_role |
identifier_expr ```[`` term_expr ```]'' |
identifier_expr ```.' size'' |
identifier_expr ```.' method ```' state'' ;
range_expr := constant ```'' constant |
constant ```'' identifier_expr ```.' size'' ;
```
3.4.1. Semantics of the temporal operators in CNL

In the following is described the semantics of the temporal operators defined in the CNL grammar; a pictorial view is given in Figure 9.

**Always**
Given a property \( \varphi \), “always \( \varphi \)” is true at a given time point \( t_0 \) if the property \( \varphi \) is true in all the time points \( t' \) such that \( t' \geq t_0 \).

**Never**
Given a property \( \varphi \), “never \( \varphi \)” is true at a given time point \( t_0 \) if the property \( \varphi \) is not true in all the time points \( t' \) such that \( t' \geq t_0 \).

**In the future (or will eventually hold)**
Given a property \( \varphi \), “in the future \( \varphi \) (will eventually hold \( \psi \))” is true at a given time point \( t_0 \) if the property \( \varphi \) is true in some time point \( t' \) such that \( t' \geq t_0 \).

**Until**
Given two properties \( \varphi \) and \( \psi \), “\( \varphi \) until \( \psi \)” is true at a given time point \( t_0 \) if:
- the property \( \psi \) is true in some time point \( t' \geq t_0 \);
- the property \( \varphi \) is true in all the time points \( t'' \) such that \( t_0 \leq t'' < t' \)

**Infinitely many times**
This operator can be viewed as a composition of the operators “always” and “in the future”. So, “Infinitely many times \( \varphi \)” is true at time \( t_0 \), if \( \varphi \) is true in infinitely many time points after \( t_0 \).

**Every time …, holds …**
Given two properties \( \varphi \) and \( \psi \), “every time \( \varphi \), holds \( \psi \)” is true at a given time point \( t_0 \) if, for every time point \( t' \geq t_0 \), the property \( \varphi \) is true than \( \psi \) is also true.
3.4.2. Examples of use of CNL in ETCS domain

Some examples are given of the use of the constraint languages in order to describe some typical constraints in the ETCS specifications.

Example 1
Change of value for a generic class attribute. It describes a general behaviour of the attributes in the model.

**BEHAVIOR**

always attribute=0 implies in the future attribute=1

Example 2
A message sent into a channel (e.g. a Movement Authority request) can be lost. It describes the representation of a scenario.
**SCENARIO**

*there exists* message \( m \)

*(in the future)*

\( (\text{send}(m) \text{ and never receive}(m))) \)

**Example 3**

A Balise Group is linked if its linking information has been received by the onboard subsystem

**BEHAVIOR**

*for all* Balise\_Group \( bg \) *(always ((there exists On\_Board\_Sub\_System s such that s.receive\_linking\_information(bg.linking\_info)) implies always bg.linked=\text{true})))*

**Example 4**

It is possible that the message \( m \) is received by the onboard subsystem \( s \) only after being sent three times by the RBC \( r \)

**SCENARIO**

*sequence matching*

\{ 

\{  

\( r.\text{send}(m) \text{ and not s.receive}(m); \)

\{\text{not s.receive}(m)\}[^*]  

\}[^*3];  

s.receive(m) 

\}

**Example 5**

Two trains can never be in the same position. It is a general property of the model.

**PROPERTY**

*never* (\( \text{train1.position} = \text{train2.position} \))
4. Formal Validation phase

The validation of the formalized requirement fragments (step M3 in Table 1) aims at improving the quality of the requirements. This goal is achieved by performing several formal analysis steps that range from the identification of inconsistencies (that may be not trivial to detect in an informal setting), to checks aiming at verifying that the categorized requirement fragment and its corresponding formalized requirement fragment meet the design intent, for instance by verifying that desired (undesired, respectively) behaviors have not (have) been ruled out by over-constraining (under-constraining) the requirements. The formal validation phase of the methodology will be accomplished as follows:

M3.3 Check the well-formedness of formalized requirement fragment. This initial activity aims at verifying that the formalized requirement fragment syntactically adheres to the formal language syntax, and that all the elements mentioned have been previously defined. Each syntactic error or undefined symbol identified in this preliminary validation phase requires a revision of the formalized requirement fragment to remove the flaw.

M3.2 Narrowing of the formalized requirement fragment. This phase aims at focus the validation to a particular subset of interest of the formalized requirement fragment (e.g. to restrict the validation of the classes/functions of a specific module).

M3.3 Formal validation of the identified formalized requirement fragment. The subset of interest identified in the previous phase is formally analyzed to identify flaws if any.

Whenever a problem is identified in any of the above sub-phases, in order to try and solve the identified flaw, it may be required going back to a previous phase. We remark that, in this phase, the railway expert responsible of the validation can specify additional desired and undesired behaviors w.r.t. those already formalized in previous phases as to guarantee that the design intents are captured, thus further enriching the formalized requirement fragment.

4.1. Validation

The formal validation phase (step M3.3) of the formalized requirement fragment can be further decomposed depending on the scope and on the level of domain knowledge required to perform it. For this purpose, we classify the validation checks in domain independent and domain dependent checks. There is a third kind of checks aiming at further analyzing the quality of the results produced by the domain dependent checks e.g. by performing vacuity analysis, coverage analysis and safety analysis.
Domain Independent Checks These checks aim at verifying properties of the formalized requirement fragment that do not require any domain knowledge i.e. logical consistency and realizability.

Consistency checking. The formal notion of logical consistency can be intuitively explained as “freedom from contradictions”. It is possible for the considered subset of the formalized requirement fragment to be inconsistent since two formalized requirement fragments mandate mutually incompatible behaviours (i.e., traces of evolution of the set of objects in the domain); this means that given a certain condition requirement $A$ says that it is possible to observe a trace $X$ in a given set of traces $T$, the requirement $B$ says that it is possible to observe a trace $Y$ in a set of traces $T'$ and $T$ and $T'$ are disjoint sets. This check aims at formally verify the absence of logical contradictions in the considered formalized requirement fragment. Consistency checking is carried out by dedicated state of the art formal verification algorithms [Cim07]. As outcome of the verification it is possible to produce a witness of the consistency as scenario compatible with the considered formalized requirement fragments. While, if the specification is inconsistent, we notice that no behavior can be associated to the considered formalized requirement fragments. In this case diagnostic information can be generated to be shown to the domain expert to try to fix the problem. In particular, the domain expert is given a subset of inconsistent formalized requirement fragments. These formalized requirement fragments can be then traced back to the categorized requirement fragments up to the original requirements in order to remove the inconsistency.

Realizability. Considering an open system$^1$, that in our context, is a system having the set of inputs controlled by the environment$^2$, realizability [Pne89,Chu63] intuitively amounts to checking if there exists an open system implementing the considered formalized requirement fragments. The feasibility study showed that the formalization of the subset of ETCS considered in this project requires using temporal constructs that make the realizability check unfeasible for the current technology.

Domain Dependent These checks aim at verifying that the considered set of formalized requirement fragments really captures the design intent. In this case the formalized requirement fragments are validated against desired and undesired behaviors to guarantee that no desired behaviors has been rule out (scenario compatibility, and, dually, to verify that no undesired behavior is allowed by the considered formalized requirement fragments.

---

$^1$ This kind of systems can be also called reactive systems.

$^2$ The name open system is in opposition to the concept of closed systems, that have no external inputs and that internally model both the automaton describing the behaviour of the system and that of the environment.
fragments (property checking).

**Scenario compatibility.** This check aims at verifying whether a set of conditions (also called a scenario) is possible, given the constraints imposed by the considered formalized requirement fragments. Intuitively, the check for scenario compatibility can be seen as a form of simulation guided by a set of constraints. The behaviors used in this phase can be partial as to describe a wide class of compatible behaviours (e.g., it is possible to specify that in the future there is a “wait for a signal” without specifying in details the trace before the signal). If the scenario is compatible, then as witness a behavior trace compatible with the formalized requirement fragments and with the specified scenario is produced and can be shown to the domain expert to analyze it. Otherwise, a subset of the considered formalized requirement fragments preventing the scenario to happen is identified and given to the domain expert as diagnosis information. A possible explanation for the prevention of the scenario to happen can be traced back to a misinterpretation in the formalization of the categorized requirement fragments into the corresponding formal version. In this case the affected formalized requirement fragments need to be revised to remove the ambiguity that led to the misinterpretation of the original requirements. The same verification techniques used for consistency checking are used here to perform the verification. In fact, the check for scenario compatibility can be re-conducted to problem of checking the consistency of the set of considered formalized requirement fragments with the constraint describing the scenario. In order to perform this kind of check, the analyst should introduce in the domain under analysis the constraint describing the scenario, using the same CNL syntax used to specify the constraints in the model, or via the specification of a set of sequence diagrams.

**Property checking.** This check aims at verifying whether an expected property is implied by the considered formalized requirement fragments. This check is similar in spirit to Model Checking [Clk99] where a property is checked against a model. Here the considered set of formalized requirement fragment plays the role of the model against which the property must be verified. When the property is not implied by the specification, a counterexample is produced. A counterexample is a behavior witnessing the violation of the property, i.e. a trace that is compatible with the considered formalized requirement fragment, but does not satisfy the property being analyzed. If the verification of the property fails, two causes are possible: the first one is that the property is not correctly formalized; the second possibility is in a wrong formalization of the informal sentences in the categorized requirement fragment that need to be disambiguated and/or corrected. An inspection of the counterexample can be carried out in order to discriminate between the two possibilities. If the property is wrong, then it is corrected and the check is repeated. Otherwise, the formalized requirement fragments have to be corrected, either by modifying the formalization or by adding additional constraints, until the satisfaction of the given property is achieved. The same verification techniques used for consistency checking and for scenario compatibility are used here to perform the verification. In fact, the check for property checking can be re-conducted to problem of checking the consistency of the set
of considered formalized requirement fragments with the negation of the property. If we find that this set is consistent, then a witness is generated. This witness is compatible with the considered formalized requirement fragment and satisfy the negation of the property, thus it is a counterexample for the property. Similarly to scenario compatibility, to perform this kind of check, the analyst can introduce in the domain under analysis a set of constraints describing the properties that should be validated, using the CNL syntax.

**Analysis of the quality of verification results** The previous analysis can produce diagnostic information in several forms:

**Traces.** For consistency checking, scenario compatibility and property checking a behavior in form of an execution trace may be generated. For consistency checking, the trace represents a witness of the consistency of the requirement fragments. For scenario compatibility, the trace represents a witness of the compatibility of the scenario with the requirement fragments. For property checking, the trace represents a counterexample showing a violation of the property.

**Unsat cores and vacuity checking.** For consistency checking, scenario compatibility and property checking a subset of requirement fragments may be generated. For consistency checking, the subset represents a core of inconsistent requirement fragments. For scenario compatibility, the subset represents a core of requirement fragments incompatible with the scenario. For property checking, the subset shows that the property is vacuously satisfied, because the subset is sufficient to prove the property and the requirement fragments not in the subset do play a role in the proof.

**Formal safety analysis.** It aims to identify all the causes leading to the violation of an expected property. The subset of the ETCS chosen for the feasibility study did not show any failure model, and thus it was not possible to enable fault-tree analysis.

**Validation Loop** The above described validation phases are the building blocks for complex validation activities. For instance, the narrowing phase allows the domain experts to focus only on a subset of the formalized requirement fragments by selecting specific modules and consider only some of the functions of the selected module thus enabling for a modular validation approach. Several kind of what-if analysis can be performed playing with the subset of the formalized requirement fragments selected in the narrowing phase. We can de-select a previously selected formalized requirement fragment and check whether the resulting set of formalized requirement fragments still fulfills all the properties. If this is the case, then the de-selected formalized requirement fragment was not necessary for the fulfillment of the given properties or of a subset of them. Similarly, we can check whether the selection of additional formalized requirement fragments to be added to the previous set of formalized requirement fragments is such that all the selected scenarios are still admitted by the resulting set of formalized requirement fragments. We
can also replace one or a set of formalized requirement fragments and check whether all the selected scenarios are still admitted by the resulting formalized requirement fragments, and all the properties are still fulfilled. Moreover, in the narrowing phase we can ignore the requirements with low-level details and consider the requirements at a higher level of abstraction thus enabling for a hierarchical verification approach. The validation steps can be iterated arbitrarily by correcting formalized requirement fragments if necessary, creating new scenarios, new properties, performing several what-if analysis. Every check will increase the confidence of the domain expert in the correctness of the formalized requirement fragments. When the expert is satisfied by the validation of the selected set of formalized requirement fragments, the final formalized requirement fragments plus the added properties and scenarios can be added into a database in order to be re-used in the subsequent validation processes.

4.2. An example of validation phase

In the following is described a possible process a user can adopt in order to build the problem and validate a model after its specification.

• User choose a set of requirements in RequisitePro

• The ETCS tool highlights a set of conflicts and the dependencies between requirements defined at the specification level

• Once dependencies and conflicts are solved, user can open the RSA view of the formalization components

• User may refine the selection in RSA where the elements that can be selected are: classes, methods, associations, state machines, sequence diagrams and constraints

• The concepts and artifacts previously selected may have conflicts or dependencies; here the user can refine the selection in order to make it compatible with dependencies and conflicts at the RequisitePro level

• The user can instantiate the set of objects in a given “real” domain via the definition of a new class, having as attributes the set of domain objects. The user may add a set of new constraints on the attributes of this class, and/or a set of new sequence
diagrams.

• User can refine the selection of the model at the formal specification level, such as, he can modify constants of the model, select and deselect temporal formulas induced by state machines and sequence diagrams

• The window for the selection highlights the kind of requirement in the formal model
  - R behaviours/specification (constraints, state machine)
  - P scenarios/possibilities (constraints, sequence diagrams)
  - A properties/assertions (constraints)

• User can drive the selection on the bases of the preferred check:
  1. R a consistency check
  2. R and a subset of P (scenario check)
  3. R and a property in A (property check)

• The tool transforms the UML/CNL specification and uses NuSMV to fined a model

• If a model is found, it is shown as a scenario (trace plus sequence diagram) to the user (Note: in 1) e 2) the scenario is shown as a witness, in 3) as a counterexample)

• In the case a model is not found, the check is repeated to find a subset of R (Note: in 1) and 2) the subset is presented as “unsat core”, in 3) as vacuity/coverage feedback)

4.3. **Example of problem definition**

Here we give some guidelines related to the definition of a problem by narrowing the defined requirements fragments (step M3.2). In particular, we describe the definition of the “main” class that allows describing the entities in play and an example of specification of a generalized test scenario.

4.3.1. **Definition of main class**

The set of entities involved in the verification problem can be specified via the generation of a new class (in the following named *main* class). In particular, the attributes of the class represent the instances of the elements in play we are instantiating the model. Moreover, it is possible to specify some CNL constraints on the attributes of the main class that allows describing constraints in the class.
Referring to the attributes of the class main in the example of Figure 10 we have: 1..* Movement authorities, an On Board, a Train, an End of Authority, 1..* Sections, a number of Danger points and Overlaps.

![Class diagram](image)

**Figure 10: The class main defining a verification problem in the ETCS domain.**

4.3.2. **Definition of a generalized test scenario via CNL**

Here we give an example of how it is possible to specify a scenario in which a sequence of actions has to be performed by the elements in play. In particular, the scenario is specified via CNL clauses that allow expressing some constraints on the behaviour of the entities.

The hypotheses is that of having two classes:
- OnBoard in which the method `send_ma_request_rbc()` is defined
- RBC in which the method `receive_ma_request_onboardss()` is specified,

and having defined a problem via the “main” class in Figure 10 in which two attributes have been defined:
- onboard of type OnBoard
- rbc of type RBC

Exploiting the CNL grammar it is possible to write the following generalized test scenario:

```
SCENARIO a sequence matching
{
    onboard.send_ma_request_rbc();
    {true}[*];
    rbc.receive_ma_request_onboardss()
}
```

that describes a sequence of events in which:
1. an *onboard* sends a movement authority request to *rbc* (the method `send_ma_request_rbc()`)
2. after some time described via the clause `{true|*}`,
3. the *rbc* receives the movement authority request (the method `receive_ma_request_onboardss()`)

This Scenario can be associated to the “main” class that defines the problem to be checked.
5. An example of the formalization and problem definition process

In the following a short but complete example of the use of the methodology is given. Here we focus on some requirements related to the Movement Authority and RBC concepts.

5.1. Classification of requirements

Example Requirement 3.8.1.1:

The following characteristics can be used in a Movement Authority:

a) The End Of Authority (EOA) is the location to which the train is authorized to move.

   (Glossary requirement: in this requirement there is an explanation about the End of Authority and it is possible to understand that “EOA” will be translated in a class).

b) The Target Speed at the EOA is the permitted speed at the EOA;

   (Glossary/Environmental requirement: definition of Target Speed, it is possible to understand that “Target Speed” will be an attribute of the class EOA).

   when the target speed is not zero, the EOA is called Limit Of Authority (LOA).

   (Glossary/Environmental requirement: in this requirement there are two information about the translation: the former is the definition of the Limit of Authority; the latter describes when this definition is valid, this will be translated in a constraint).

   This Target Speed can be time limited.

   (Property requirement: this requirement defines a constraint on the attribute Target Speed; it is possible to understand that is necessary to provide a timer that defines the validity of the Target Speed).

   ...

   e) A release speed is a speed under which the train is allowed to run in the vicinity of the EOA,

   (Glossary/Environmental requirement: this is a definition of the “Release Speed” as an attribute of EOA, in addition this requirement defines a constraint that bounds the speed of the train close to EOA).

   when the target speed is zero.

   (Environmental requirement: defines that at the EOA the Target Speed must be zero, it will be translated as a constraint).

   One release speed can be associated with the Danger Point, and another one with the overlap.

   (Glossary requirement: it defines the necessary to have the attribute “Release Speed” also in the classes “Danger Point” and “Overlap”).

   Release speed can also be calculated on-board the train.
(Behavioural requirement: this requirement defines that the attribute “Release Speed” can be calculated also on board. It means that there will be a method called “calcReleaseSpeed” in the class “OnBoard”).

f) The MA can be split into several sections, The last one is called End Section.  
(Glossary requirement: defines that the MA is composed of one or more sections, and the last one is called “End Section”. It will be translated into a class and an association).

**Example Requirement 3.8.6.1:**

It shall be possible to shortening a given MA using a special procedure between on-board equipment and RBC. The procedure is as follows:

a) RBC requests (proposes) a new end of authority together optionally with danger point and overlap information at a location closer to the train than the current End of Authority;

b) The ERTMS/ETCS on-board equipment checks if the train can be stopped without brake intervention at the requested location:
   a. If this is possible, the on-board equipment shall accept the new End of Authority;
   b. If not possible the request is rejected and the previously received MA remains valid;

c) The RBC shall be informed about the decision.

(Scenario requirement: in this requirement is described a particular ETCS scenario, it will be translated in a state machines and/or in a sequence diagram. Moreover, the Class Diagram for this requirement will be composed of 2 classes: “RBC” and “ErtmsEtcsonBoardEquipment”)

5.2. **Creation of the Class Diagram**

Considering the requirements analyzed in the previous step it is possible to create the class diagram in Figure 11.
**Figure 11: class diagram of the requirement 3.8.1.1.**

**Discussion**

In this step are described the set of classes related to the concepts identified in the previous step and the relationship between these classes. For example, the requisite 3.8.1.1/f points out that the movement authority is divided in sections; this statement is translated in the definition of the two classes *MovementAuthority* and *Section* and in the definition of a relation between the class *Section* and the class *MovementAuthority*, with a multiplicity of 1..n for the former and 1 for the latter. Considering the glossary definition of the Danger Point and the Overlap is possible to post a relationship between these classes and the class section. This translation is done with multiplicity of 1 for the class “Section” and n for the classes “DangerPoint” and “Overlap” because they are a specifics points/parts of a section of the railway with certain specific property.
5.3. Creation of the constraints in CNL

The environment requirements, detected in the specification, describing the constraints on the model can be translated in constraints expressed in CNL associated to a specific class.

Here in the following some examples.

1) Requirement 3.8.1.1/b: "when the target speed is not zero, the EOA is called Limit Of Authority (LOA)."

Derived constraint:

1b) CNL on class EndOfAuthority: "INVAR: targetSpeed > 0 implies limitOfAuthority;".

2) Requirements 3.8.1.1/e: "A release speed is a speed under which the train is allowed to run in the vicinity of the EOA,"

Derived constraints:

2b) CNL on class Train: "INVAR: ((onboard.MovementAuthority.EndOfAuthority.position > position) and (onboard.MovementAuthority.EndOfAuthority.position - position <= onboard.MovementAuthority.EndOfAuthority.admittedPosition) implies der(position) <= onboard.MovementAuthority.EndOfAuthority.releaseSpeed); ";

2c) CNL on class Train: "INVAR: ((onboard.MovementAuthority.EndOfAuthority.position < position) and (position - onboard.MovementAuthority.EndOfAuthority.position <= onboard.MovementAuthority.EndOfAuthority.admittedPosition) implies der(position) <= (0 - onboard.MovementAuthority.EndOfAuthority.releaseSpeed));".

Discussion

This requirement is translated in two CNL constraints in order to distinguish two possible situations: the first one is that the train is going through the line in nominal way (the EOA has a position greater than the train one) and the second one is that the train is going through the line in reverse way (the EOA has a position smaller than the train one). The attribute “admittedPosition” in the class EndOfAuthority defines the limit under which the Release Position is valid. In these examples there are a particular expression of the CNL language, defined as “der” that provides the temporal derivation operator; in this case this expression is used to manage the speed of the train. Moreover, the prefix “INVAR” explains that the follows constraint must be always true.

3) Requirements 3.8.1.1/e: "...when the target speed is zero"

Derived constraints:

3a) CNL on class Train: "INVAR: ((onboard.MovementAuthority.EndOfAuthority.position = position) and (onboard.MovementAuthority.EndOfAuthority.limitOfAuthority = FALSE) implies der(position) = 0);";
5.4. Creation of the state machines

Here an example of requirements translated in state machines is given. In particular, we analyze part of the requirement 3.8.6.1 proposing a possible translation using state machines and CNL constraints.

**Example requirement 3.8.6.1:**

It shall be possible to shortening a given MA using a special procedure between on-board equipment and RBC. The procedure is as follows:

d) RBC requests (proposes) a new end of authority together optionally with danger point and overlap information at a location closer to the train than the current End of Authority;

e) The ERTMS/ETCS on-board equipment checks if the train can be stopped without brake intervention at the requested location:
   a. If this is possible, the on-board equipment shall accept the new End of Authority;
   b. If not possible the request is rejected and the previously received MA remains valid;

f) The RBC shall be informed about the decision.

(Scenario requirement: this requirement describes a particular procedure of an ETCS scenario, it will be translated in a state machine and/or in a sequence diagram. In the example we refer to a Class Diagram composed of two classes: "RBC" and Ertms – Etc s "OnBoardEquipment")

![Class Diagram](image)

**Figure 12:** RBC and OnBoardEquipment for requirements 3.8.6.1.

**CNL on class RBC:** "BEHAVIOR: for all MA m (always if SendProposeMA( m ) then in the future onboardequipment.ReceiveProposeMA() returns m);";

**CNL on class OnBoardEquipment:** "BEHAVIOR: for all MA m (always if SendRejectMA( m ) then in the future rbc.ReceiveMARejected() returns m);";

**CNL on class OnBoardEquipment:** "BEHAVIOR: for all MA m (always if SendConfirmMA( m ) then in the future rbc.ReceiveMAConfirmed() returns m);";

The state machines describes the behavior of the methods, and in this case will be created only the state machines for the methods “ReceiveProposeMA” Figure 13 and “CheckMA” Figure 14 of the class “OnBoardEquipment”.
Figure 13: State machine of the method ReceiveProposeMA.

Figure 14: State machine of the method CheckMA (with a determinism way).

Discussion
In a state machine it is possible to have:
- **States**: that describe a state of the method. Here is possible to have a single action at the time of entry in the state. The semantics is described in the CNL documentation.
Moreover, there are two particular states: the initial state and the final state where are not possible to have actions;

- **Transitions**: that are links between states. These operators describe the possibility to move from a state to another one. Furthermore a transition has a guard (that defines when the transition is enabled) and a possibly action (that starts when the transition is enabled). If two or more transitions that start from a state are enabled at the same time there will be a deterministic selection.

### 5.5. Creation of the sequence diagrams

In order to explain the sequence of some actions it is possible to describe it via a sequence diagram.

**Example requirement 3.8.6.1:**

![Sequence diagram for requirement 3.8.6.1.](image)

**Figure 15:** Sequence diagram for requirement 3.8.6.1.
Discussion.
In a sequence diagram is described the temporal ordering of the events, where the temporal line starts from the top and ends at the bottom. In our framework, a sequence diagram is composed of:
- **Class instances and attributes**: the objects that communicate in a possible scenario, they are described as vertical lines. Moreover, there is the object called “*” that is a virtual caller of the send methods;
- **Method calls**: described as an arrow that starts from the caller and ends to the method called;
- **Time of computation**: described as squares on the vertical lines (class instances).
Moreover, there are some fragments that can describe different possible scenarios. In this case the fragment called “alt” distinguishes from the scenario when the Movement Authority is accepted and rather when is rejected.

5.6. Specification of the validation problem via the creation of a new class

The creation of the problem consists in adding a class called “main” in the class diagram with the aim to describe multiplicity of the objects in the model composing the problem exploiting the class attributes and some constraints exploiting CNL language. This step is explained in the following example.

**Example requirement 3.8.1.1:**

![Main class diagram](image)

**Figure 16: the main class to define the problem.**

Referring to the attributes of the class main in the example of Figure 16 we have: 1..* Movement authorities, an On Board, a Train, an End of Authority, 1..* Sections, a number of Danger points and Overlaps.
Associated to the main class some constraints can be specified to express invariant properties on the objects of the class.

**Relation between classes MovementAuthority – OnBoard (1;1):**

"INVAR: movementAuthority.onboard = onBoard;"
"onBoard.movementauthority = movementAuthority;";

**Relation between classes OnBoard – Train (1;1):**

"INVAR: onBoard.train = train;"
"INVAR: train.onboard = onBoard;";

**Relation between classes MovementAuthority – EndOfAuthority (1;1):**

"INVAR: movementAuthority.endofauthority = endOfAuthority;"
"INVAR: endOfAuthority.movementauthority = movementAuthority;";

**Relation between classes MovementAuthority – Section (1;1..*):**

"INVAR: section.movementauthority.size = movementAuthority.size;"
"INVAR: for all i in 1..movementAuthority.section.size(section.movementauthority[i] = movementAuthority.section[i]);";

**Relation between classes Section – DangerPoint (1;*):**

"INVAR: dangerPoint.section.size = section.size;"
"INVAR: for all i in 1..section.dangerpoint.size(dangerPoint.section[i] = section.dangerpoint[i]);";

**Relation between classes Section – Overlap (1;*):**

"INVAR: overlap.section.size = section.size;"
"INVAR: for all i in 1..section.overlap.size(overlap.section[i] = section.overlap[i]);";

Moreover, it is possible to specify *a message sequence scenario* to check, for example, that the message \( m \) is received by the onboard subsystem \( \text{onBoard} \) only after being sent three times by the RBC \( \text{rbc} \).

**SCENARIO**

*a sequence matching \{\text{rbc.send}(m) \text{ and not onBoard.receive}(m); \{\text{not s.receive}(m)\}_3; s.receive(m)\}*

The next step here is the choice and activation of the model checking functionalities of the tool.
Appendix A: methodology summary

M1 Informal analysis phase: it consists of the categorization and structuring of the informal requirement fragment described in the requirements document to produce categorized requirement fragments

M1.1 isolation of the fragments that identify a requirement unit of the requirements document;

M1.1.1 Select the text describing the relevant requirements from the ETCS specification in Microsoft Word;

M1.1.2 Create the new requirement fragment in RequisitePro

M1.2 categorization of the informal requirement fragments;

M1.3 creation of the dependencies among the informal requirement fragments;

M1.4 analysis of the informal requirement fragments based on standard inspection-based software engineering in order to identify flaws such as, e.g., recursive definitions.

M2 Formalization phase: the categorized requirement fragments are described through the set of concepts and diagrams in UML, and additional constraints in the defined CNL to produce formalized requirement fragments

M2.1 Describe the requirements fragments exploiting UML and Controlled Natural Language formalizing them;

M2.2 Select a set of elements of the formalization;

M2.3 Link UML elements selected in M2.2 to the requirements fragments in RequisitePro; the link is used to trace the formalization, to ease the maintenance of the formalization after updates on the requisites, to select the requisite to check directly from the Word document.

M3 Formal validation phase: it consists of the identification of a subset of the formalized requirement fragments (together with the definition of a series of validation problems) for an automatic validation analysis

M3.1 Check the well-formedness of formalized requirement fragment. This initial activity aims at verifying that the formalized requirement fragment syntactically adheres to the formal language syntax, and that all the elements mentioned have been previously defined. Each syntactic error or undefined symbol identified in this preliminary validation phase requires a revision of the formalized requirement fragment to remove the flaw.

M3.2 Narrowing of the formalized requirement fragment. This phase aims at focus the validation to a particular subset of interest of the formalized requirement fragment (e.g. to restrict the validation of the classes/functions of a specific module).

M3.3 Formal validation of the identified formalized requirement fragment. The subset of interest identified in the previous phase is formally analyzed to identify flaws if any.
References


