FP7 Project 2007- Grant agreement n°: 218575

Project Acronym: **INESS**

Project Title: **INtegrated European Signalling System**

Instrument: Large-scale integrating project
Thematic Priority: Transport

WS D – Generic requirements

Deliverable D.1.2 – Requirements expression document

Due date of deliverable	31-05-2009
Actual submission date	29-05-2009

Deliverable ID:	**D.D.1.2**
Deliverable Title:	Requirements expression document
WP related:	D1 – Generic requirements
Responsible partner:	ProRail
Task/Deliverable leader Name:	UIC
Contributors:	M. Blazič, C. de Courcey-Bayley

Start date of the project: 01-10-2008 Duration: 36 Months

Project coordinator: Paolo De Cicco
Project coordinator organisation: UIC

Revision: SB finalised Dissemination Level[1]: CO

[1] PU: Public, PP: Restricted to other programme participants (including the Commission Services), RE: Restricted to a group specified by the consortium (including the Commission Services), CO: Confidential, only for members of the consortium (including the Commission Services).

Document Information

Document type: Deliverable Report
Document Name: INESS_WSD_DeliverableD.D.1.2_ SB_Finalised_Report_Ver2009-05-29
Revision: SB_Finalised
Revision Date: 29-05-2009
Author: UIC
Dissemination level: **CO**

Approvals

	Name	Company	Date	Visa
WP leader	Khalid AGROU	UIC	2009-04-21	OK
WS Leader	Wendi Mennen	ProRail	2009-05-28	OK
Project Manager	Emmanuel Buseyne	UIC	2009-05-28	OK
Steering Board	-	-	2009-05-29	approved

Document history

Revision	Date	Modification	Author
1	2009-04-21		M. Blazič, C. de Courcey-Bayley
SB_Finalised	29-05-2009		Richard VAUX / ALMA

TABLE OF CONTENTS

GLOSSARY

DOORS Dynamic Object Oriented Requirements System

DOS Drive On Sight

GENERIS GENEric Requirements for Interlocking System

LCL Logical Concepts Layer

LSA Local Shunting Area

SELRED Structured English Language for REquirements Development

TVP Track Vacancy Proving section

UML Unified Modelling Language

xUML Executable UML

Section 1 – EXECUTIVE SUMMARY

This document is a guideline on the format and language for requirements expression. It explains how to express the functional requirements of an interlocking system and is divided into two main parts: the first focuses on textual expression of requirements, whereas the second explains how these requirements can be formally translated.

Section 2 – INTRODUCTION

This document corresponds to the requirements expression document of Workstream D, Task D.1.2 as mentioned in the INESS description of work. It gives the guidelines for requirements expression at a functional level for an interlocking system.

It relies on the work carried out during the forerunner Euro-Interlocking project, the latter being therefore the main source for the current deliverable.

The presented methodology encompasses two complementary ways of expressing the requirements: the first details how they can be expressed in natural, but nevertheless, stringent English using a specific environment. Furthermore, it shows how one can define a format for these requirements, and then build a structured database of requirements.

The second part is dedicated to formal expression of requirements. After an explanation on the added value of modelling, the reader is introduced to object oriented modeling in UML (Unified Modelling Language), and is given the overview of the application of UML: showing the different components needed to model a system in terms of class diagrams for the static description of the system, and state machines for the dynamic part of it, together with some syntactic considerations.

Given the complexity of an interlocking system, the approach chosen makes a study case of certain concepts of particular interest, emphasizing the various nuances developed in order to properly elicit and model a given functionality.

It is of paramount importance to understand that compared to natural English, modeling requirements is a complementary way of eliciting and defining precise functional requirements: while written requirements are the source material for the modeling, the latter can lead to a reformulation of the former when it raises particular questions that were not envisaged initially.

Section 3 – FORMAT AND LANGUAGE FOR REQUIREMENTS EXPRESSION

3.1 Functional requirements expression in text

Functional requirements for interlocking systems in the INESS project will be captured and developed in a standardised textual format. The notation will use natural English language, with the guidelines of the SELRED (Structured English Language for REquirements Development) strategy. The requirements will be developed and managed with the use of requirements managements tool Telelogic DOORS.

3.1.1 Environment for textual requirements expression

3.1.1.1 Telelogic DOORS

Telelogic DOORS (Dynamic Object Oriented Requirements System) is requirements management tool used by engineers in many sectors. It is designed to capture, link, trace, analyze and manage a wide range of information required to provide solutions in complex projects.

The requirements and related information are stored in a central database in DOORS. The database can be accessed in a variety of ways and exists throughout the lifetime of the application. The information in the database is stored in modules. These can be organized by using folders and projects.

DOORS folders are used to organize the modules in the database in the same way as folders are used to organize computer files.

DOORS projects are a special kind of folder that contains all the data for a particular project. It can contain all of the data related to the requirements, design, development, test, production and maintenance of an application. The project has capability to manage users and their access to the data, to back up the data and to distribute parts of the data to other DOORS databases of other users.

The information within each module is divided into objects and their attributes. An object may be a block of text, a graphic image or a table from another program. Each object has its own identifier, which does not change in the project lifetime.

DOORS maintains a historical log of all module and object actions that modify the contents of a module, its objects or attributes. Every change that is recorded includes the information about the user, the contents and the time of the change.

DOORS also provides baselining features. A baseline is a frozen copy of a module. These are typically created at a significant stage of a project. This allows the various states of the requirements document to be easily reproduced at any time. Baselines are read-only copies of a module. They include all the history since the previous baseline was created.

3.1.1.2 SELRED guideline

The elements of the SELRED (Structured English Language for REquirements Development) strategy are as follows:

High-level structure:
Clarity can be greatly improved by adopting a standard high-level structure for requirements definitions. By clearly distinguishing domain knowledge from system requirements, conditions from operations, one provides a consistent framework for the different sorts of requirements statements. By clearly delimiting the context of each statement, it greatly reduces the scope for ambiguous interpretation.

Good term definitions:
To provide a firm foundation for expressing requirements, it is essential to have a clear definition of the terms used. It is very important to use terms consistently and to avoid overlapping terms, such as having more than one name for the same thing.

Structuring of operations:
Just as the high-level structure provides a clear context for the different types of requirements, a consistent structure for defining operations focuses attention on the essential aspects of each operation. For example, clearly identifying inputs, outputs, preconditions and post conditions, it helps to ensure that the definition of each operation is complete and unambiguous.

Logic construction guidelines:
A more productive approach is to draw up guidelines, based on experience, to avoid the most common pitfalls. A set of guidelines for expressing logic constructions clearly in English would cover, for example, the use of the conjunction ‘AND', the use of the disjunction ‘OR' and advice on the degree of nesting acceptable.

Sentence construction guidelines:
Similarly, guidelines for the construction of clear English sentences are required. These would include issues such as using a reduced set of imperative verbs and minimising the number of subordinate clauses. The requirement sentence should only include allowed adverbs and adjectives.

Definition of verbs:
One of the most important construction elements of an unambiguous sentence is the verb. Correctly-used verbs ensure that the meaning of the sentence is understood in the same way as the writer intended.

3.1.2 The format of the requirements

The requirements must be written in an understandable, uniform and structured manner. Such format enables clarity and comparison of the requirements. Important format issues are clear structure, uniform syntax and the terminology used in developing the requirements.

3.1.2.1 Requirements structure

Each functional requirements document is represented as one module in DOORS. All functional requirement documents in the database contain requirements with the same structure. Such structure enables the capturing of national functional requirements for each country with minimal duplication and maximal reuse. The structure also makes it easy to change a requirement for a given country by adding or deleting conditions. In this way it is simple to add or delete a requirement for a country.

Requirements are written in accordance with basic templates as often as possible, unless a different structure is required to explain a certain complex requirement.

Basic templates include the following structures:

<*System*> shall be able <*action*>.

<*System*> shall contain the functionality <*action*> if/when <*operational condition*>.

<*System function*> shall <*action*> while <*operational condition*>.

<*System function*> shall remain <*state*> while <*operational condition*>.

<*System function*> shall <*action*> if <*operational condition*>.

<*System function*> shall be requested by <*request*>.

<*System function*> shall consist of <*action*>.

<*System function*> shall <*action*> if <*operational condition 1*> while <*operational condition 2*>.

When <*system function*> becomes <*state*>, <*element*> shall <*action*>.

<*Element*> shall become <*status*> if <*operational condition*>.

<*System function*> shall <*action*> while <*operational condition*>.

When <*element*> becomes <*state*>, <*element*> shall <*action*> unless <*operational condition*>.

The use of templates ensures similar structure throughout all documents, which results in short and consistent sentences.

Complex requirements, which consist of different conditions and relationships between them, have the following structure:

Requirement R1:

- *condition C1*
- *condition C2*
- *condition C3*
- …

The formulation of the text in the requirement R1 defines the nature of the relationship between the conditions C. These may be may be organized in a list, or logically connected with the use of AND/OR statements.

Where a relationship between the conditions is more varied, it is structured in a manner as described in the following examples:

a) Requirement R1:

- *condition C1*

- *OR*
- *condition C2*
- *AND*
- *condition C3*
- *OR*
- *condition C4*

The above example should be read as:

Requirement R1:

(condition C1 or condition C2) and (condition C3 or condition C4).

b) Requirement R1:

- *condition C1*
- *OR*
- *condition C2*
- *AND*
- *condition C3*

The above example should be read as:

Requirement R1:

(condition C1 or condition C2) and condition C3.

c) Requirement R1:

- *condition C1*
- *OR*
- *condition C2*
- *AND*
- *condition C3*

The above example should be read as:

Requirement R1:

condition C1 or (condition C2 and condition C3).

Care should be used to always indicate the priority of relationships with the use of indentations, not relying on the logical operators priority rule of the AND over OR.

Requirements, which indicate a list of objects, have the following structure:

Requirement R1:

- *listing L1*
- *listing L2*
- *listing L3*
- …

Each requirement and condition or listing is its own object in DOORS. A self-standing operator (AND, OR) must be structured as an individual object as well, and nested to its parent requirement. This enables setting an object's *"Applicable to"* attribute to show that a requirement or a related condition or listing is applicable to one or more railways. The *"Applicable to"* attribute must be set for the requirement object, the appropriate condition or listing, and to the AND/OR statement, thus enabling the proper selection or filtering of requirements for a particular railway.

3.1.2.2 Requirements syntax

To keep the database and all contents consistent, certain rules have been implemented for the syntax of requirements.

Syntax Rule	Example
All words in headings are capitalized.	2.1 Setting a Local Shunting Area
Bullet points are not capitalized and are in italics.	• *points* • *derailers*
Commands are described as requests and are listed in quotes.	A request 'Set local shunting area' has been received from the signaller.
Aspects are listed in quotes.	Setting a signal to the 'stop' aspect.
Functional statuses defined by these requirements are listed in quotes.	'trailed', 'occupied', 'blocked'
The lack of a state is described as *not 'state'*	*not 'occupied', not 'blocked'* *Unoccupied, unblocked* must not be used.

Table 1

3.1.2.3 Requirements terminology

The main guideline for expressing terms in requirements is to use already standardised terms. The use of terms which are not self-explanatory or commonly known and which can be misinterpreted in different contexts shall be omitted or clarified by defining the relevant concept in the glossary of terms.

The list of terms should not be exhaustive. The glossary shall only include necessary terms. All other terms used in the requirements shall be interpreted as they are defined in the Oxford Dictionary.

Abbreviated terms shall be used with care, and their use shall be limited to those cases where it is unlikely to cause confusion. An abbreviated term shall be specified only if used subsequently in the requirement. If a list of abbreviated terms is not given in the document, then the first time that an abbreviated term is used, the full term shall be given with the abbreviated term following in parentheses.

The verb "control" is to be avoided. "Supervise" is to be used instead of the verb "control" for the purpose to find out input states and "drive" is to be used instead of the verb "control" for purpose to steer an output.

3.1.3 The requirements database

The requirements database is structured to enable the capturing of functional requirements for interlocking systems in a common way for all participating railways. Therefore the functional requirements form the major part of the database. The basic principle behind the database is that each requirement is an individual object with its properties described in attributes. All requirements are written in an atomized manner, minimizing the content as much as possible. That allows the attribution of a requirement to different railways in a transparent and traceable manner.

A high-level view of the database structure is displayed on the following diagram.

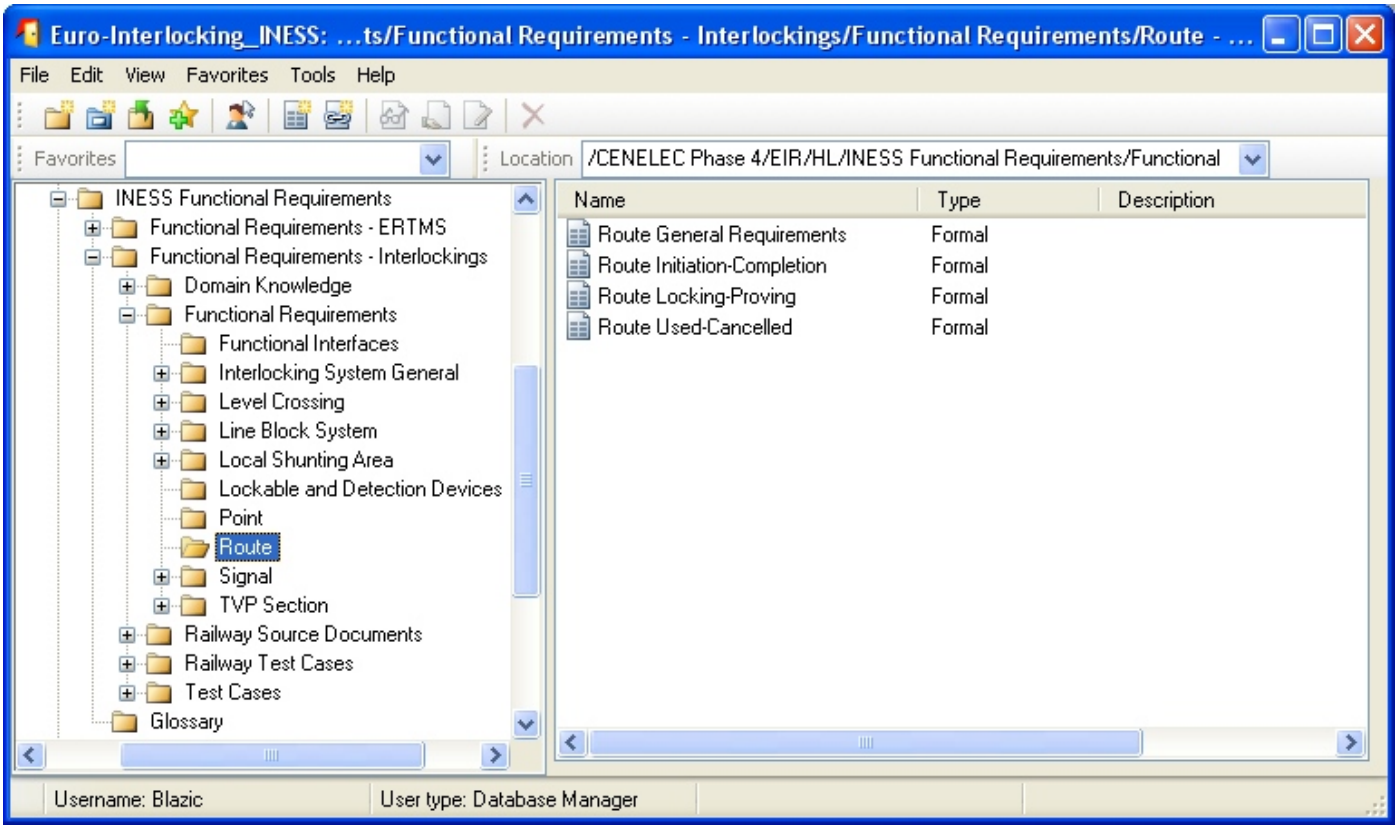

Figure 1

Functional requirements are located in the folder *INESS Functional Requirements*. This folder contains sub-folders containing functional requirements for ERTMS, functional requirements for conventional interlocking systems and the glossary.

The folder *Functional Requirements – ERTMS* will contain ERTMS requirements corresponding to WP D3. The contents will be defined once the work package will start.

The folder *Functional Requirements – Interlockings* will contain interlocking system functional requirements corresponding to WP D2. The folder consists of sub-folders representing the domain knowledge and the functional requirements. The functional requirements are grouped by logical interlocking concepts in the following manner:

Folder	**Description**
Interlocking System General	General requirements describing Interlocking System start-up procedures, bordering issues, operation modes…
Route	Requirements for setting, locking and using routes
Point	Requirements regulating powered points (moveable elements) and key locked points (moveable elements)
Signal	Requirements regulating signals and monitoring
Lockable and Detection Devices	Requirements regulating miscellaneous lockable and detection devices such as bridges, gates, slide detectors…
TVP Section	Requirements regulating TVP systems, including track circuit and axle counting types
Line Block	Requirements regulating line block systems
Level Crossing	Requirements describing the functionality of level crossings from the perspective of the interlocking system
Local Shunting Area	Requirements describing the local shunting area
Functional Interfaces	Requirements for handling commands, statuses, driving values, detected values

Table 2

A detailed overview of the structure of *Functional Requirements* folder is displayed on the following diagram.

FUNCTIONAL REQUIREMENTS FOLDER STRUCTURE

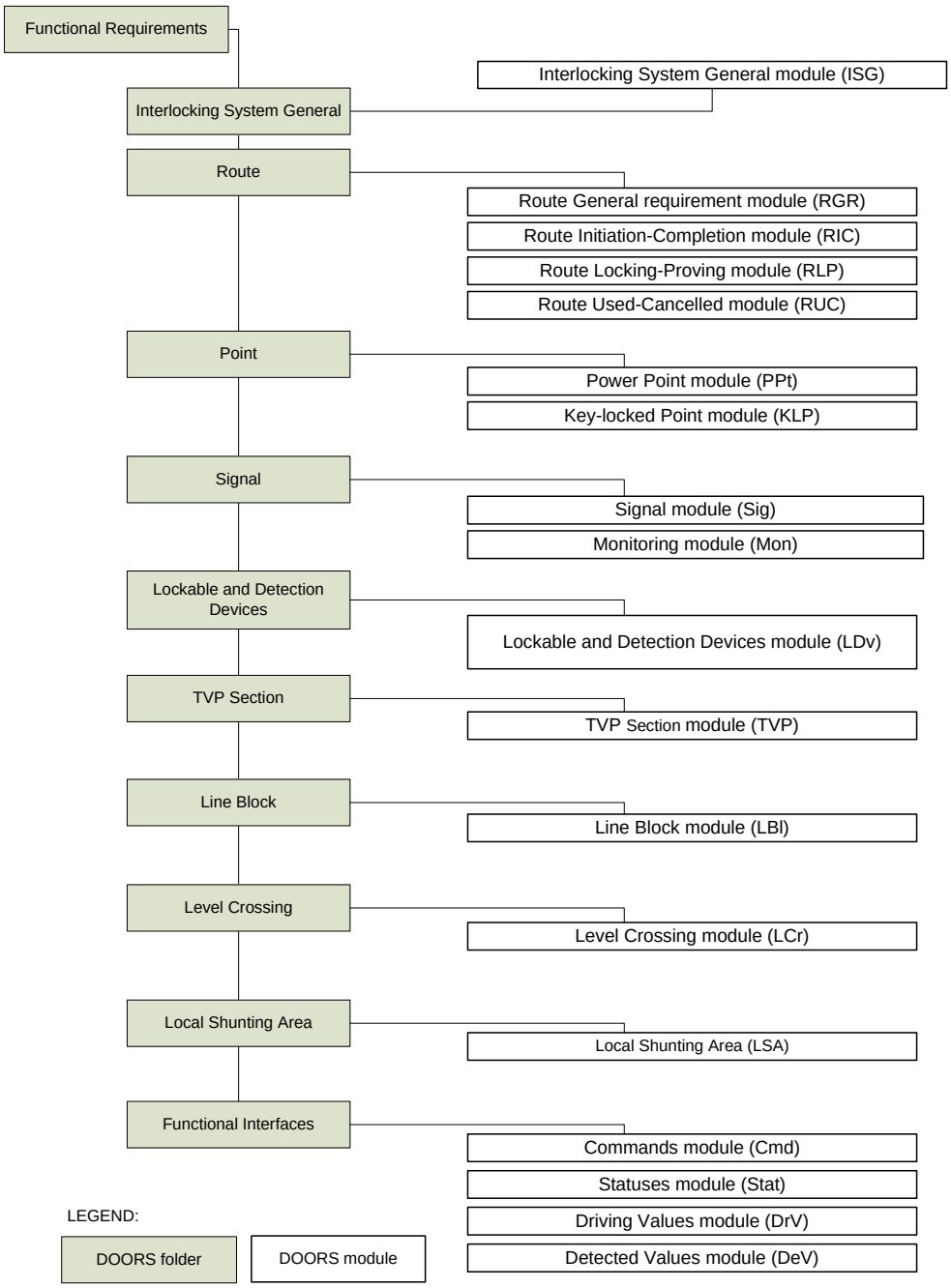

Figure 2

A functional requirements domain knowledge module is located in the folder ***Domain Knowledge***. The module is written with the intention to support and explain the functional requirements.

The folder ***Glossary*** contains the glossary of terms used in the development of the functional requirements, with the translations to the national terms of the participating railways.

3.1.3.1 Module properties

A functional requirements document in DOORS is a module. A module has its own attributes (not to be mixed with object attributes described in 3.1.3.2 Object properties).

In the database, the system attribute *Prefix* has to be used to define a prefix for the object identification number (Cmd, PPt, Sig…).

The attribute *Module Approved by* should indicate which modules have been baseline approved by a national railway. It uses the type Railway List, therefore the applicable country name has to be selected once a baseline becomes approved.

3.1.3.2 Object properties

An individual requirement in a DOORS module is an object within a module. An object in DOORS contains information in the form of attributes.

Content wise, the object contains attributes ***heading*** and ***text***. One object should never contain information in both, so care must be used to keep headings and text as separate objects.

Each object in DOORS has its own unique identification number (ID). The ID is set automatically by the system once the object is created. In the database, an attribute *Identifier* is used to indicate a suffix to the ID to allow easier identification between objects from different modules. A suffix is created based on the *Object Type* attribute.

The object's **attributes** provide more detailed information about the object. Some of the properties are system type and cannot be edited by the users. Such properties are access rights, history and linking information. Other user defined attributes are used to define additional characteristics of an object. Attributes assist in searching, filtering and sorting through the data within the module.

An object in a module utilises the following user defined attributes:

Attribute name	***Attribute type***	***Description***
EI-Comments	Text	A comment regarding the requirement from a Core Team engineer, to be used for describing open issues between the Core Team and the railways
Comment	Text	A comment regarding the requirement, providing more information about the requirement or referring to see also another requirement
Applicable to	Railway List	Railways to which the requirement applies (*Germany, Netherlands, ... U.K.*) **Attribute value must be empty for objects with headings only. This enables the inclusion of all headings in a single railway view, despite the empty section contents.**
Object Type	ObjType	Defines the type of statement (*comment, scenario, requirement, domain knowledge, term, diagram*)

Attribute name	*Attribute type*	*Description*
EI-Test Case	Text	States the in-link properties (source module name, source object ID) of the *TESTS* link from the EI test case which validates the requirement *to be created automatically based on the linking in DOORS (Encode links as text attributes - script)
EI-Safety Requirement	Text	States the in-link properties (source module name, source object ID) of the *REFERENCES* link from the Safety Requirement reference *to be created automatically based on the linking in DOORS (Encode links as text attributes - script)
EI-Domain Knowledge	Text	States the in-link properties (source module name, source object ID) of the *EXPLAINS* link from the EI domain knowledge object which explains the requirement *to be created automatically based on the linking in DOORS (Encode links as text attributes - script)
Railway-Status	Status Type	Status of the requirement from the national railway's perspective (*For team comment*, *For review, In preparation, Approved*)
Railway-Function Status	Function Type	Defines the status of the functionality (*existing, new, consideration*) Use of this attribute is optional.
Railway-Priority	Priority Type	Defines the priority of a requirement/function (*mandatory, optional*) Use of this attribute is optional.

Table 3

The national railway attributes (*Railway-*…) are created for each participating railway (see the Object Properties diagram), so the resulting attributes are for example *Germany-Priority, Germany-Status*…
The *Railway-Status* attribute of the requirement is used to indicate the stage of the review process (see also Change Proposals) and is to be selected based on the following:

Status	**Usage**
For review	The object (requirement) is new or the text has changed since the last baseline.
In preparation	Any unresolved change proposal concerning the object is pending.

Status	**Usage**
Approved	The object is approved by a railway and the containing module is baselined. The object has not changed since the last baseline and there are no unresolved change proposals concerning the object pending.
For team comment	The object is undergoing changes and update is not yet complete. This status applies only during the modules update, but not after its release for review.
Agreed	The object has been agreed and tagged by a railway and is waiting for approval.
Removed	Not used

Table 4

The *Railway-Function Status* attribute of the requirement is used to indicate the status of the functionality in terms of existing or new functions and is to be selected based on the following:

Status	**Usage**
Existing	The functionality described in the requirement is currently employed
New	The functionality described in the requirement is new and is agreed/approved for future use by the railway
Consideration	The functionality described in the requirement is under consideration for future use and requires further actions prior to agreement/approval

Table 5

The *Railway-Priority* attribute of the requirement is used to indicate the priority of a requirement and is to be selected based on the following:

Status	**Usage**
Mandatory	The requirement must be built into every interlocking system. The interlocking system cannot function without such a requirement.
Optional	The requirement may be built into an interlocking system, depending on the operational requirements, station layout, environmental conditions.

Table 6

The following Object Properties diagram shows the object with all of its attributes, attribute types and attribute values. Additionally, an example of an object with a requirement is shown.

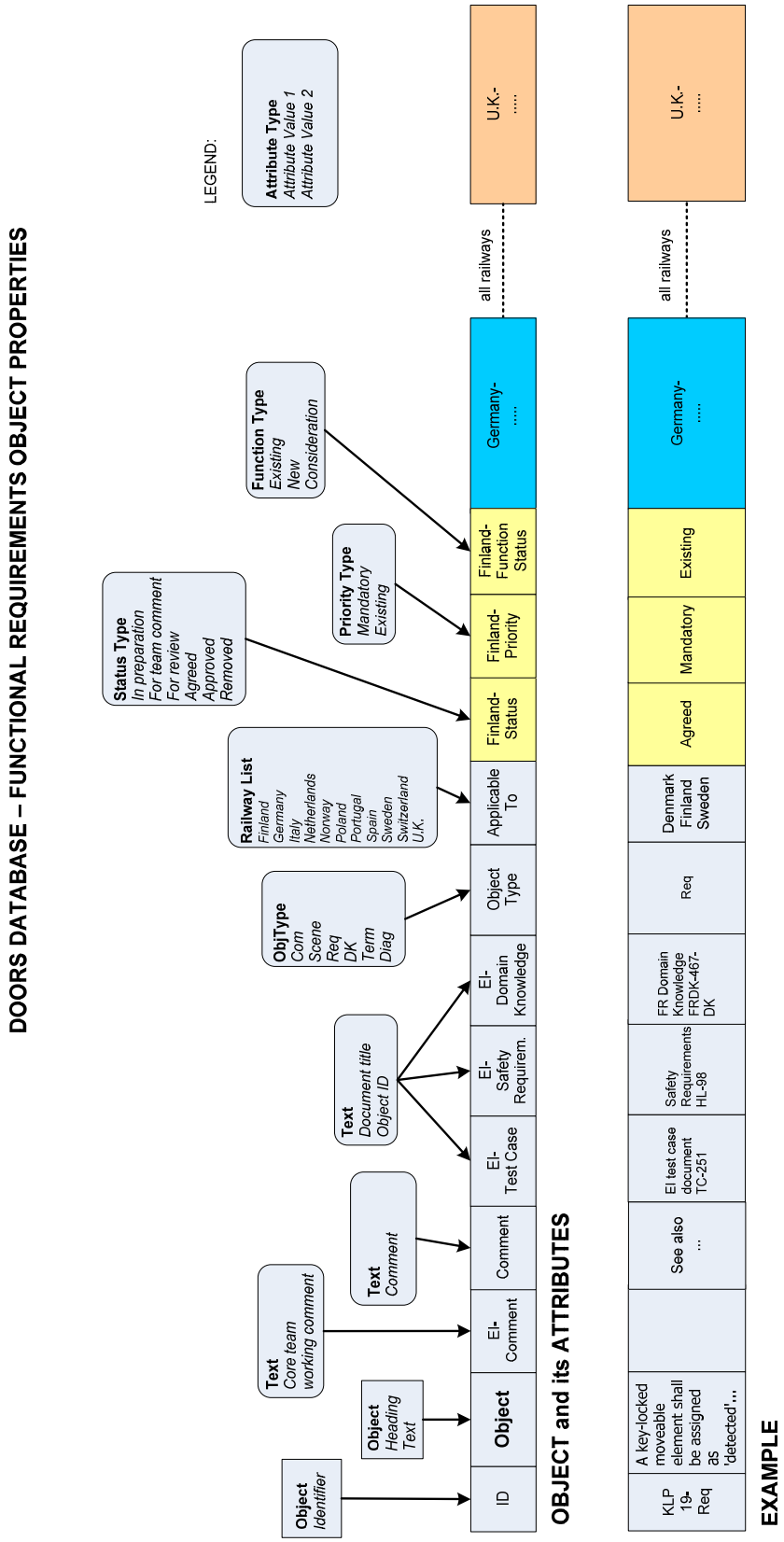
DOORS DATABASE – FUNCTIONAL REQUIREMENTS OBJECT PROPERTIES
LEGEND:
Attribute Type
Attribute Value 1
Attribute Value 2
Object Identifier
Object Heading Text
Text Core team working comment
Text Comment
Text Document title Object ID
ObjType Com Scene Req DK Term Diag
Railway List Finland Germany Italy Netherlands Norway Poland Portugal Spain Sweden Switzerland U.K.
Status Type In preparation For team comment For review Agreed Approved Removed
Priority Type Mandatory Existing
Function Type Existing New Consideration
ID
Object
EI-Comment
Comment
EI-Test Case
EI-Safety Requirem.
EI-Domain Knowledge
Object Type
Applicable To
Finland-Status
Finland-Priority
Finland-Function Status
Germany-.....
all railways
U.K.-.....
OBJECT and its ATTRIBUTES
KLP 19-Req
A key-locked moveable element shall be assigned as 'detected'...
See also ...
EI test case document TC-251
Safety Requirements HL-98
FR Domain Knowledge FRDK-467-DK
Req
Denmark Finland Sweden
Agreed
Mandatory
Existing
Germany-.....
all railways
U.K.-.....
EXAMPLE

Figure 3

3.1.3.2.1 Functional Interfaces object properties

In addition to the general attributes listed in the section above, the Commands module includes the following attributes:

Attribute name	***Attribute type***	***Description***
Command parameter	Text	The parameter to identify the object the command is applicable to (element identification (ID), station ID, route ID, local shunting area ID...)
Command User	Command User Type	The user which issues the command (signaller, shunter...)
Railway – Command properties	Command Level type	The level of command for the applicable railway
Term - Railway	Text	The translation of the command to the national command name of each railway

Table 7

The national railway attributes (*Railway-*…) are created for each participating railway (see the Object Properties diagram), so the resulting attributes are for example *Germany-Command properties, Term-Germany*…
The *Command User* attribute of a command is used to indicate the user of the interlocking system that issues the command. It is to be selected based on the following:

Status	**Usage**
Signaller	The signaller is the main user of the interlocking system to operate with traffic, accessing the interlocking system through a traffic control system or a local operating panel.
Maintainer	The maintainer accesses the interlocking system for the purpose of maintenance through a traffic control system, local operating panel or a maintenance system.
Shunter	The shunter accesses the interlocking system for the purposes of shunting movements from the field through local point control panel.
Local device operator	Local device operator operates a local device such as a level crossing or a lockable device.
Platform staff	Platform staff operates with the interlocking system for the purpose of departing trains.
Automatic route setting system	Automatic route setting system is a user which sets routes automatically.

Table 8

The *Railway-Command properties* attribute of a command is used to indicate the level of command regarding the need to additionally validate the command or to record the command, and is to be selected based on the following:

Status	**Usage**
Level 0	Command severity level 0
Level 1	Command severity level 1
Level 2	Command severity level 2

Table 9

The levels must be apportioned separately to each railway, with the use of domain knowledge module.

In addition to the general attributes listed in the section above, the Statuses module includes the following attributes:

Attribute name	***Attribute type***	***Description***
Term - Railway	Text	The translation of the status to the national status term of each railway

Table 10

The following Object Properties diagram shows the object with all of its attributes, attribute types and attribute values for the Commands and Statuses modules.

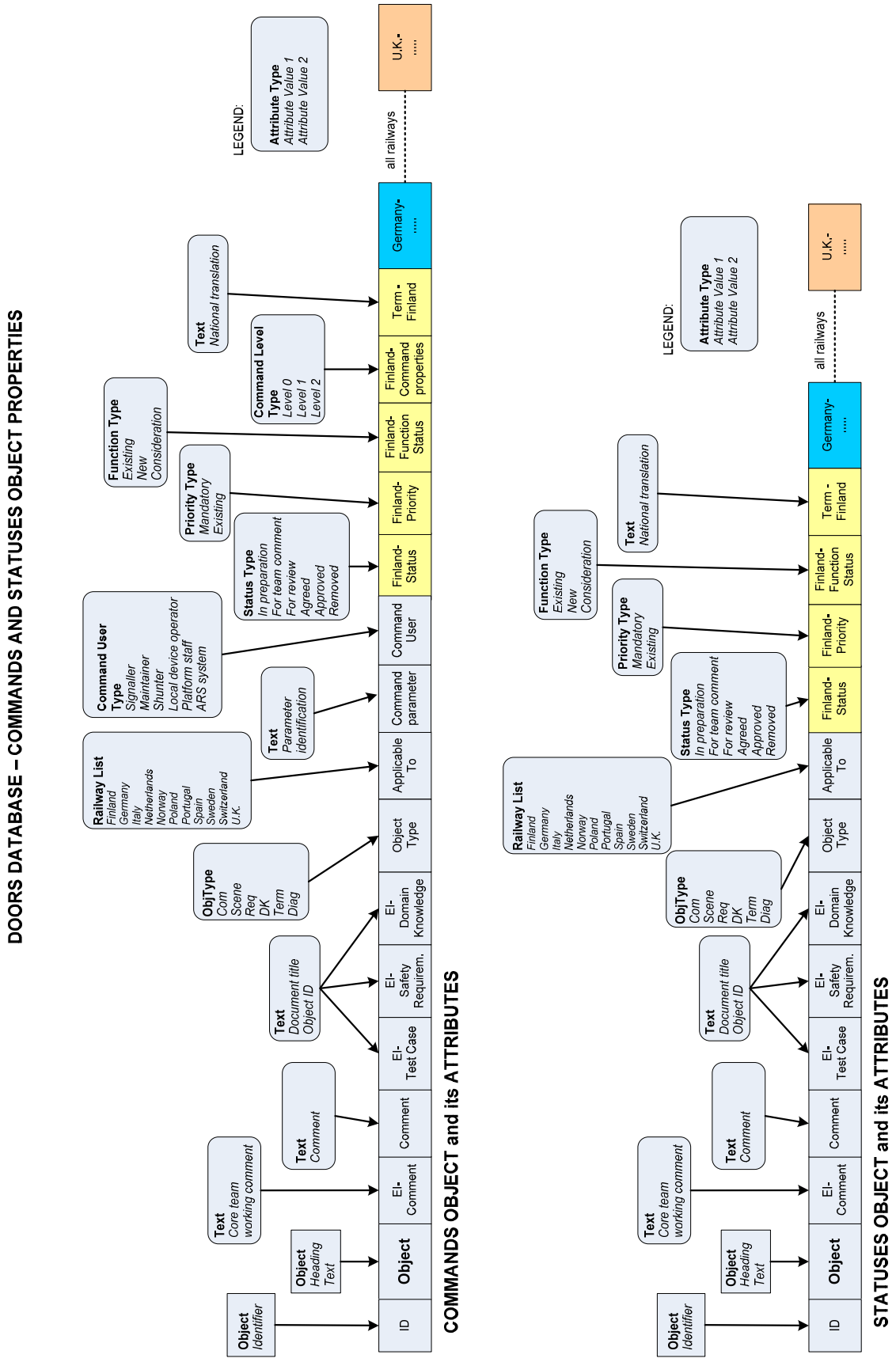
DOORS DATABASE – COMMANDS AND STATUSES OBJECT PROPERTIES
COMMANDS OBJECT and its ATTRIBUTES
STATUSES OBJECT and its ATTRIBUTES
LEGEND:
Attribute Type
Attribute Value 1
Attribute Value 2
all railways
U.K.-
Germany-
ID
Object
Object Identifier
Object Heading Text
Text Core team working comment
Text Comment
EI-Comment
EI-Test Case
EI-Safety Requirem.
EI-Domain Knowledge
Text Document title Object ID
ObjType Com Scene Req DK Term Diag
Object Type
Railway List Finland Germany Italy Netherlands Norway Poland Portugal Spain Sweden Switzerland U.K.
Applicable To
Text Parameter identification
Command parameter
Command User Type Signaller Maintainer Shunter Local device operator Platform staff ARS system
Command User
Status Type In preparation For team comment For review Agreed Approved Removed
Finland-Status
Priority Type Mandatory Existing
Finland-Priority
Function Type Existing New Consideration
Finland-Function Status
Command Level Type Level 0 Level 1 Level 2
Finland-Command properties
Text National translation
Term - Finland

Figure 4

3.1.3.2.2 Glossary object properties

In addition to the general attributes listed in the section "Object Properties", the Glossary module includes the following attributes:

Attribute name	***Attribute type***	***Description***
Term - Railway	Text	The translation of the term to the national term of each railway

Table 11

The following Object Properties diagram shows the object with all of its attributes, attribute types and attribute values for the Glossary module.

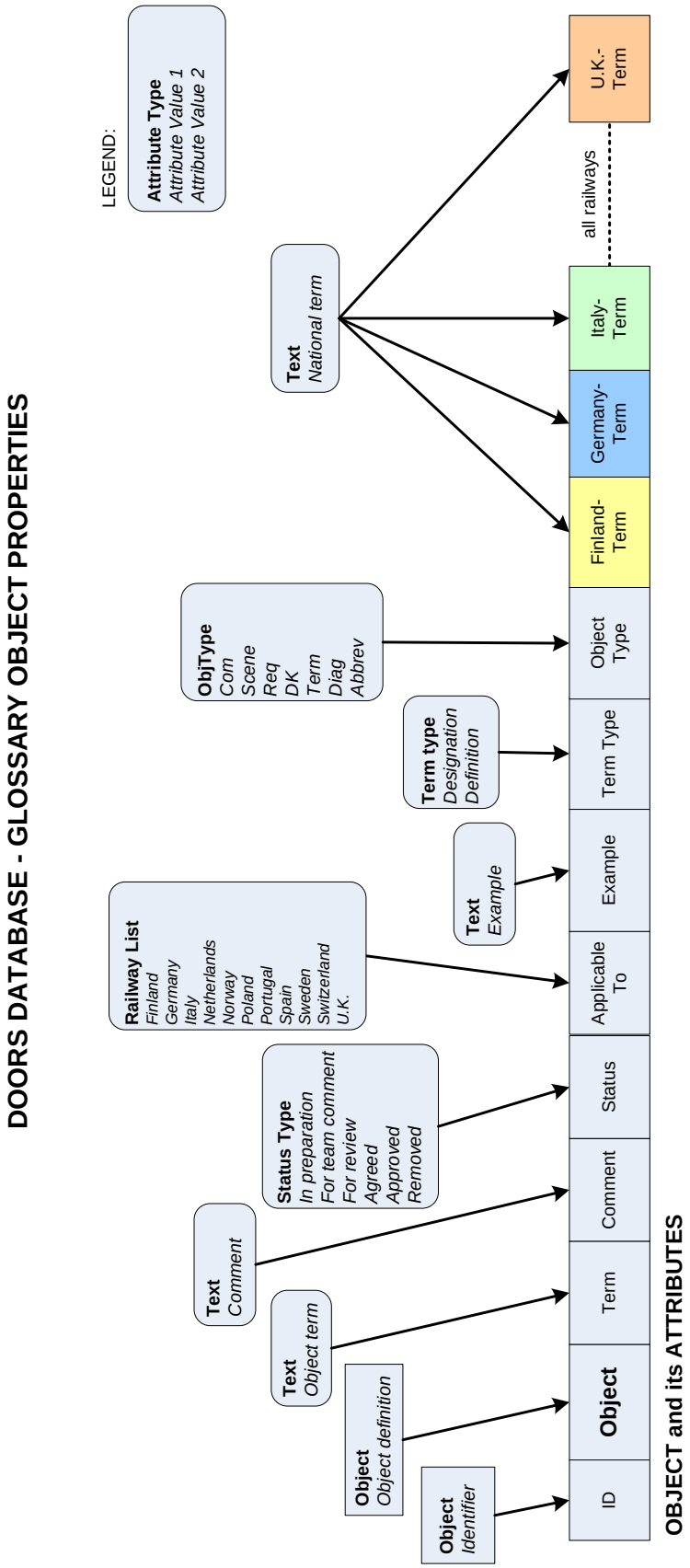
DOORS DATABASE - GLOSSARY OBJECT PROPERTIES
LEGEND:
Attribute Type
Attribute Value 1
Attribute Value 2
Text
National term
all railways
U.K.-Term
Italy-Term
Germany-Term
Finland-Term
ObjType
Com
Scene
Req
DK
Term
Diag
Abbrev
Term type
Designation
Definition
Text
Example
Railway List
Finland
Germany
Italy
Netherlands
Norway
Poland
Portugal
Spain
Sweden
Switzerland
U.K.
Status Type
In preparation
For team comment
For review
Agreed
Approved
Removed
Text
Comment
Text
Object term
Object
Object definition
Object
Identifier
ID
Object
Term
Comment
Status
Applicable To
Example
Term Type
Object Type
OBJECT and its ATTRIBUTES

Figure 5

3.1.3.3 Change proposals

Change proposal system for a particular module or a set of modules is to be activated once a functional requirements module in DOORS is considered stable. This is performed by the *Database Project Manager*.

Once the change proposal system is activated, the requirements review process is to be used as shown below. The handling of change proposals in DOORS is described in the Euro-Interlocking document "Using DOORS in the Requirements Development Process".

The following diagram indicates the review process of the requirements with the relations to change proposals, with the actual DOORS statuses indicated.

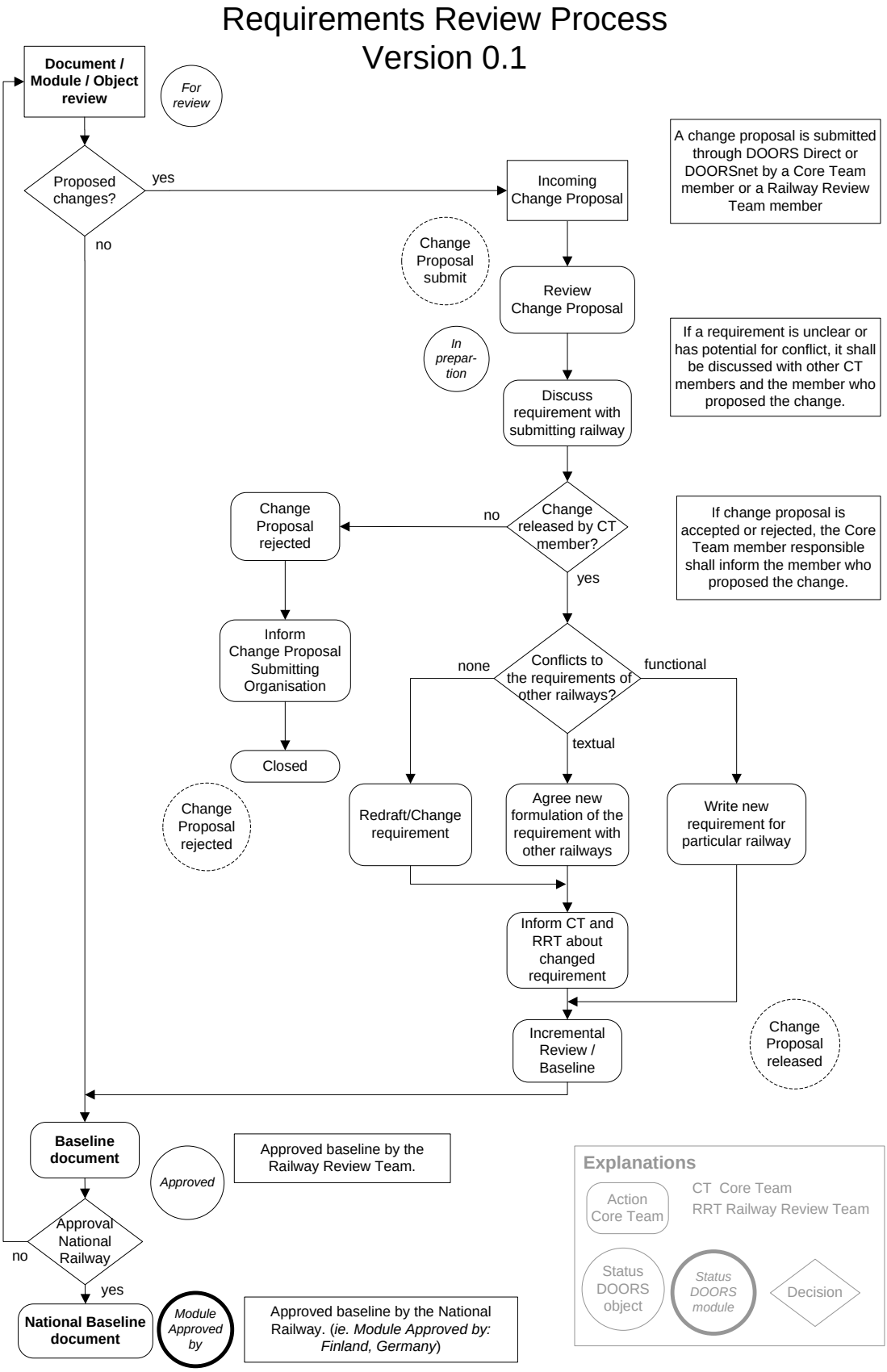
Requirements Review Process
Version 0.1
Document / Module / Object review
For review
Proposed changes?
yes
no
Incoming Change Proposal
A change proposal is submitted through DOORS Direct or DOORSnet by a Core Team member or a Railway Review Team member
Change Proposal submit
Review Change Proposal
In prepar-tion
If a requirement is unclear or has potential for conflict, it shall be discussed with other CT members and the member who proposed the change.
Discuss requirement with submitting railway
Change released by CT member?
no
yes
Change Proposal rejected
If change proposal is accepted or rejected, the Core Team member responsible shall inform the member who proposed the change.
Inform Change Proposal Submitting Organisation
Conflicts to the requirements of other railways?
none
functional
textual
Closed
Change Proposal rejected
Redraft/Change requirement
Agree new formulation of the requirement with other railways
Write new requirement for particular railway
Inform CT and RRT about changed requirement
Incremental Review / Baseline
Change Proposal released
Baseline document
Approved
Approved baseline by the Railway Review Team.
Approval National Railway
no
yes
National Baseline document
Module Approved by
Approved baseline by the National Railway. (ie. Module Approved by: Finland, Germany)
Explanations
Action Core Team
CT Core Team
RRT Railway Review Team
Status DOORS object
Status DOORS module
Decision

Figure 6

3.2 Formal expression of the requirements

3.2.1 Purpose of functional requirements modelling

In addition to the structured natural English used for the expression of the textual requirements, it has also been decided to model the interlocking specification in a more formal syntax, in order to achieve a rigour that is unavailable if the functions are only specified textually. The primary reason for this, is that given the complexity of the system and the inevitable degree to which closely-related functions are spread over a number of modules in the DOORS database, it is often difficult for the reader to keep all the relevant functional dependencies in mind when reviewing a textual document. However, by modelling the interlocking functions, the requirements become more integrated and this aids comprehension. Furthermore, by modelling the complete requirements in a manner that enables them to be simulated, the problems raised by the dependencies of the textual requirements disappear altogether and this actually allows the requirements to be properly validated in a manner that is simply not possible by merely reading a set of written documents.

The simulation itself is a dynamic realisation of the model complemented by the configuration data of a particular layout. Thus although the primary aim is to validate the requirements themselves, it could be argued that in fact one is checking a combination of the model, the layout configuration data and indeed the simulator itself when validating. For the purposes of this document, the simulator is transparent and can be presumed to be an honest broker, interpreting the model transparently. However, the layout clearly needs to be compatible with the functional requirements if the simulation environment is to work correctly. To put it another way, unexpected results during simulation can just as much be the result of erroneous data configuration as mistakes in the requirements or the modelling. Thus merely having a simulation environment available does not license the modelled requirements to become decoupled from the textual requirements. Quite the contrary. Because of the difference of the complementary, location-specific configuration data from that of the generic rule-based requirements present in the simulator, it is best that the simulation environment easily allows a run-time differentiation of the rules from the configuration and ultimately the best guarantee for this, when the simulation environment is transparent, is by having a model that is clear and in which the representation and dependencies of the textual requirements is easily recognisable.

Consequently, when seeking to simulate a functional specification (as opposed to writing the software for a final product implementation) it is imperative that the manner in which the requirements are modelled is *in itself* as clear and as declarative as the requirements themselves, so that the relationship between the text and the model is as explicit as possible. It is not necessary that there is always a clear one-to-one relationship between a written requirement and the manner in which it is modelled, as this would often result in ineffective modelling. However, it is necessary that the model reflects the requirements lucidly, as this ultimately provides the best guarantee to transparency. For this reason, it has been decided not to try and artificially link the requirements in DOORS with their modelled counterparts, as although this solution would work for many requirements, the fact that it is not practicable for all would undermine its benefit. Instead the approach is that the model of the requirements should follow the syntax of the written requirements in as far as is feasible and practical, as it is this syntax that provides the traceability between the two artefacts. For this to be effective requires a high level of consistency in the modelling approach, but above all, it requires that the language used in the model is as clear and declarative as possible.

3.2.2 Object oriented modelling in UML

Given the above-mentioned circumstances and in order to reconcile the need for formality, executability and transparency, it was decided in the Euro-Interlocking project to adopt UML for the formal expression of the requirements and this will also apply for the INESS project. There were a number of issues which lead to this choice which are worth taking note of:

- UML is object-orientated, thus its concepts for the representation of concrete objects in the physical world can be expressed in a manner which is only minimally abstracted. This has the advantage of enabling the modeller to retain all the concepts and principles of the textual database within the model.
- The associated expression and action language used for determining the functionality of the interlocking (xUML) is declarative, not procedural (it says what to do and not how to do it) and it thus closely adheres to the text in which the requirement is expressed. The close relationship between the DOORS modules and the UML classes and between the structured English syntax and that of the action language means that the functions in both the textual database and the model can be syntactically close. This promotes a high-degree of traceability between the two artefacts which in turn, promotes transparency.
- Modelling in itself promotes the consistent use of terms, thus reducing the confusion associated with free text requirements in which references are not harmonised. Indeed the particular CASE tool which will be used to model the requirements in the project, supports consistency of terms throughout the model by storing them in a glossary. This means that by always referring to a central repository for a particular term, unnecessary redundancy can be avoided and this in itself raises the quality of the model, as there is less risk that similar, but non-identical terms are defined for ones that already exist.
- The modelled artefacts are replicable, which means that once the function of a given concept or object has been correctly captured, it can be instantiated as often as needed. Furthermore, as the model is generic and its applicability not restricted to a given track layout, this further raises the value of it being replicable, as each class represents a single, unified and complete repository of the global functionality of the given object. The fundamental advantage of this is that there is no sacrifice of quality as a result of instantiation, as all objects of a similar type are always created from the same source and will thus behave in the same way.
- UML supports the concept of generalisation / specialisation, by which objects that share common features need only vary from one another in as far as they really have differences; all common features need only be modelled once. This has the advantage of reducing redundancy and the errors that are often associated with it. In this respect the model does differ somewhat from the representation of the requirements in the DOORS database, as there, similar functionality is written out in full. However, given the advantages inherent in the generalisation / specialisation feature of UML, it is felt that it would be of little advantage to ignore the concept and the advantages it brings for the sake of promoting conformity with the more restricted nature of natural language and the redundancy that it brings.

Thus although UML is by no means the only language available in which the requirements could be represented, given the importance of the above-mentioned issues, it is felt that it provides the most apposite level of abstraction with which to model the requirements formally for the scope of this project. This is especially the case, given that it is desired both to have a declarative model and the possibility to simulate it, as it is clear that a complex specification can only be thoroughly validated by being executed.

3.2.3 Unified Modelling language overview

It is perhaps important to note at the outset that UML is *not* a method that guides you through the development process, it is merely a standardised language that defines *how* to represent (i.e. document) certain artefacts of an object-oriented system. That means, although it defines how to represent certain aspects of the system, it does not say anything on how to discover those artefacts. UML just defines the syntax and the semantics of various modelling elements that should be used to document an object-oriented system.

The goal of modelling requirements based on the object-oriented paradigm is to abstract concepts of the real world that are relevant for the planned system. These abstractions are documented as classes in a graphical form; the so-called class diagram. A class diagram describes the abstracted objects of the real world, together with information about their properties and the relationships between them. As only what is specified in the requirements modules in the DOORS database will be modelled, the resulting model will capture the functions of the system, but not its performance and for that reason it is important to note that although the simulator possesses many of the features of an interlocking, it considers none of the usual performance requirements of a real interlocking, for the simple reason that they are not expressed in the functional requirements specification.

3.2.3.1 Expression and action language

Many of the modelling choices made in the translation of the textual requirements to xUML will be laid out in a follow-up document, but in as far as this document is concerned with the language in which both the textual and the modelled requirements are expressed, certain precepts of the xUML syntax can be laid out here. We shall start by examining the language in which the conditions and rules of the specification are expressed.

3.2.3.1.1 Expression language

There are a number of variants of the expression language, but in what follows the majority is based on Boolean expressions, whereby expressions are used to express rules and rule sets which are then evaluated to true or false. To a large degree this is determined by the nature of the interlocking specification, but it is partially also a conscious choice, in order to keep the modelling style close to the written requirements.

In the chosen modelling style most of the expressions are apportioned to one of the two forms of class property that are used; these are the ‘declared’ and the ‘derived’ attribute. In the former a value is simply appended to an attribute, such as 'signal_direction' := ‘outbound’. The attribute will carry this value (and return it when asked to) for as long as it exists or until its value is changed. The second type, the derived attribute is more interesting in the context of the language of the model, as it contains not just static values, but also rules or sets of them which need to be evaluated. These rules can be quite varied, but generally the expression behind them requires that they return a Boolean value as and when required by the functionality of the system. As the naming convention for them is liberal, they can actually additionally describe the nature of the rule which they store and as we shall see, these rules follow the syntax of the textual requirements quite closely.

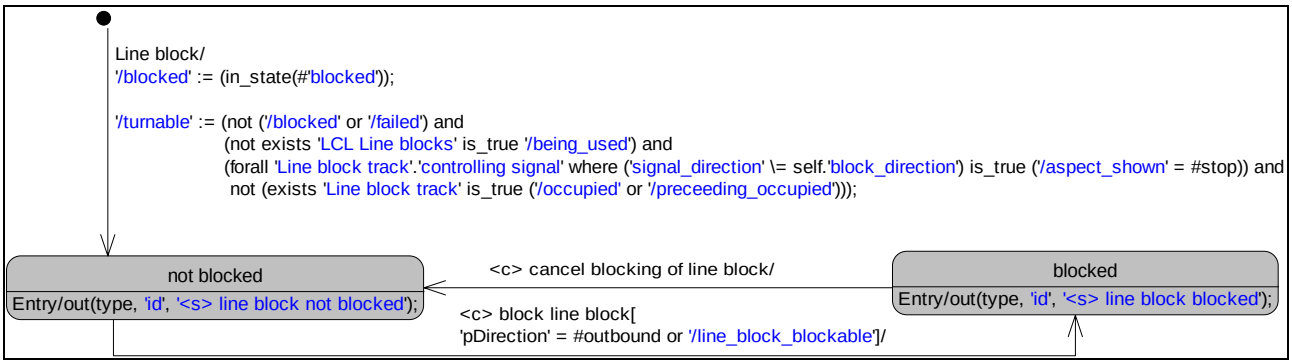

Figure 7 State diagram of the class Line block

The state diagram shown in Figure 7 shows the possible blocking statuses of a Line block. Simply put, the object created from this class can be 'blocked' or 'not blocked' and this information is shown in the two grey state boxes on either side of the graphic. These statuses correspond to defined states in the written requirements and in the context of the Line block itself they have both meaning and functional significance. However, strictly speaking whether or not an instance of this Line block is 'blocked' or not can only be known by querying the private states of the object; thus the Line block would be blocked if it is in the state 'blocked'. In the expression language used in the INESS model, this will be expressed so: in_state(#'blocked'). However, although this formulation is not exactly counter-intuitive, it is not what appears in the textual requirements. There we would read that such and such a condition "*shall be prevented while the block travel direction is 'blocked'".* Clearly *'blocked'* is preferable to in_state(#'blocked') and indeed the expression language can support this close alignment with natural English rules once derived attributes are used.

The fact as to whether or not the Line block instance is in_state(#'blocked') can simply be assigned to an attribute via the ':=' mechanism and the attribute can then be named according to the information it provides:

'/blocked' := (in_state(#'blocked'));

This solution also has the additional advantage that now the blocking information is not only available in the private states of the class, it can also be made available via the object's interface to the rest of the system and moreover, this information can be published via the moniker '/blocked' i.e. exactly as it is expressed in the written requirements. This is because unlike states which are private to a class, UML provides a mechanism by which it can be declared whether an attribute's value is available only to itself but also to other objects. This is denoted via the '+' (public) or '-' sign allocated to the attribute, as can be seen below in Figure 8.

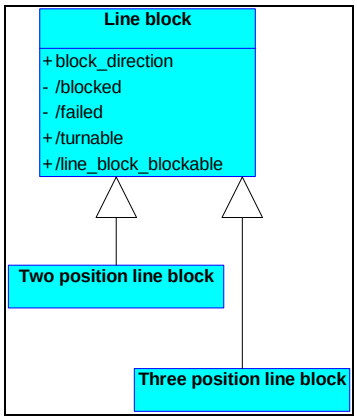

Figure 8 Class diagram showing the sub-types of the class Line block.

The only difference between the name in the textual requirements and the model, is that the attribute name is preceded by a slash '/'. This merely indicates to the reader and to the simulator that the attribute needs to evaluate its allocated rule to find out its current value.

The nature of expressions that can be handled by derived attributes is not limited to simple state evaluations; more complex rules can be specified too.

The other derived attribute in the Line block class shown in Figure 7 reads as follows:

'/turnable' := (not ('/blocked' or '/failed') and
(not exists 'LCL Line blocks' is_true '/being_used') and
(forall 'Line block track'.'controlling signal' where ('signal_direction' \= self.'block_direction')
is_true ('/aspect_shown' = #stop)) and
not (exists 'Line block track' is_true ('/occupied' or '/preceding_occupied')));

Although this attribute will ultimately also evaluate to either 'true' or 'false', the rule behind it is clearly much more complex, so we will examine it line by line:

'/turnable' := (not ('/blocked' or '/failed')…

As the information as to whether the Line block is in_state(#'blocked') has already been captured in the attribute '/blocked', it reads more easily to refer directly to that attribute, even though the state information is also available, as we are still in the same instance. The rule for the second derived attribute; '/failed' is not visible on this class, even though the attribute belongs to the Line block class, as can also be seen in the class diagram below in Figure 8. The reason for this is because it is only populated on its sub-classes, Two position line block and Three position line block. This issue need not be examined in detail here, as it will be dealt with in a subsequent document dealing with the modelled requirements.

The next line of the expression reads:

not exists 'LCL Line blocks' is_true '/being_used'

This bring us to another advantage of the naming conventions of UML: associations. Although the script of the both '/failed' and 'LCL Line blocks' appears in the same blue-coloured text, it is a convention of the INESS modelling style that attributes are consistently written in lower case characters with low lines instead of spaces between the individual words in their names, e.g.:

'/physically_in_position' := (in_state(#'active'.'position'.'in position'));

The fact that 'LCL Line blocks' is written without spaces, shows, in this context, that it is a role name. It describes the nature of the relationship between the two classes which it links; in this case the class Line block and Logical line block – as shown in Figure 9. Thus in this part of the rule for the evaluation of '/turnable', it is required to leave the state Line block and navigate along the role 'LCL Line blocks' to check the status of all the Logical line block objects at the end of the association – as there may be more than one of them.

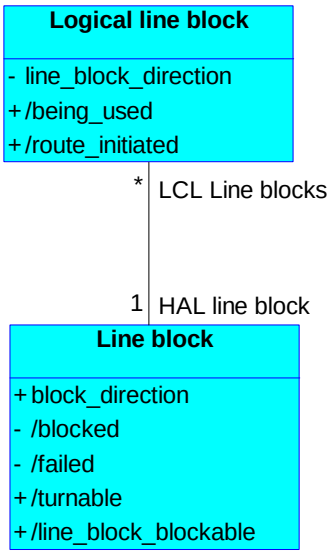

Figure 9 Diagram of the associations between the classes Line block and Logical line block

It is worth noting that this is done by querying a derived attribute on the Logical line block object and not by using an 'in_state' expression. As mentioned before, for an object to check its own status by evaluating a derived attribute rather than its own states directly merely helps raise the clarity of the expressions. However for an object to request internal state information of an another object via an association would be wrong, as it cannot be presumed that the information is 'publically' available and thus it is important that the data which is available to other instances is clearly defined in attributes which are explicitly accessible to other objects.

In the particular example shown in Figure 9 there is only one association between the classes Line block and Logical line block. However classes may be linked by more than just one association, and if they do not have clear names, it is not possible to determine along which association to navigate. This would cause problems if different instances of the target class were at the end of the two associations, as it would not be clear which object to query. Thus assigning role names to the ends of associations enables the target object to be defined in a particular context. This is particularly relevant in the relationship between tracks and points shown below in Figure 10. Here there are many possible roles by which objects of these classes can be linked and thus it is imperative to name them all clearly. Taking an example from Figure 10; it would make no sense for the same instance of Track to be both the 'fouling track' of a point as well as being the 'fouled track' of the same Point. These associations will lead to different instances of Point and for that reason the roles must be clearly named so as to allow the correct object to be addressed.

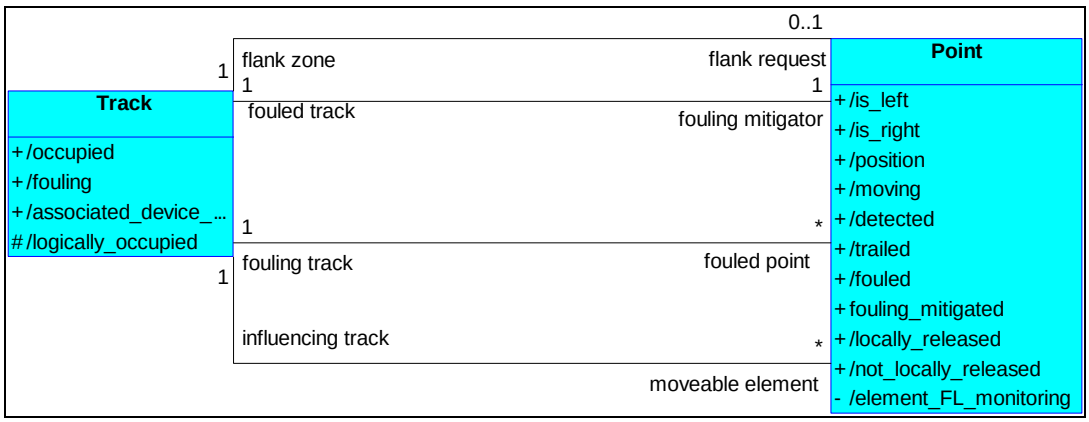

Figure 10 Diagram of the associations between the classes Track and Point

Returning to our former example, we can see that by navigating away from Line block to the class Logical line block via the role 'LCL Line blocks' more information can be evaluated to see if it is going to be possible to turn the direction of the line block. In this the query is as to whether the particular 'LCL Line blocks' addressed are '/being_used'.

Seen superficially 'being used' could mean any number of things, but given what we have seen above, it should not come as a surprise to see that in the class Logical line block that there is the following derived attribute:

'/being_used' := (not in_state(#'inactive'));

Again, if one wishes to further discover what 'inactive' really means in the context of a Logical line block, one can read the corresponding state diagram to see what leads it to become not in_state(#'inactive'). This procedure gives an idea of the degree of nesting which this modelling style gives rise to. As mentioned above it is considered important to avoid the redundancy of having the same information present in more than one place, for the simple reason that at some point the values will diverge. A result of this approach is that information generally needs to be collected from its source, and this often involves a degree of navigation, but at least guarantees the information is correctly provenanced.

Moving on to the third line of the expression we read:

forall 'Line block track'.'controlling signal' where ('signal_direction' \= self.'block_direction') is_true ('/aspect_shown' = #stop) …

This time instead of just navigating along the association identified by the class 'Line block track' it is specified to continue from there along the role 'controlling signal'. i.e. we have to navigate via two classes. Naturally it is hoped that the names used in the navigation are sufficiently clear to imply that the object to be queried is in fact the controlling signal of the Line block track. It is worth noting that the association between Line block and Line block track has no name. In this case the role simply takes the name of the class being addressed. This is only possible when two classes are linked by only one association. The reason that no name has been used for the role is simply that the class name itself is already an apposite role name. Incidentally, navigating via the Class name is only possible if there is no role name. For example to navigate from Line block track to Line block signal via the class name and not via the role 'controlling signal' would be incorrect.

This brings us to the preposition of this part of the expression, which is not 'exists', but 'forall' and this presents an opportunity to examine the issue of cardinalities. A line block is potentially made up of any number of Line block tracks, each of which must be controlled by a signal. This is shown in the class diagram, Figure 11.

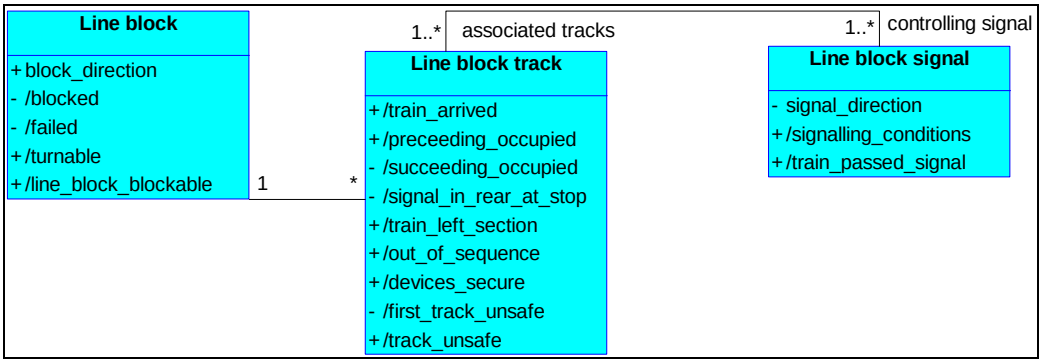

Figure 11 Diagram of some Line block classes

We have already seen how an association can define the existence of a relationship between two objects. In addition to this, it can also determine the number of objects which can be connected via the association. The delimiters used in INESS are 0…1, 1, or many (which is indicated with a *). By this

simple mechanism many requirements can be implicitly fulfilled by restricting or enforcing certain cardinalities.

In the case of the relationship between the Line block and the Line block track the star at the far end of the relationship simply shows that a Line block can have any number of tracks associated to it. The Line block track however, must have at least one signal that governs access to it.

Thus to return to the expression:

> forall 'Line block track'.'controlling signal' … is_true ('/aspect_shown' = #stop) …

means that for each and every track in the Line block the corresponding controlling signal must be at stop. If there were no tracks at all – which would be odd, but permitted – the expression would also be true, but as soon as there is one instance of Line block track, if its 'controlling signal' does not have '/aspect_shown' = #stop, the expression would have to evaluate to false.

Previously the 'is_true' concept was used to check whether another derived attribute was true (not exists 'LCL Line blocks' is_true '/being_used'). In this example, a derived attribute is compared to a pre-defined string. This rule could have been formulated more simply as

> forall 'Line block track'.'controlling signal' is_true '/at_stop'

the reason why it does not lies in the requirement from which it is derived, which reads:

> *all block signals opposing the current block travel direction are displaying a 'stop' aspect.*

This is an example in which the interlocking itself is not in a position to provide the information required for the rule to be evaluated, but instead it has to fetch it.

Thus the rule behind '/aspect_shown' reads as follows:

> '/aspect_shown' := (call 'physical'.'<dv> displayed aspect');

This is another example of a derived attribute, but one which uses a different mechanism to evaluate its value. Instead of navigating to another class and reading its status, it navigates outside the system and requires the target to provide its status by 'calling' the event that causes the signal to deliver the status. This is actually an example of action language being in an expression. The reason for this practice is that it is a modelling precept in INESS that whereas information present within the system can be implicitly derived, information that lies beyond the system boundary must be explicitly collected. This guarantees that the response is the actual aspect currently being displayed by the signal rather than the aspect that the interlocking last commanded it to display. Given that light bulbs burnt out, these two values can be different and thus this requirement veers on the side of safety in following the part of the textual requirement that reads "*are displaying a 'stop' aspect*", by checking the aspect rather than merely relying on what the system commanded. Consequently the model consistently shadows the written requirements.

Now we shall examine the 'where' clause of the expression:

> forall 'Line block track'.'controlling signal' where ('signal_direction' \= self.'block_direction') is_true ('/aspect_shown' = #stop) …

As the syntax and brackets imply, 'where' is a form of pre-filtering that should take place before the main clause is evaluated. The sense behind the clause is implied in the *opposing the current block travel direction* of the textual requirement mentioned above. A Line block track can have a signal at either end of it, governing the flow of traffic in both directions. However, the requirement is only concerned with those signals that are not facing the current direction of the line block traffic and consequently only these signals are quizzed as to whether or not they are displaying a stop aspect.

As to the actual syntax of the clause, the construct '\=' simply means 'is not equal to' and the appearance of 'self' before 'block_direction' bring us to the issue of context.

As a rule, clauses are evaluated in the context of wherever the navigation has currently reached. In the case of our example, this is the 'controlling signal'. However, although 'signal_direction' is something that is known to the controlling signal, the direction of the block is not. Thus for the expression to be evaluated, it is necessary to leave the current context and go to the object on which the property is defined, which in this case is an instance of the class Line block. For as Figure 11 clearly shows, while 'signal_direction' is a property of the Line block signal, 'block_direction' is a property of the Line block itself. This change of scope is limited only to the property qualified by 'self', since the rest of the expression, in which the aspect of the signal is checked, needs to happen in the previously-established context, namely, the Line block signal identified by the 'controlling signal'.

It is also worth noting here that 'signal_direction' and 'block_direction' are both declared attributes whose values are not derived from other information present in the model, but are assigned as the result an action. We shall examine the process by which this is specified in the section below on action language.

We have now arrived at the last clause of the expression for '/turnable'.

```
not exists 'Line block track' is_true ('/occupied' or '/preceding_occupied')
```

This clause raises little new and even covers the requirements "*all TVP sections in the line block are not 'occupied*" and "*the last TVP section in the station in the current outbound direction of the block travel direction is not 'occupied*"" more succinctly than the text itself. It is worth noting the importance of the brackets around the 'or' clause. Without them, the expression would only evaluate true if there was a Line block track that had a preceding track that was not occupied, which is clearly not what is in the requirement.

Lastly, one expression form not in the Line block state diagram is the conditional expression 'if / otherwise'. This allows a dynamic evaluation to determine which particular expression is actually to be evaluated in a given situation. It is a particularly powerful concept, as it basically allows for an attribute to change its rule according the circumstances.

Taken to extremes, this would mean that each class could store all its rules in just one derived attribute and which sub-rule was to be evaluated would be determined by the circumstances. However, this would clearly contravene the principle of clarity in the model. Going to the opposite extreme, it could also be argued that if an object's behaviour varies according to the circumstances, then it is not really the same object and it would be better to model its functionality separately in a new class. However in the case in point, I feel it can be argued that sometimes the behaviour of an object changes and if this is due to a major change in the environment or setting, then it is reasonable for the rule sets in the attributes to be toggled.

In railway parlance there is a variation of a main route that is called a drive on sight route (DOS), for which the safety conditions are slightly different than for a main route, but not sufficiently so that every route object is affected. Thus in order to avoid creating a whole set of DOS route objects, it was chosen to define alternative rule sets for those objects that behave differently for a DOS route than for a main route. As these differences persist for an entire route cycle, the time span for the change is clearly defined.

When setting up a main route, all the tracks in the main body of the route must be free and if part of the route is being used for the overlap of another route, the prior route must be sending trains in the same direction as the requested route. In xUML this is expressed so:

```
'/route_precondition_ok' :=
   ((not ('tvp'.'/overlap_locked' and 'route direction' \= 'tvp'.'direction') and
         'tvp'.'/element_DOS_available' if 'route'.'DOS_route');
     not ('tvp'.'/overlap_locked' and 'route direction' \= 'tvp'.'direction') and
        'tvp'.'/element_MR_available' otherwise);
```

If the route being requested is a DOS route, the conditions for the shared overlap remain, but some tracks in the main body of the route now may, or even must be occupied. Thus according to whether or not the requested route is a DOS route, which is indicated via 'route'.'DOS_route', the occupation condition to be evaluated can simply be toggled between '/element_DOS_available' and '/element_MR_available', while the other conditions are kept the same. The result is reduced redundancy, as no new class is needed, which in any event would have been virtually identical to the existing one. Equally, it is felt that the clarity of the model is in fact raised, as relevant information is kept together and not unnecessarily spread over many classes.

3.2.3.1.2 Constraint language

Various UML modelling concepts can be used to establish invariants – expressions that are always required to be true. However many tend not be modelled linguistically, so they will not be considered here, but in a subsequent document. Rather, here we shall deal with the explicit, textual conditions that prevent actions being executed, which are generally referred to as guards.

As can be seen above in Figure 7, it is possible to move between the states 'blocked' and 'not blocked' by virtue of the transitions between them. These transitions are triggered by events raised against the object, but whether or not the event can be carried out is determined by the guard. The rule behind the guard is generally an expression, built up in exactly the same manner as we saw in the section above.

In the example shown below in Figure 12, the cancellation of the release of a Swiss lockable device can be requested by raising the event '<c> cancel lockable device release' and this will be carried out, as long as the expression within the square brackets; ['/release_withdrawable'] is true. The definition of this is shown above in the assignment of a rule to an attribute on the state's initial transition.

When the device is *not* released, the guard controlling the transition to 'released' consists of a expression constructed in much the same way as we saw in the previous section (in fact it could be argued that the entire expression should be assigned to a derived attribute and called '/lockable device releasable' - the effect would be exactly the same).

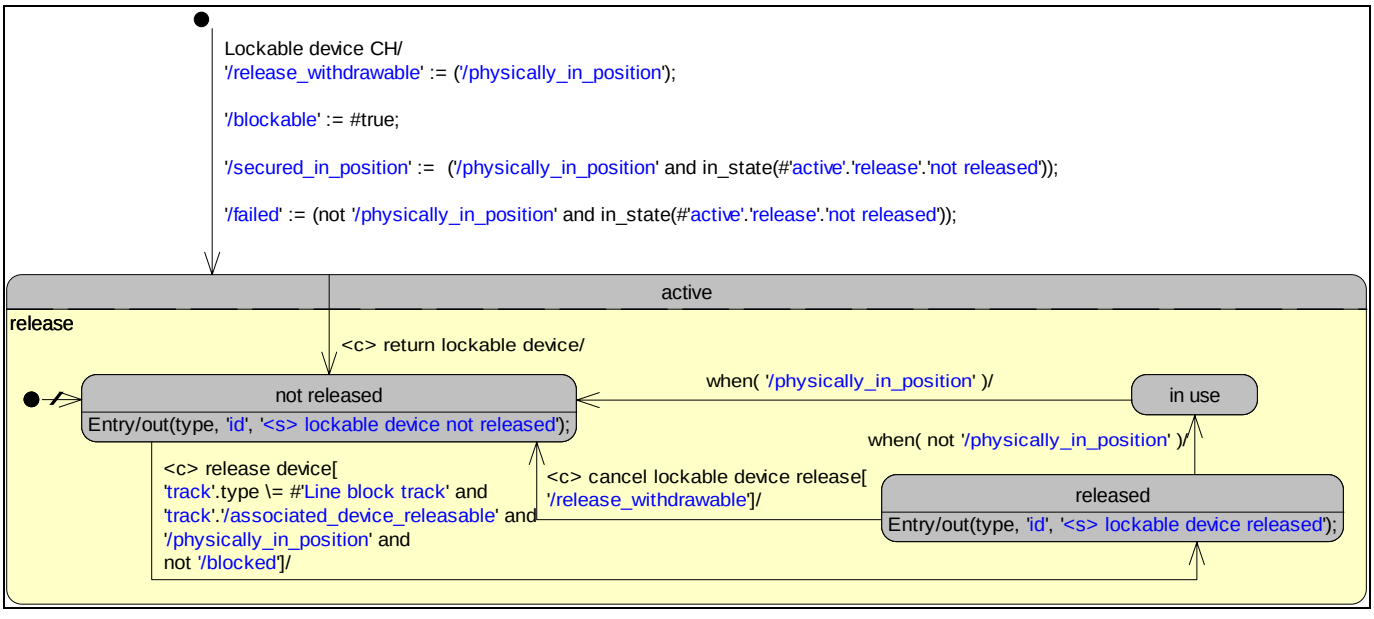

Figure 12 State Diagram of an SBB Lockable device.

The transition from the 'released' to 'in use' and from 'in use' to 'not released' also have conditions to them, but they are not placed in square brackets. The reason is that instead of a guard being evaluated once the event arrives at the object and the object is in the correct state to process it, now the guard is formulated as a 'when event', which means that it is always being evaluated and the transition will

happen as soon as the condition is fulfilled with no event needing to be sent. For this reason no explicit guard is needed because the event is the guard and the guard is the event. This modelling decision makes a simulation much more reactive and hugely raises the quality in the modelling of sequenced dependencies.

The last example state transition in Figure 12 is to the state ‘not released’ as a result of the event ‘<c> return lockable device’ being sent. This event has neither a guard nor indeed any constraint at all as to from which state of the ‘lockable device’ it can come; it is a global event that can arrive from the super state boundary, i.e. from any state. Thus whenever the event is raised on an object of this class, it will be executed.

3.2.3.1.3 Action language

The language examined up until now served mainly to specify the conditions under which something is true. However this alone would not enable the simulator to do very much. Thus in addition to the expression language, xUML has an action language that permits the actual functionality of the interlocking specification to be executed. Roughly speaking, the action language performs the positive functionality of the system, while the expressions and constraints express the safety aspects.

Speaking generally it is also tempting to say that whereas the expression language evaluates, the action language does; but as we have seen above, it is possible to include actions in the evaluation of an expression and we shall see below that it is also possible to evaluate within the context of an action.

In general only few internal transitions appear in the model and those shown in Figures 7, 12 and 13 are largely necessitated by the need to display the status of an object on the graphical user interface – which is interesting, but not strictly a functional requirement.

Figure 13 below provides a number of illustrative examples of actions.

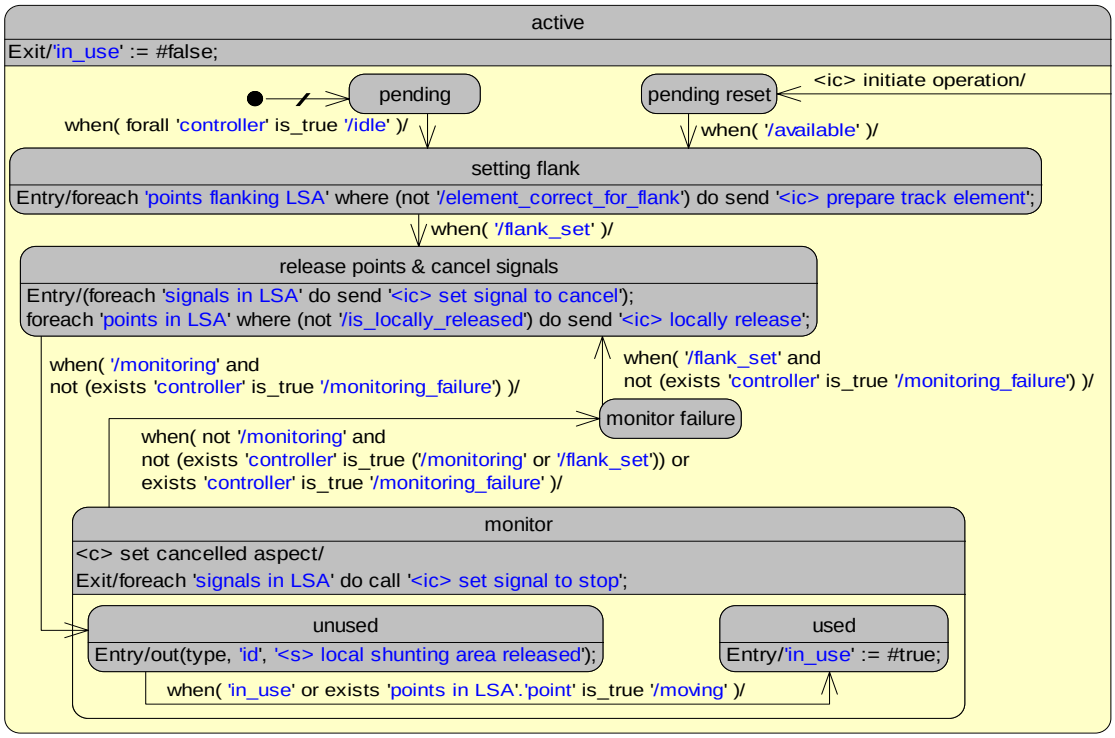

Figure 13 Part of the State Diagram for a local shunting area.

The first action we shall examine, ('in_use' := #false;), occurs at the level of the superstate ‘active’ and defines that whenever this state is left the value ‘false’ will be assigned to the attribute 'in_use'. The

mechanism by which this is done is exactly the same as the assignment of an expression to a derived attribute, namely ':='. The value 'false' will remain until the next time the object enters the superstate.

All the other actions in Figure 13 are variants on the following concept: for each X do Y. We saw that in the expression language the keyword for iterations is 'forall'; for actions it is 'foreach'. The sense is the same, except that instead of evaluating an expression in each target object, an action should be performed on each object.

There is a small variation on the exit event from the state 'monitor'. Here the action reads:

foreach 'signals in LSA' do call '<ic> set signal to stop';

Call and send are similar, but the difference lies in the fact that whereas a 'send' is an asynchronous action, which is simply processed like any other, a 'call' is synchronous, which requires that it and all its consequences are fully processed before any other event can be dealt with.

Another variant on the concept, foreach 'association' do send 'action'; is simply to write: send.'association'.'action'. There is no semantic difference between the two and it is merely a modelling principle that if the cardinality of the given association is more than one the 'foreach' construct will be used and if only one object is addressed by the action the latter can be used. It is hoped that consistency in even such trivial matters still helps raise the transparency of the model.

Although actions are generally plain commands to do something, it is also possible to evaluate whether or not an action itself should be performed on a transition. It must be stressed that this is not the same as a guard; it does not evaluate whether or not the transition should happen, merely whether or not the action on the transition should be executed. Such a situation is quite rare, but one occurs in connection with the functionality of the DOS routes and is shown in Figure 14 below.

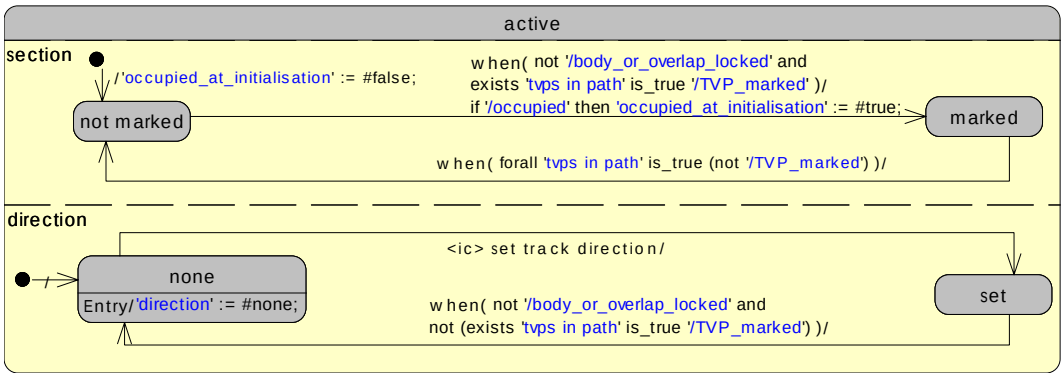

Figure 14 Part of the State Diagram for TVP section.

In the diagram it can be seen that the transition from the state 'not marked' to 'marked' is triggered by a when event. In addition as a result of this transition an action may or may not happen – based on criteria that are entirely different from those that determine whether or not the transition is allowed.

In the case of this action, only if the TVP section itself is '/occupied' at the moment that the event occurs will the attribute 'occupied_at_initialistation' be set to true. This construct could of course be avoided by having two transitions from the state 'not marked' to 'marked', in which the guard of one of them would check the occupation status of the track and accordingly assign the value 'true' to the attribute 'occupied_at_initialistation'. However, it is felt here that the conditions of the action are sufficiently clear as not to threaten the clarity of the model and thus it is possible to avoid adding extra transitions.

In addition to the commonly-used actions shown in Figure 13, there are a number of miscellaneous actions which appear in the model, and it is useful to explain them here. The 'out' event sends information to an external graphical interface and the 'say' event can be used to output any given text to

an external user log. Neither of the functionalities that these actions support is central to the SIL-4 safety kernel of the interlocking, but they help enhance the usefulness of the simulation.

We have also been able to cover the entire expression and action language of the GENERIS model by focusing on just a very few classes. The fact that this is possible demonstrates that only a narrow palette of syntax is needed for the specification of a complex system, providing that the chosen declarative syntax is sufficiently powerful. For more details on any of the syntax used here, please refer to the CASSANDRA/xUML User's Guide, which is available from Know Gravity inc. (http://www.knowgravity.com/).

Section 4 – CONCLUSIONS

We presented here a complete methodology for the rigorous textual expression of functional requirements. This methodology was successfully applied during the Euro-Interlocking project, and INESS will obviously benefit from its return of experience, allowing the capturing of requirements of a larger number of railways throughout Europe.

The next step involves using this methodology to in order to outline the shape of a common core of interlocking functional requirements taking into account the ERTMS environment.

Section 5 – BIBLIOGRAPHY

Elizabeth Hull, Ken Jackson, Jeremy Dick: *Requirements Engineering*, Springer, 2005

SELRED, Structured English Language for REquirements Development, Euro-Interlocking project, 2005

Using DOORS in the Requirements Development Process, Euro-Interlocking project, 2005

CASSANDRA/xUML User's Guide, KnowGravity Inc., 2004