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.1.5 – Report on translation of requirements from text to UML

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

Deliverable ID:	**D.D.1.5**
Deliverable Title:	Report on translation of requirements from text to UML
WP related:	D1
Responsible partner:	ProRail
Task/Deliverable leader Name:	UIC
Contributors:	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

DISCLAIMER

The information in this document is provided “as is”, and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

PROPRIETARY RIGHTS STATEMENT

This document contains information, which is proprietary to the INESS Consortium. Neither this document nor the information contained herein shall be used, duplicated or communicated by any means to any third party, in whole or in parts, except with prior written consent of the INESS consortium.

[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_WS D_Deliverable D.1.5_ 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	2009-05-27	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
0	2009-05-15	WS_Finalised	C. de Courcey-Bayley
1	27/05/2009	TAB comments taken into account	Khalid Agrou
SB_Finalised	29-05-2009		Richard VAUX / ALMA

TABLE OF CONTENTS

Section 1 – EXECUTIVE SUMMARY 4
Section 2 – INTRODUCTION....4
Section 3 – REPORT ON TRANSLATION OF REQUIREMENTS FROM TEXT TO UML 5
3.1 Overview....5
3.1.1 Object Oriented Modelling 5
3.1.2 The Unified Modelling Language....6
3.1.3 The modelling framework 7
3.1.4 The Analysis Process....8
3.1.5 The Development Process 9
3.1.6 Models and objects 10
3.2 Diagrams....10
3.2.1 Class Diagram....11
3.2.2 Class Properties 12
3.2.3 Package Diagram 15
3.2.4 Use Case Diagram 15
3.2.5 State Diagram 16
3.2.6 Sequence Diagram 17
3.3 Organising the model....17
3.3.1 Modelling use cases....17
3.3.2 Domain Object Modelling 19
3.3.3 Identifying Events....23
3.4 Behaviour modeling 24
3.4.1 Interaction Modelling 24
3.4.2 State Modelling....25
3.5 Other modelling aspects....27
3.5.1 Results....28
Section 4 – CONCLUSIONS....28
Section 5 – ANNEXES 29
Appendix A UML Notation 29
Appendix B Stereotypes 31

List of abbreviations[2]

DOORS Dynamic Object Oriented Requirements System

GENERIS GENEric Requirements for Interlocking System

HAL Hardware Abstraction Layer

LCL Logical Concepts Layer

SELRED Structured English Language for REquirements Development

UML Unified Modelling Language

Section 1 – EXECUTIVE SUMMARY

This document is the report on the translation of requirements from text to UML corresponding to deliverable DD1.5, as specified in the INESS Description of Work. It details the experience gained from the Euro-Interlocking project in the domain of functional requirements modelling.

The target audience for the document are consumers of the GENERIS model, i.e. interlocking engineers and developers interested in using the modelled requirements for further tasks such as transformations and validation. It is helpful if the reader has some familiarity of abstract modelling languages, although not necessarily of UML.

Section 2 – INTRODUCTION

This document serves as a guideline to show how the various diagrammatic aspects of the Unified Modelling Language have been used to represent the interlocking functional requirements in the Euro-Interlocking project.

In the first part, it presents the object oriented modelling paradigm, together with the reasons for the emergence of the Unified Modelling Language as a standard, and an architectural framework for technical systems modelling. Then, the analysis and development processes are explained.

[2] For the control/command and signaling terms, please refer to report DD1.1.

In the second part, the five best suited diagrams for functional requirements modelling are presented, with particular consideration given to those concepts used extensively in the Euro-Interlocking Generis model. This part contains also various examples extracted from the Generis model.

Finally, the last part is a methodological guideline explaining how one can benefit from the elements given in the previous sections in order to model the functional requirements in their static and dynamic aspects. The methodology clarifies the objectives and provides the steps to follow to carry out a proper modelling work. This is completed by illustrations from the Generis model.

Section 3 – REPORT ON TRANSLATION OF REQUIREMENTS FROM TEXT TO UML

3.1 Overview

3.1.1 Object Oriented Modelling

One of the most important basic concepts of object orientation is that objects in a control system are models of real-world objects. Object Orientation is a paradigm to decompose a system so that the individual components:

- represent things of the real world
- encapsulate data and related functionality
- are replicable, i.e. can be instantiated as often as necessary
- can be generalised and specialised.

The resulting components may (or may not) be subsequently implemented as (software) classes in an object-oriented programming language.

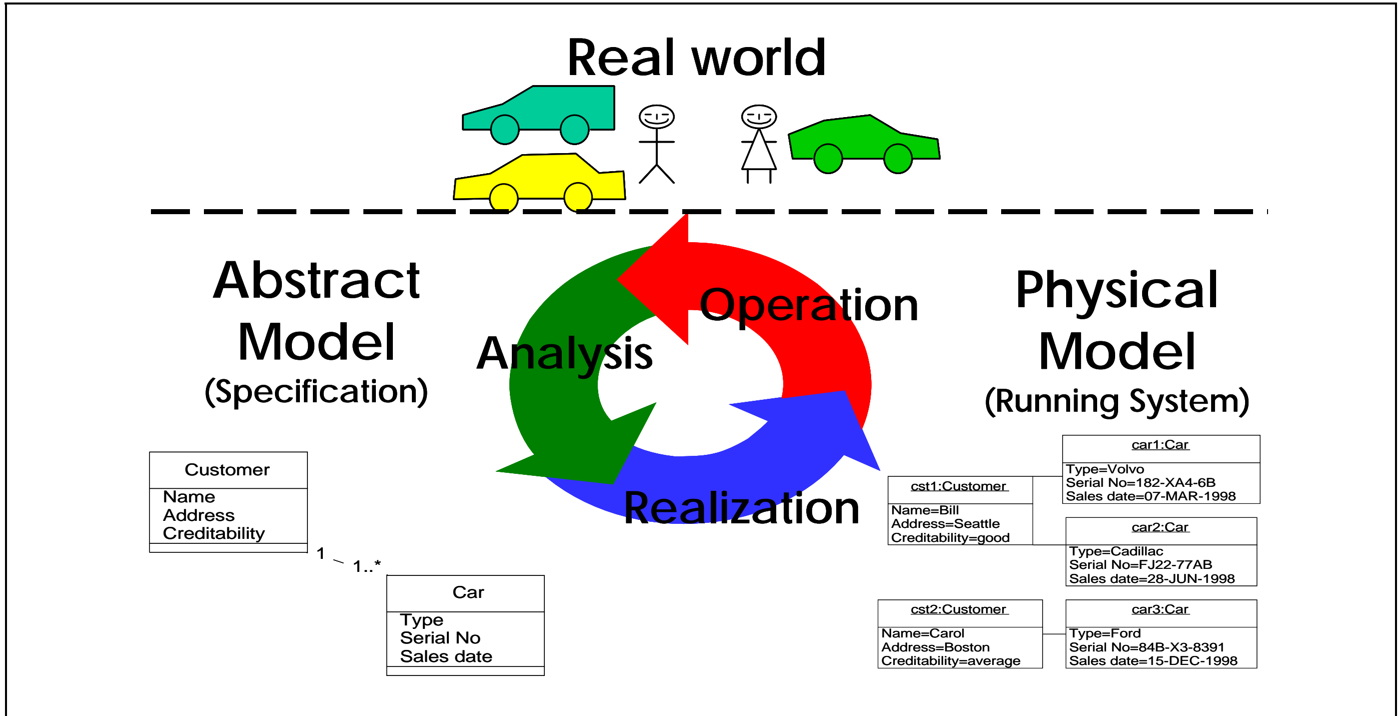

Figure 1: Object Models

Real life systems are often so complicated that they need to be simplified to be understood. Models represent a reductive and incomplete version of reality. However, rather than this constituting a weakness, by consciously omitting what is immaterial for a given perspective, models can be created which reveal phenomena not easily observed in the real world and which can yield unexpected results and relationships, leading to new ideas. A model is usually either descriptive or prescriptive. However, in the case on GENERIS, the model is both, as it reflects requirements which often present an optimised view of the current situation for each railway covered. This in itself does not matter, as long as it is clear which perspective a model is intended to represent.

The goal of modelling requirements according to 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 as classes (bottom left in Figure 1), together with information about their relationships and behaviour.

By running the realised system, 'live' instances of these classes are created (bottom right in Figure 1). Each instance of such a class corresponds exactly to one object of the real world, its 'original'. Finally, the instances are viewed and manipulated by the users of the system. In UML an important distinction exists between instances and the classes from which they come and it is perhaps worth mentioning here that as this document is mainly concerned with the diagrammatic aspects of UML, the issues concerning instances and the user-specific data with which they are configured will not be dealt with here, as this is a theme in its own right which will be handled fully in a subsequent INESS document on testing and validation (**DD.4.2.1** Documented methods for expressing test cases in UML).

3.1.2 The Unified Modelling Language

The Unified Modelling Language (UML) was developed from 1995 as a consequence of a situation in which there was no consensus as to how to develop and represent object-oriented software systems. 1997, UML was adopted as a standard by the Object Management Group (OMG), which is the largest

and most important non-profit organization that focuses on the establishment of vendor-independent standards for the software industry. Following its publication in 1997, this long-awaited standard was rapidly adopted by the software industry worldwide. Since then, several versions and updates of standard "UML" have been published, of which Version 2.0, adopted by the OMG in 2003 was the most important.

It is important to mention 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 or elaborate those artefacts. UML just defines the syntax and the semantics of various modelling elements that should be used to document an object-oriented system.

Although UML has been developed by the software engineering community, it is important to note that it is not limited to modelling software systems. It is a generally applicable modelling language to represent the functionality of a system, based on the object-oriented paradigm. Today it supports various application areas such as business engineering, specification of heterogeneous systems, software requirements modelling, software architecture design, agent technology or hardware design.

3.1.3 The modelling framework

To assist in organising an interlocking requirements model, an architectural framework based on the general framework for technical systems is presented in Figure . The interlocking system controls the trains via track elements in the field, such as points, tracks and signals. These elements are controlled by signallers via the interlocking system. The interlocking system itself is structured into the following three layers:

- **Hardware Abstraction Layer (HAL)**
  Abstracts the track elements into classes that encapsulate their safety-relevant functionality.
- **Logical Concepts Layer (LCL)**
  Contains classes that abstract the logical concepts of an interlocking system such as routes, flank protection, approach areas etc. Its functions are enabled by the existence of the classes identified in the HAL.
- **Operating Layer**
  Represents the black box view perceived by the operator. This black box view is mainly represented as a set of use cases of the interlocking system. The model contains no traffic management layer, which is usually to be found in a real interlocking implementation, as it is not central to the safety-core of the interlocking as defined in the scope of the project.

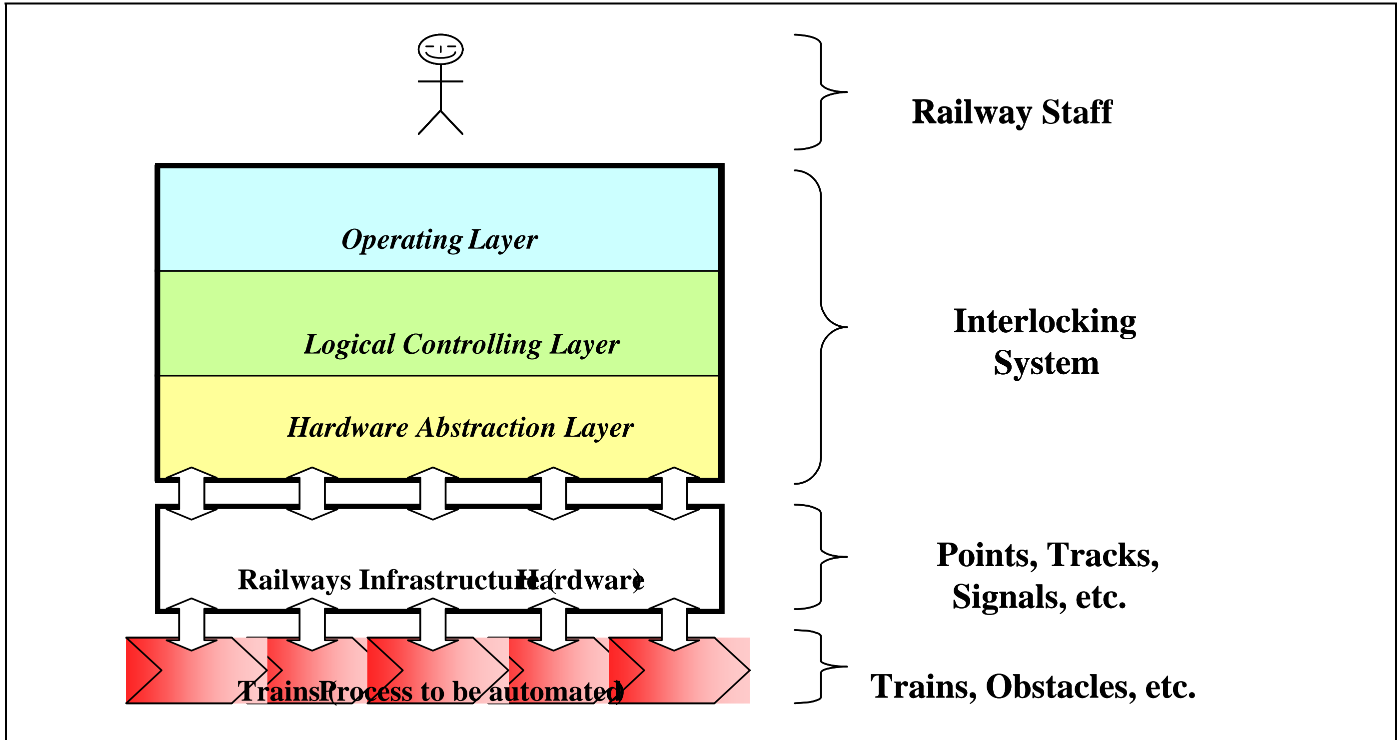

Figure 2: The Modelling Framework

3.1.4 The Analysis Process

The primary goal of the Analysis Process is to extract domain knowledge (Facts) from the domain experts by means of suitable techniques (Fact Finding). These facts must then be precisely documented (Fact Representation) which in turn may raise domain questions that must be clarified with the domain specialists (Fact Validation).

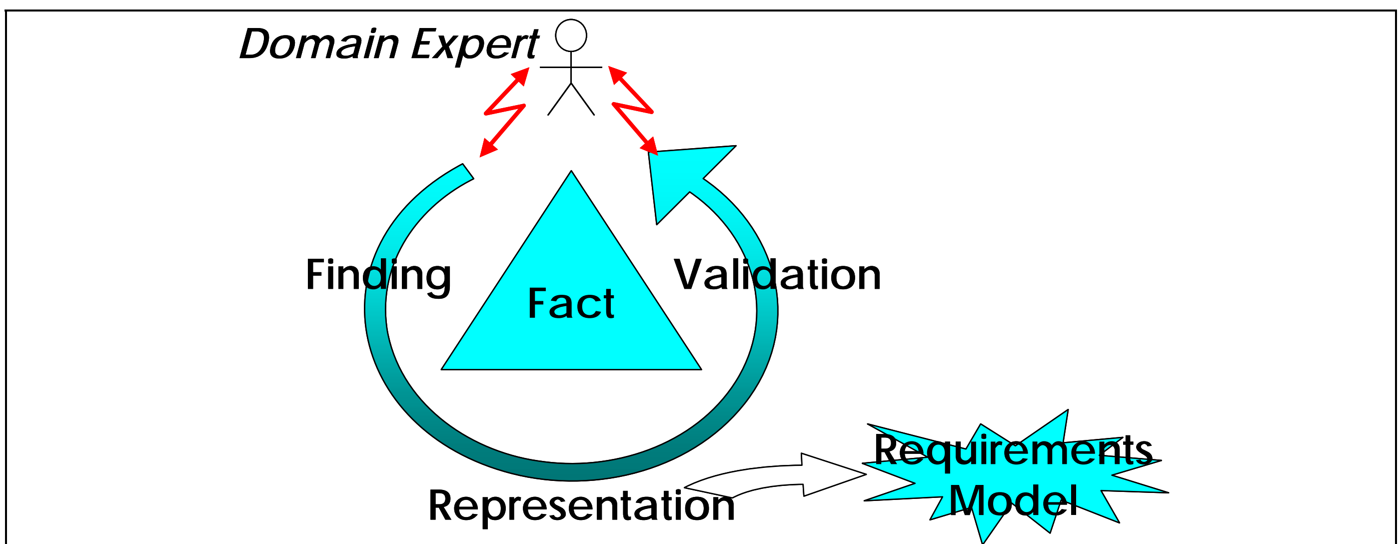

Figure 3: The Analysis Process

UML only provides the means for Fact Representation, by allowing the formal representation of such facts. For Fact Finding and Fact Verification, specific skills and engineering know how are needed to elaborate the contents of the requirements model to be developed. In the Euro-Interlocking project, this was undertaken by dedicated signal engineers conducting interviews with each country's domain experts and capturing the resulting functionality in a DOORS database. In turn, this database forms the input for the modelling process. During modelling there was no fact elicitation between the modeller and the signallers of the various countries, as it had been decided that the DOORS database had to be the sole source of functionality – thus promoting consistency between the textual and modelled requirements.

3.1.5 The Development Process

3.1.5.1 Context

The outcome of the fact-finding elicitation process illustrated above was that the system requirements were established and recorded in DOORS. These were subsequently analysed in order to be expressed in UML. In so doing the DOORS artefact was fundamentally considered as a basis to establish the system boundaries, terms and requirements.

- **System Boundary**

  When analysing complex functionality for a system with many interfaces, it is of paramount importance to establish, as early as possible, what the scope of the system to be examined actually is. This involves defining what is internal and what external to the system, what is controlled by it directly, indirectly or not at all. The assumptions made during this process need be valid for the entire project or else re-documented as they change over time.

- **Terms**

  Terms are important for interlocking systems and form the foundation on which any further requirements are built. These include nouns (concepts) as well as verbs (activities) and adjectives (properties). When a railway entered the project, before its requirements could be captured, it was important to map the terms used in their domain to the Euro-Interlocking glossary (or to create new glossary entries where this was not possible) in order to establish that the concepts on which the requirements are based were in fact common.

- **Operational Requirements**

  Requirements that describe the desired functionality of the interlocking system. In other words: they define the "positive" behaviour of the interlocking system as observed from outside the system. These too have been captured and tagged so that the current GENERIS model contains the HAL functionality of a number of different railways.

- **Safety Functions**

  These are not safety constraints in the formal sense of invariants, but rather the functional safety aspects which have been implemented in the guards that restrict certain functions at certain times, varying on the state of the system.

UML diagram types and system analysis

Once the system boundary was set, these three broad types of requirements were initially formulated textually according to the SELRED rules. However, to become more precise, more formality is required and this is where UML diagrams can refine the textual requirements. The syntax of this transformation is covered in the complementary requirements expression document on the UML language used in GENERIS; the diagrammatic aspects will be dealt with in what follows.

In the operational requirements it is mainly the verb terms that are used to derive use cases which represent the black box view of the interlocking system. Important use cases were identified and then further refined into sequence diagrams that illustrate the black box interactions between interlocking users and the interlocking system as well as between the interlocking system and the railways infrastructure to be controlled. This process has up until now, not progressed beyond listing the various events crossing the user interface which are associated with the given use case on a sequence diagram – i.e. no work has been done to show the chain of resulting system behaviour that follow such inputs, however it would be hoped to build on this work and extend it in INESS, as this would be of central importance to the validation work, once a more comprehensive test environment is available. Thus in the GENERIS model, although sequence diagram activity between the system boundary and the system is executable, that within the system is merely illustrative, and neither executable nor verifiable.

3.1.6 Models and objects

Most of what follows deals the specification model, but it is important not to forget the simulation, which is based on the given model. While the model is a static representation of the given system, the simulation is its living realisation, which is created by bringing together instances of the model's classes and the data with which it is to be configured.

This distinction is important and there is even a separate vocabulary in UML which differentiates the world of classes from that of living objects (instances). This document is primarily concerned with the class model, but it is worth briefly discussing instances. They are based on the classes of the model and are infinitely replicable; they have all the properties of the class from which they are created, but the data values of these properties may well vary from instance to instance as determined by the configuration values. These differences are what makes the model specific and are also in fact what make it realistic and interesting to the domain users, but given the virtually infinite number of permutations of data configuration settings that can be instantiated, they also restrict the possible completeness of any model validation work, making it harder to prove that the model will act correctly in all circumstances. However in the case of the INESS model, this restriction does not matter, as no specific application needs to be validated, but merely the specification on which it would be based. This means that once the specification has proved that a given rule is guaranteed, no further testing of that function is required. Nonetheless, the interaction of the model with the configuration data means that the Data Preparation work is, at least indirectly, part of the requirements. This matter will be examined in detail during work in the INESS project concerning the modelling of the validation requirements specification.

3.2 Diagrams

UML is mainly a graphical language. It has a set of thirteen diagram types, each of which represents a specific view of the system to be represented. In the context of interlocking systems, the following diagrams are considered as most important:

- **Use Case Diagram**
  Represents the functionality of the system from a black box perspective.
- **Class Diagram**
  Represents the (glass box) decomposition of the system into its (logical) components (i.e. classes) as well as their associations and properties.
- **State Diagram**
  Precisely specifies the behaviour of an individual class in terms of its states and the transitions between them.

- **Sequence Diagram**
  Illustrates the interactions between two or more objects which can be (sub-)systems, classes, or users.

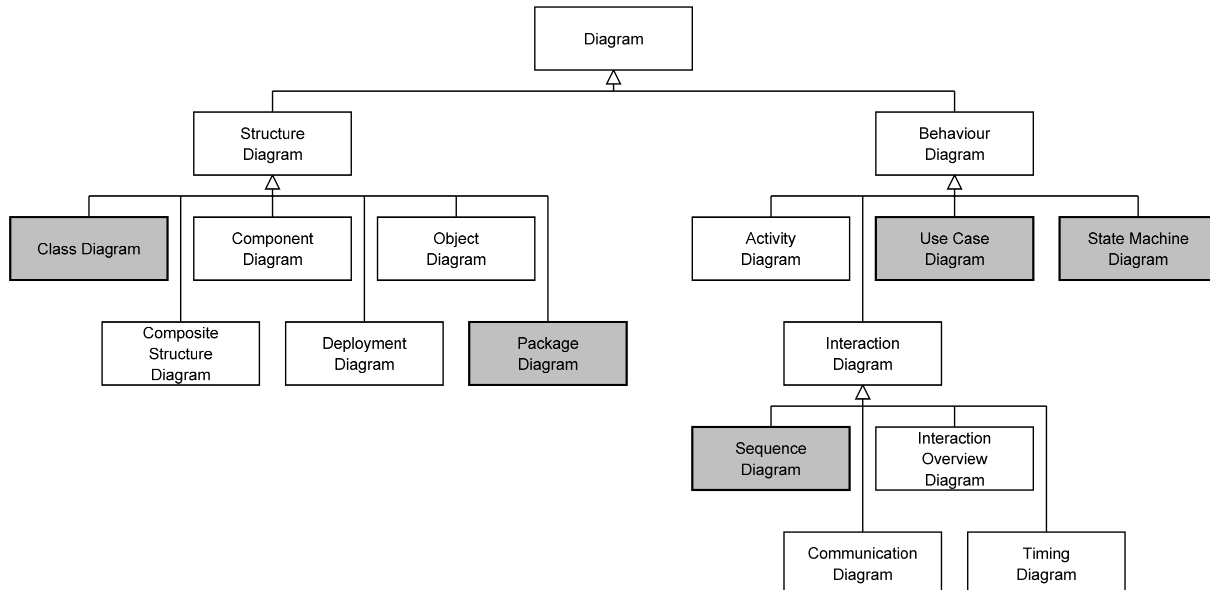

Figure 4: An overview of the UML Diagram types

The diagrams described above are sufficient to represent the requirements of an interlocking system in an implementation-independent way. In the following section, each diagram will be discussed in as far as is relevant for its influence on the expression of the requirements. Other diagrams of the UML represent aspects of the actual implementation of an object-oriented system, which is beyond the context of this document. A brief summary of the UML notation can be found in Appendix A of this guideline.

Structural Diagrams

3.2.1 Class Diagram

We have noted that the domain object model is derived from the terms identified and defined in the requirements. In general domain objects tend to be noun terms, and their attributes and associations tend to derive from adjectival terms. The individual domain objects can be represented in UML via classes. Classes themselves are abstractions of entities that encapsulate a delineated part of system and contain properties that are, in general, unique to it. Grouping classes together in a diagram enables the static decomposition of the system to be shown in terms of its various defined components (i.e. the classes themselves) as well as these links (associations) and properties (attributes).

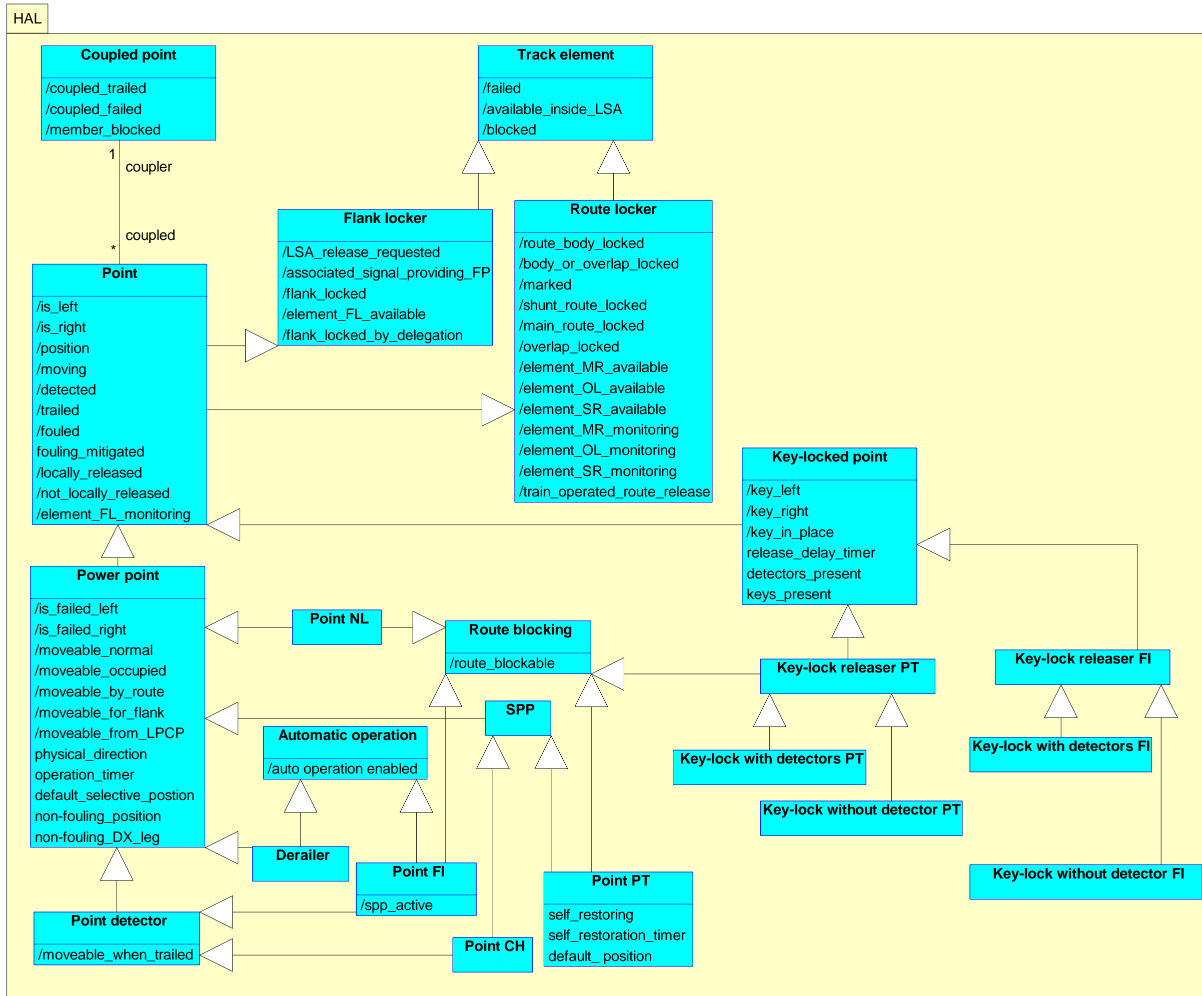

Figure 5: The sub-types of "point".

3.2.2 Class Properties

3.2.2.1 Associations / Roles

The class diagram is perhaps one of the most powerful graphical expressions of the requirements. By illustrating the architecture for a system, this diagram type allows the permissible relationships between the logical components defined in the system to be shown. This is achieved via associations which set out the relationships between the system's defined entities. All associations in the model are driven by the requirements. This diagram type allows it to be specified, for example, that a point 'knows' the track section in which it appears. Furthermore, this concept can also be used to quantify the number of particular objects of a given type that another object can be linked to. For example whereas a lockable device belongs to at most one track section, a track section can have an unlimited number of lockable devices associated to it. This concept can very powerfully and simply constrain the model to fulfil the requirements. Simply put, any given object that is compelled by the requirements to know about the status, or influence the behaviour of another object needs a defined relationship to that object. This does not need to be a direct link, but if the model is to remain clear, it is often best if it is. Once this link exists, it is a matter of the data configuration as to how it is populated (i.e. which particular instances are linked

to one another), but no relationship will be allowed that contradicts the associations and multiplicities declared on the class diagram. Thus the dependencies between the various modules in DOORS can be formally specified in the model without any text being used whatsoever.

In the GENERIS model, it is the general philosophy that each end of a given association between two objects be named; in UML this is called the ‘role’. The reason for this is it enhances both the clarity of the model and enables the syntax of the expressions to more closely follow the linguistic style of the textual requirements. This concept was analysed in detail in the requirements expression document. A few further features of roles as used in the GENERIS model are given here:

- All roles can be navigated in both direction
- If no role name is present, the target class name can be used as its proxy
- Roles must be distinct if more than one association binds two classes
- Role multiplicities can be used as a constraint on an association

By stating that all roles are bi-directionally navigable, it is implicit that no objects are stored in attributes on an instance. This would both deviate from the nature of a declarative model, in which relationships are graphically visible and would also have the consequence that whereas the object on which the attribute appears would know the target object, the opposite would not be true, and thus there would exist only a uni-directional relationship between the two entities.

3.2.2.2. Inheritance: Generalisation/Specialisation

The second fundamental feature of classes is their ability to inherit properties from one another. This is achieved via the generalisation / specialisation relationship. By virtue of this it is possible to specify facts which are common to a number of objects only once and have each object “inherit” these properties as need be. This is a powerful mechanism for increasing the clarity of a model and for avoiding redundancy and the errors to which it generally leads. A few further features of generalisation / specialisation as used in the GENERIS model are:

- All inheritances are disjoint and there is no class transformation
- No abstract classes (leaves) are ever instantiated
- An instance is a union of all the state compartment ‘slices’ of its own and its parent classes
- All the slices’ state machines of an instance run in parallel
- Diamond inheritances are correctly supported.

By disjoint, it is understood that no overlaps are permitted between the instances of sub-classes. To take an example from Figure 5, an instance of a track element may be a flank locker or a route locker, but it cannot be both and nor can it change from one to another while keeping the same identity. i.e. flank lockers and route lockers are separate entities. By stating that there is no class transformation, it is meant that an instance is not dynamic and thus its properties are persistent for its life-cycle.

By stating that no abstract classes are ever instantiated, it is meant that objects are only ever created from class ‘leaves’ (sub-types). In the case illustrated in Figure 6, this would mean that although the class key-locked point is entirely valid, it would never be instantiated without a national type being chosen (Finland for FI and Portugal for PT). In addition, both countries know two types of key-locked point and either type needs to be chosen to create a valid instance. This means implicitly that the top three classes would never be instantiated on their own, as they do not incorporate the complete functionality of the track element and thus a valid key-locked point object would always also have to include one of the four classes at the bottom of the diagram.

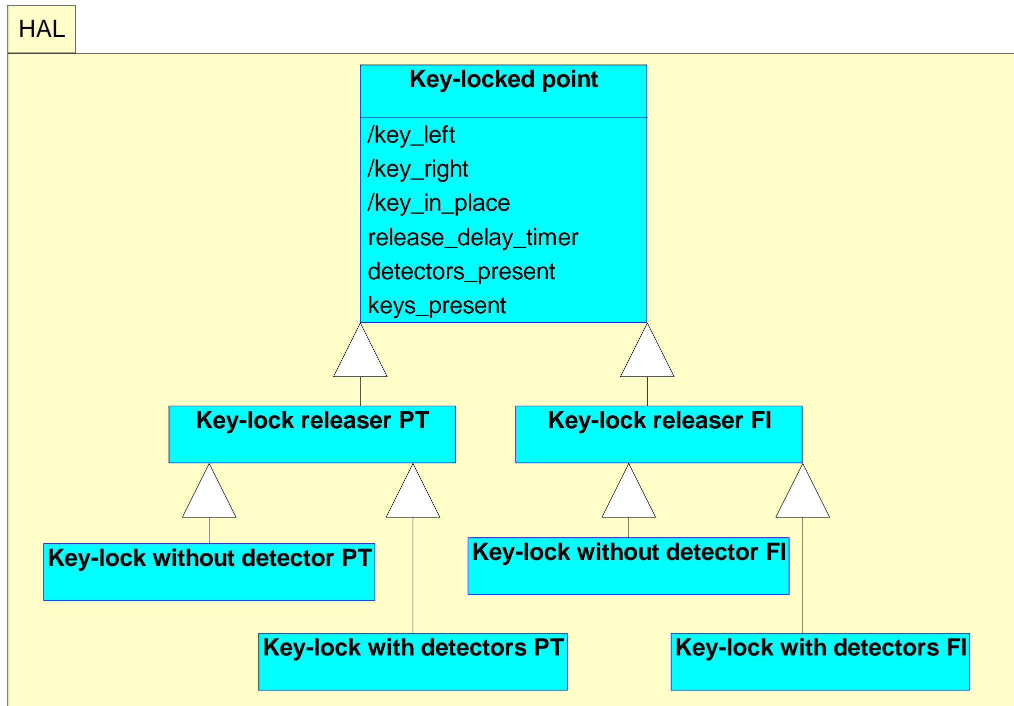

Figure 6: The Key-locked point and its sub-types.

When classes with a diamond form of inheritance (see the example of Point CH and Power point in Figure 5) are instantiated only one instance of the state machine in the super class will be created. Expressed in terms of the cited example from Figure 5, despite the 'double inheritance' of the class Power Point via both SPP and Point Detector an instance of Point CH will only have one instance of the slices from the class Power point.

3.2.2.3 Attributes

An attribute is a property of a class, which enables data about a given instance to be stored and referred to in the execution of the functionality of the class's state machine. In GENERIS many attributes are used for defining the data constants of the model, such as timers or route types, or else for variables such as the direction of a track in a route. However, UML also offers an attribute type, the value of which can be calculated from other attributes or states in the same or even other classes. This is the derived attribute and it is designated in UML by a leading slash symbol, e.g. '/failed'.

Such derived attributes have been extensively in the Generis model (see Figure 5), especially to provide information to both the system (and the glass box user) in a manner as close to natural English as possible, without compromising the formality of the model. Many derived attributes' expressions are based on the values of other derived attributes, but in general most are simple Boolean derivations. However, occasionally more complex rule sets are used, which enable complex functionality to be invoked in the minimum of space and complexity. For more details on the expression language used in the attributes of the GENERIS model see the requirements expression document.

Attributes can also be ascribed differing levels of visibility, according to whether or not it is desired for their value to be accessible via associations (in which case they are 'public'), via generalisation / specialisation relationships (for which they are declared 'protected') or not at all to any external class; ('private'). While in object-oriented design, it is generally preferred to keep attributes private to control access to the data, in the GENERIS model the derived attributes are often used to define the interface between objects; the place at which private class-relevant information is made available in a declarative manner and thus many derived attributes have been made public.

In addition, these further points about attributes in the GENERIS model are worth noting:

- The multiplicity of the attributes is always 1
- Declared attributes are static for the given instance over its entire life-cycle
- No data types are used

That the GENERIS model has no data types is largely the result of the fact that their inclusion would have added little in the context for which the model was to be used in the Euro-Interlocking project. Given that the INESS project foresees undertaking formal verification, data types could be included in the INESS model, if it can be shown that there is some validity in terms of the verification work of, say proving that a Boolean derived attribute has indeed been assigned a Boolean value.

To state that the multiplicity of a declared attribute is 1 means that it should not be 'nullable' (which is not to say that it cannot have the explicit value 'none' or 'null', but rather to state that it should not be empty). However, as the simulation environment does not currently enforce the lower end of a attribute multiplicity, and the fact that this was anyway not central to the scope of the GENERIS model, little effort was spent to check the correctness of attribute population in the model at instantiation. Again, if the validation and verification work were to show that this would be of interest, its enforcement could be considered in the elaboration of the INESS model.

By stating that attributes are static over their life time, this does not mean that their values will not change, but just as objects will not be transformed, so the rules behind the apportionment of the values of attributes will not change once the given object has been instantiated.

3.2.3 Package Diagram

There is no special functionality associated with the diagram type, but given the complexity of the GENERIS system specification, its use helps to show the various relationships of blocks of related classes. Figure 5 is an example of this form of diagram, created to show that various types of points that occur in the model.

Behavioural Diagrams

3.2.4 Use Case Diagram

Diagrams of Use Cases represent the functionality of the system from a black box perspective. For the scope of the GENERIS project, this diagram type was only used to constrain interface events to particular users (Actors in the UML terminology, which can be considered as a special type of class). The distinction between whether or not the event can be raised on a particular object and who is authorised to raise it is an important one, worth describing in an example. Both a signaller and an local operator in the field have permission to move a set of points, but not at the same time; as this permission is carefully allocated. However by allocating each an event with which they can move the point and defining that the event is accessible to an actor via a particular use case, the signaller will not have access to the local operator's event and vice versa. Thus the system's core does not need to know about the allocation of external permissions (which are anyway beyond its scope), as it is dealt with at the system interface. The net effect of this is that certain written requirements can be implemented at the boundary to the system by simply allocating the use of certain commands to certain users. Thus desired functionality can be achieved while also safeguarding certain model invariants – i.e. that given functionality should only be available to a given user in certain circumstances, or indeed not at all, without the use of any textual constraint. However, it should be noted that merely having the event available does not necessarily mean it will be executed and that brings us to the next diagram type, which has been used to capture most of the complex functionality of the system.

3.2.5 State Diagram

This diagram type precisely specifies the behaviour of an individual class in terms of its states. It provides a dynamic view of the object under examination, through the use of events, states, state transitions, guards, and actions. State diagrams will be used in the INESS model to represent the functionality of the interlocking specification and thus they are correspondingly complex. Each identified class will be given a state diagram in which the functionality most appropriate for that element will be placed. In general the name of the class guides and the attributes of the class provide the input for the functionality for the state machine, but the actual 'positive' functions of the class are specified in the actions of the state diagram itself. The main feature of this diagram type is in the states and the transitions between them. The initial state transition occurs by virtue of the birth event, which causes the initialisation of an object of that class as well as the first transition from the birth state. What happens subsequent to that is entirely dependent on the functionality of the object in question and can be read in the various examples of this diagram type that follow.

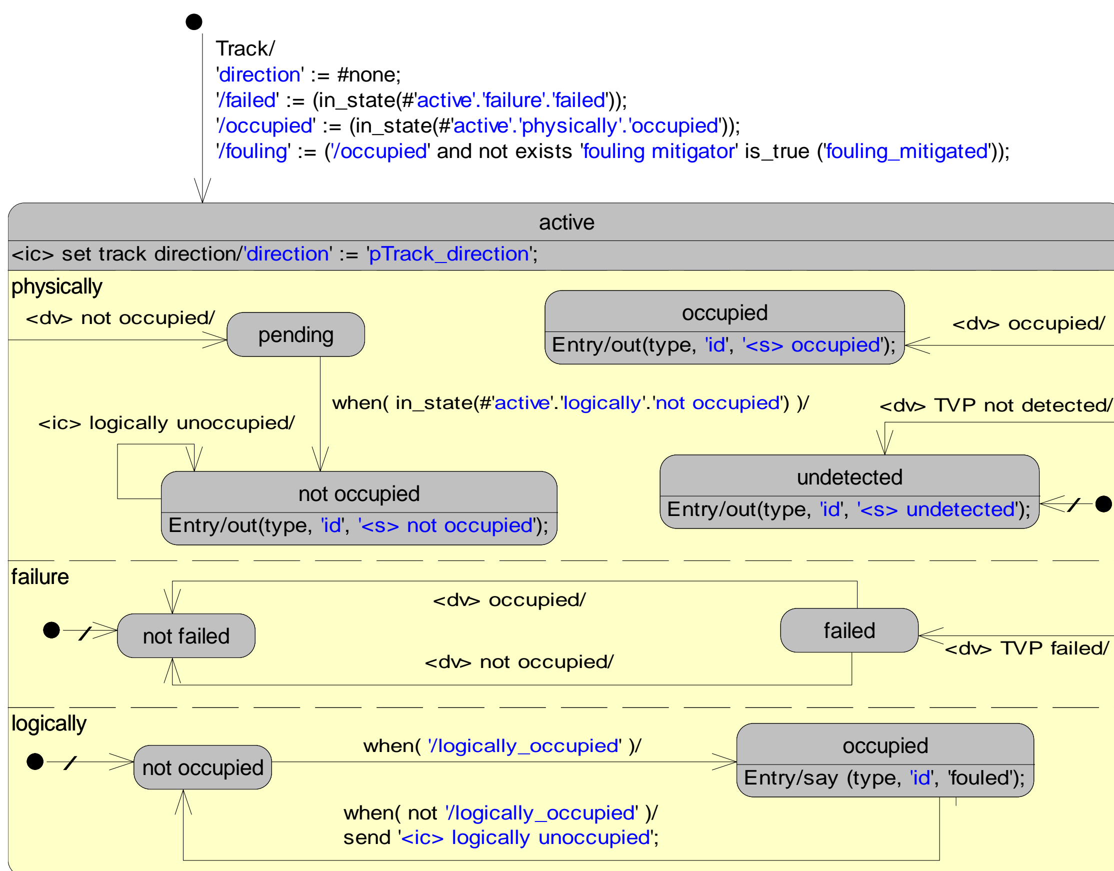

Figure 7: The state diagram of a track section.

The various transitions between the states are triggered by events – the nature of which can be various. An event may arise as the result of an input from an external (see the example of '<dv> occupied' in Figure 7 above) or internal user (not used in GENERIS), or from the system itself ('<ic> logically occupied' in Figure 7). It can be triggered by the passage of a defined period of time or even by certain predefined criteria in this (any when event on the same diagram), or any other state. State transitions are usually undertaken for a purpose, something discrete to the object itself, such as a change in its status. However quite often this status change is also the trigger for an action to be undertaken as a result of this transition.

Generally speaking, an event will be handled by those states which are in a position to receive it at the time it was raised. However, as an event may also be of interest to the entire state machine, by modelling it on the super state it can be handled globally. All the examples of events with the stereotype <dv> in Figure 7 are such events; because if something is reported from the field it needs to be acted upon without condition.

Although simple objects tend to have statuses that are mutually exclusive (‘locked’ or ‘not locked’ for example), certain other objects may be more complicated and have state properties that exist in parallel. This information can also be captured via a state diagram by the use of state compartments. The state diagram for the Track Section in Figure 7 has three of them.

3.2.6 Sequence Diagram

Strictly speaking in the UML, a sequence diagram shows both the interaction and sequence of given messages between the various instances of a system; giving a dynamic view of the area under examination. In the context of the GENERIS project however, this form of decomposition is of little benefit, as the ‘system’ under investigation is a black box specification and not an implementation; thus in this context this diagram type has a different purpose. A sequence diagram is defined for each use case that serves to identify the events (messages) available to that use case, as mentioned in chapter 3.2.4. This has the effect of defining the scope of each actor’s interaction with the system and can also illustrate the interactions between two or more partners which can be instances of (sub-) systems, classes, or users.

The diagrams described here are quite sufficient to represent the requirements of an interlocking system in an implementation-independent way. Other diagrams of UML represent aspects of the actual implementation of an object-oriented system, but this is out of the context of this project. Now, we shall look at the methods for applying each of these diagram types.

3.3 Organising the model

Objectives

The main objectives of this activity are:

- Defining the boundaries of the system to be modelled.
- Making the model navigable, maintainable and scalable by separating concerns into different packages
- Use as many class diagrams as necessary to illustrate the system context

3.3.1 Modelling use cases

Objectives

The main objectives of use case modelling are:

- Identification of the main functionalities of the system seen from a black box perspective
- Identification of the main actors outside the system and their relationship to the identified functionalities

- Understanding the main functionalities from a user's perspective and to identify important terms.

In general, it can be stated that a use case is typically performed by human actors pursuing a certain goal which is to be achieved within a reasonably short time period. However, as mentioned above, the GENERIS use cases are not particularly 'goal-oriented'. In the INESS project, this should probably be approached differently.

Elaboration

The steps in the creation of Use Case Diagrams can consist of defining:

- Flow of activities among controllers (typically humans)
- Flow of activities within the system to be controlled
- To identify the events which are to cross the system boundaries (both commands and statuses)
- Identify actors
- Identify and describe basic use cases
- Factor-out common fragments
- Identify extensions and specialisations between use cases as necessary
- Consolidate
- Model black box sequence diagrams (see chapter **Erreur ! Source du renvoi introuvable.**)

3.3.1.1 Example

The following diagram shows the use cases of the Generis model. There are two actors, "local signaller", who has the commands of a normal train dispatcher, and "tester", who is able to do everything which the "local signaller" can do and in addition can interact directly with the status of the physical elements in the field in order to produce disruptive situations to check that the model behaves properly when faced with the degraded performance of elements *beyond* its context. No other actors, such as "train" have been defined, as the tester is able to act as 100% proxy for it.

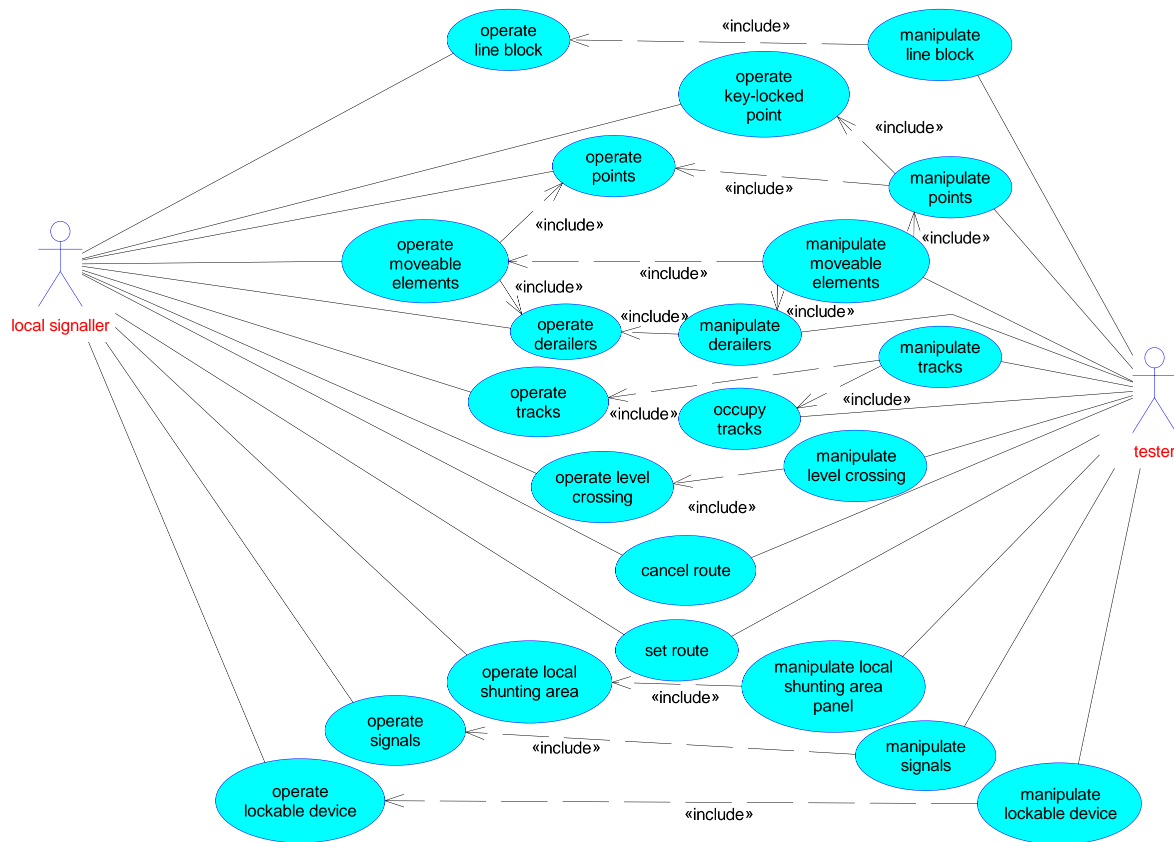

Figure 8: The Use Case Model

3.3.2 Domain Object Modelling

3.3.2.1 General

The Domain Model identifies the concepts of the interlocking system, along with their significance and describes the interactions and dependencies between these concepts. The creation of the Domain Model generally promotes a common understanding of the context of the system. In addition it serves an important basis for the data to be administered by the system as well as its rules. These rules are often also cited in various of the Use Cases, as well as the user interfaces. The Domain object needs to be constantly updated over the life cycle of the model, so that it reflects changes in the scope or the context of a given project, thus allowing inconsistencies to be easily identified. The creation of the Domain Model needs to conform to the context of the system as defined in the textual requirements and ensures that the scope of the model is always synchronized with the that of the domain expert providing the requirements.

Objectives

The main objectives of domain object modelling are:

- Identification of the physical “things” to be directly controlled by the system
- Identification of the abstraction of the physical “things” controlled by the system
- Identification of the logical “concepts” to be abstracted in the system

- Understand the associations among the "things" and "concepts"
- Establishing a precise definition of important terms as a solid foundation for further modelling.

Elaboration

Perform the following steps to elaborate the domain object model, starting from the modules as presented in the DOORS database:

- Capture and classify the physical objects of the real world
- Identify LCL objects from logical objects in the real world
- Model associations with cardinalities among objects
- Identify generalisations of objects and any model specialisations
- Consolidate

It is generally very helpful to the identification of the domain objects of both the HAL and the LCL that the DOORS modules have already been conceived in a manner that closely mirrors the classes to be used in the model. It is helpful to organise the domain objects on more than one class diagram; this is especially true once the number of classes begins to explode. In the GENERIS model the approach taken has been to have diagrams:

- showing the associations between the super classes of the HAL and the various corresponding classes of the LCL (see Figure 9).
- showing the breakdown of the LCL layer (see Figure 10)
- showing the super classes of the LCL layer
- showing the super classes of the HAL layer
- showing the sub-types of one type of physical element - point or signal etc. (see Figure 5)
- showing individual LCL concepts such as route, local shunting area etc.

The following class diagram shows an overview of the interlocking System in terms of the abstract classes of the physical elements and the logic classes for dealing with the route concepts and the multiplicities between them.

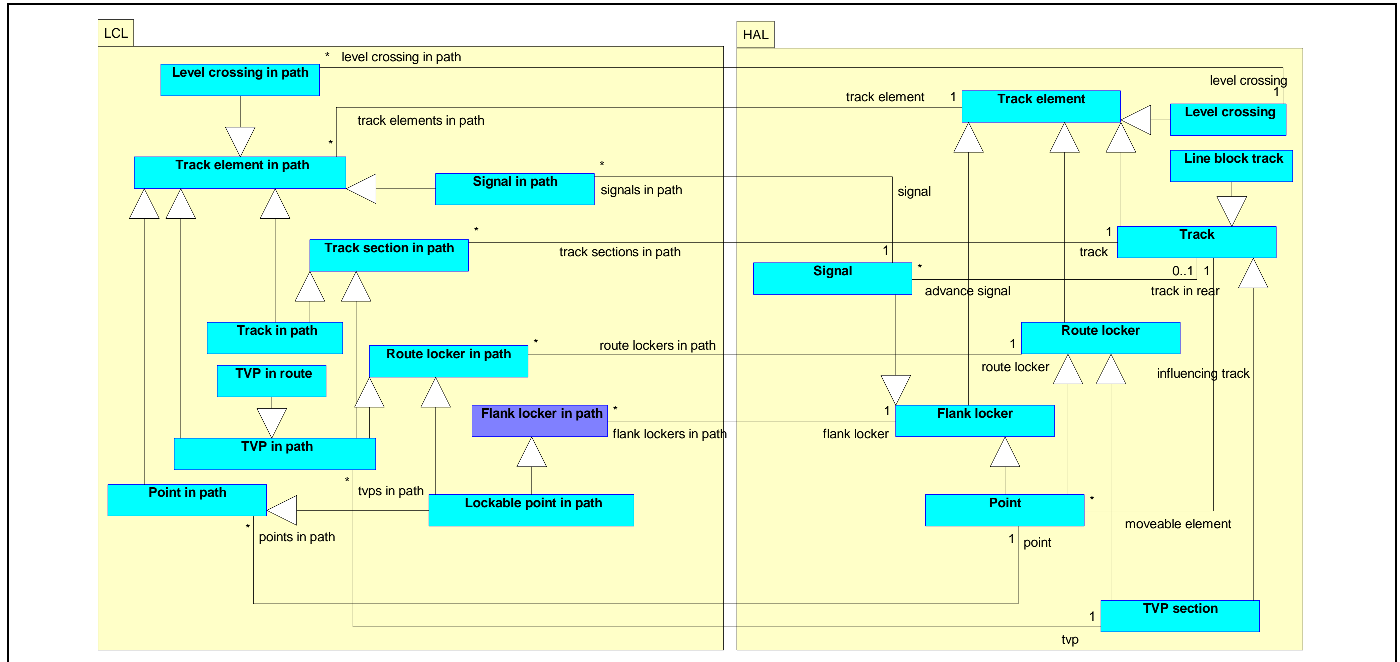

Figure 9: The Domain Object Model

The diagram above shows a view of the entire architecture of the system, but at a very high level. Only the super class of each type is shown, in order to give a rough idea of the system. Having a diagram at such a low level of detail can still be useful, as when the number of classes is high, even restricted class diagrams can quickly become complex and choices need to be made between comprehensibility and usability.

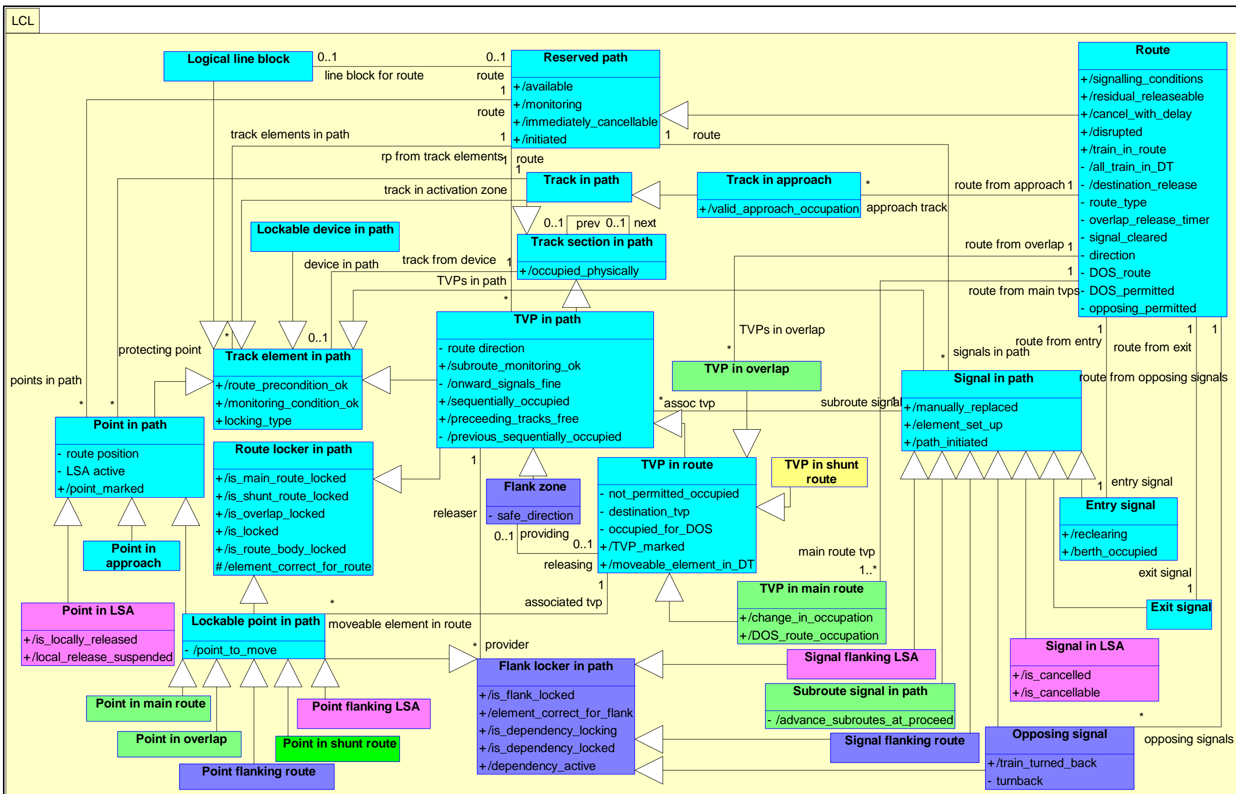

Figure 10: The Logical Concepts Layer, showing most of the generic and railway-specific sub-types.

The above figure shows part of the previous figure, but in more detail. This concept can also be used for breakdown the presentation of the functionality of physical elements. Again, this is recommendable once the number of classes has expanded to degree that they cannot feasibly be presented on one single diagram.

3.3.2.2 Generalisation / Specialisation

All of the objects to be identified will already be represented somehow in the textual requirements, however, there may also be common functionality across a number of these objects. In such cases functionality can be factored out and presented in a super class. The process by which this is done is basically one of constant iteration and re-factoring. The basic assumption to be made is that things are always different from one another until it can be proved that they are in fact not, at which point the commonalities between them can be extracted and remodelled in a common super class. It is better to assume that things are different and then bring commonalities together than vice versa – i.e. assuming everything is the same and then to look for differences. This is because it is better to compare two class which have been created to fulfil a particular purpose and then see that there is in fact redundancy between them which can be factored out, than to endlessly tweak a single class that is being forced to behave as what are in fact two separate entities, in the hope that some sort of compromise between the two is possible.

The motivation that drives abstracting out common functionality is to avoid redundancy and with it comes the hope that although there will be a larger final amount of classes when the model is finished, each function should only ever be modelled once in the entire model; thus easing the task of identifying where corrections are necessary when faults are found and equally simplifying the process for changing management.

3.3.3 Identifying Events

Objectives

The main objectives of this activity are:

- Identification of commands sent to the system by the external actors
- Distinguishing the “external” commands from the “internal” events produced and consumed by the system.
- Identification of status indications sent by the system to the external actors
- Identification of detected values sent to the system by the field elements
- Identification of steering values sent to the field elements by the system

Elaboration

Apply the following techniques to find events:

- Identify events from black box sequence diagrams
  Within a use case, what commands need to be sent to the system?
  What status indications are returned to the actor during a use case?
  What detected values may be received by the system during a use case?
  What steering values are emitted by the system during a use case?
- Use the following checklist for identifying events based on domain objects:
  What events cause an object to react somehow (e.g. by sending another event)?
  What events cause an object to react differently afterwards (i.e. changing its internal state)?
- Describe the events and, if necessary, specify any event parameters.

Events

The identification and sending of events is particularly important at the system boundary, as the raising of events is the only means by which external actors can interact with the system and by which external track elements can be both commanded and monitored. Within the system itself, the use of events is largely discretionary, and their presence is very much the result of the chosen modelling style. This is because whether an event is sent as a result of a change in conditions, or whether a ‘when’ event automatically detects the same status change and reacts accordingly is immaterial.

Parameters

Signal events, as opposed to ‘when’ events can easily be configured to carry configuration data in the form of parameters. These are simply appended to the event and populated at the time the event is sent, usually by data from the instance from which the event originates. Few parameters are used in the GENERIS model (apart from in the class that sets up all the instances and configures the layout of the tracks), as most data values that are sent are constants.

Event Naming Convention

In GENERIS the following naming conventions has been applied as far as possible:

- For commands and steering values: `action object [predicate]`
- For status indications and detected values: `object [not] predicate`
- In general always use “not” for the opposite of something, rather than looking for an antonym.

3.4 Behaviour modeling

3.4.1 Interaction Modelling

Objectives

Interaction diagrams show the interaction between two or more entities that solve a common problem by collaboration. In the context of INESS this means defining black box sequence diagrams to show the interactions between the system and its environment within a specific use case.

Elaboration

For black box sequence diagrams:

- Create a new diagram for the use case to be described
- Add the events that are raised outside the system but going into the system

3.4.1.1 Examples

The following diagram shows the interaction between the signaller and the interlocking system, as well as the interaction between the interlocking system and the field elements during the use case “move point”.

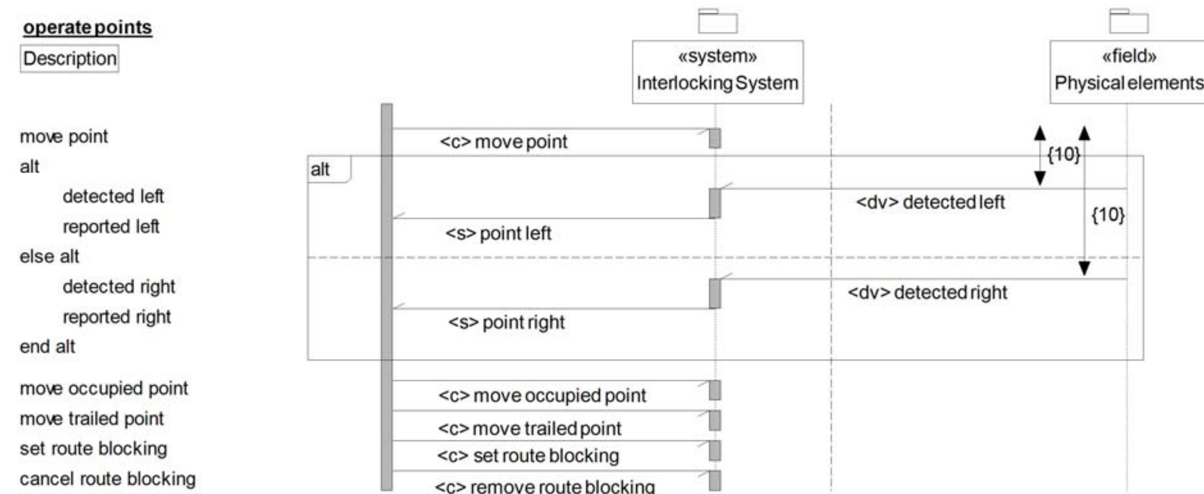

Figure 11

3.4.2 State Modelling

Objectives

The main objectives of state modelling are:

- Elaboration of the life cycle of objects inside the system
- Identification of events that are relevant to an object
- Precisely specifying the permissible sequence of events of objects
- Precisely specifying the (re-)actions of an object to the occurrence of an event.

Elaboration

For each HAL object, then for each LCL object:

- Identify states
- Organize states into disjoint groups
- Differentiate between sequential states (which are mutually-exclusive) and parallel states (which belong in separate state compartments)
- Organize states and events/transitions for the "normal" sequence
- Add guards
- Add exceptional sequences
- Add actions, including outputs to drive the user GUI.
- Consolidate

State Diagram Semantics

Assume the following semantics for any state machine modelled by a state diagram:

- Events that are not mentioned on a state diagram are ignored by that object.
- Events that are specified for the current state of an object cause the specified actions.
- When a guard evaluates to FALSE, the corresponding event is ignored by that object.
- Events that are not specified for the current state of an object but for one of its other states are rejected, i.e. signalled as an error.
- The behaviour specified in a super state is inherited (i.e. implicit) for all its sub states.

State machine “slices” and concurrent states represent concurrent state machines.

3.4.2.1. Examples

As the chosen modelling style of the GENERIS project tends to minimise internal events, the logical elements of the interlocking specification, which do not need to directly reflect the statuses of the physical elements in the field are rather static. Thus in contrast to the diagram shown in chapter **Erreur ! Source du renvoi introuvable.** above, the LCL diagrams have far fewer state transitions, or even have none at all.

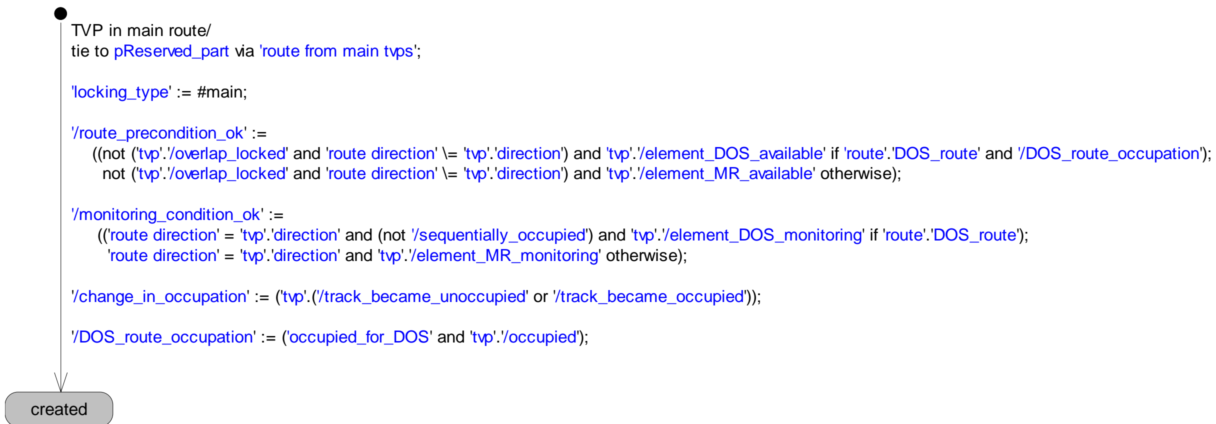

Figure 12: State diagram of the Track Section as used in a main route

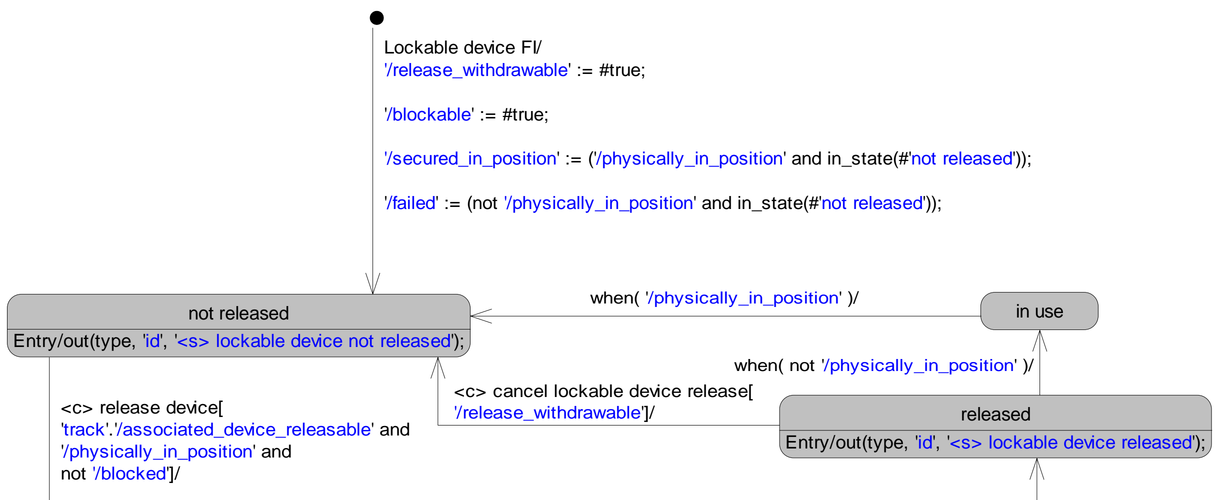

Figure 13: State Model of a Finnish Lockable device.

A HAL level state diagram, however much more closely reflects the physical world and thus Figure 13 clearly shows the stages through which a lockable devices passes as it is used.

Figure 14: State Diagram of "Route"

Nonetheless, as Figure 14 shows, not all the state diagrams of logical elements are small and devoid of complexity.

3.5 Other modelling aspects

Object Constraint Language

The Object Constraint Language (OCL) could be considered as a further diagram type of the UML. However it is not graphical, but rather a textual and very formal language used to express side-effect-free constraints on the object model. OCL can be used in a requirements model to define safety invariants and pre- and post-conditions. However, in practice the OCL syntax proved to be a limiting factor for its acceptance and consequently in the GENERIS model xUML a variant of OCL is used for expression constraints in an English-like syntax, but with the same semantics as OCL itself.

Stereotypes

Stereotypes are a mechanism to extend the UML as well as to adapt it to project or domain specific needs. Basically it is some kind of "meta classification" that potentially could be applied to any element of the UML. Stereotypes can be considered as standardized name-prefixes (enclosed in guillemots « »

or < >) for those elements that imply some stereotype-specific properties of that element. For example in the context of an interlocking system, a class could be used to represent a physical point in the field (stereotype «pe» for "physical element"). An overview of the use of stereotypes in modelling interlocking system requirements are further described in the Appendix B of this document.

3.5.1 Results

During the development of the requirements model, the results were produced and they also need to be maintained. Although this is not something particular to models created in UML, the process is equally valid for it. This effects the following artefacts:

- Textual requirements stored in the DOORS Database
- UML models stored in ARTiSAN Studio
- Data Preparation files for Simulations, i.e. the files belonging to a standalone simulation of a specific scenario based on the UML model
- Data setting for the generation of the Graphical User Interface.
- Additional Word documents such as generated model reports, test documents, this guideline or other explanatory documents.

Section 4 – CONCLUSIONS

In this document, we described the process by which the model has been derived from the DOORS functional requirements. The application of this methodology lead to the Generis model, a conventional interlocking functional representation. It is worth mentioning the powerful concepts used such as inheritance which constitute a complementary way to identify common core functionalities by virtue of generalization/specialization.

The modelling style, paradigm and syntax illustrated in the document should be useful not only for the INESS modelling work for the model upgrades towards the ERTMS functions, but also for other activities such as model transformation and validation.

Section 5 – ANNEXES

Appendix A UML Notation

The following figure shows the most important elements of a UML use case diagram:

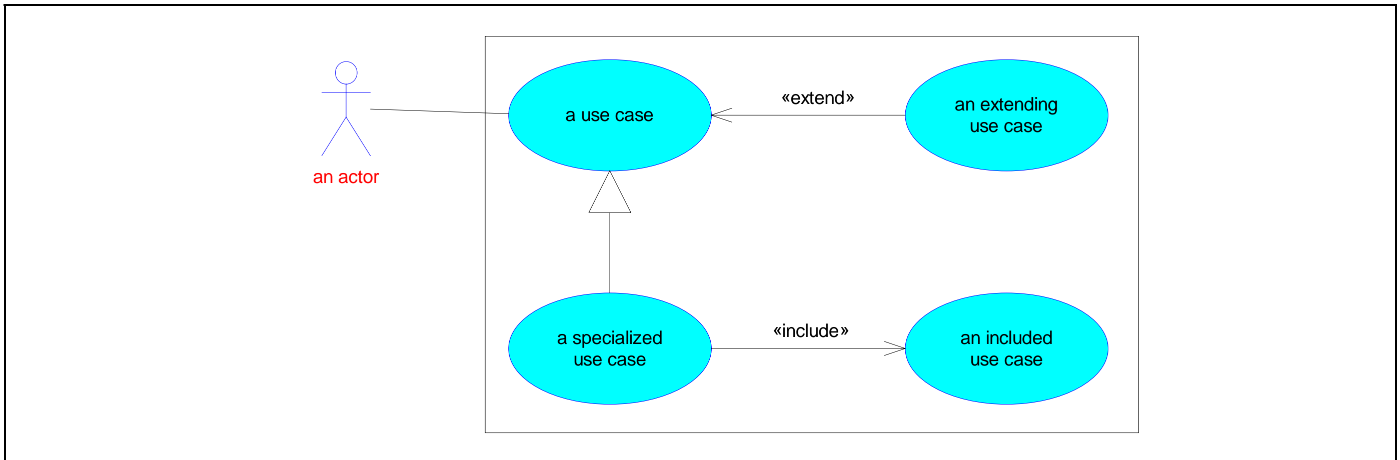

Figure A-2: Use Case Diagram Notation

The following figure shows the most important elements of a UML class diagram:

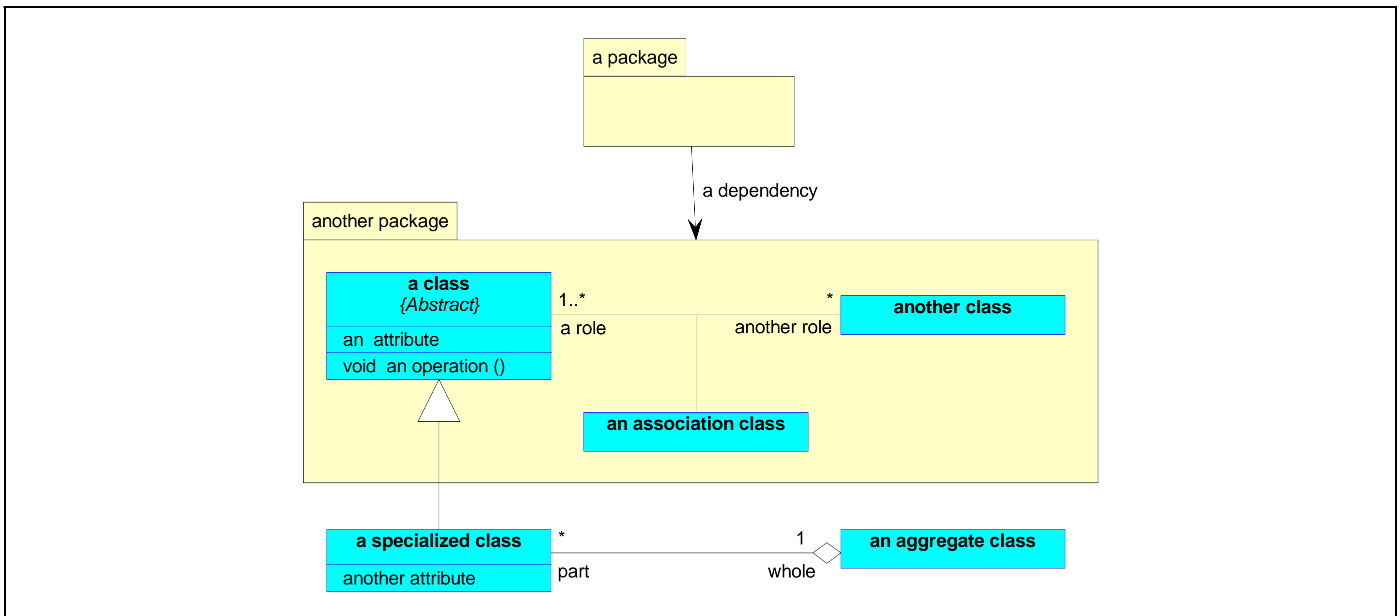

Figure A-3: Class Diagram Notation

The following figure shows two aspects of the same class “Person”: “Customer” and “Supplier”.

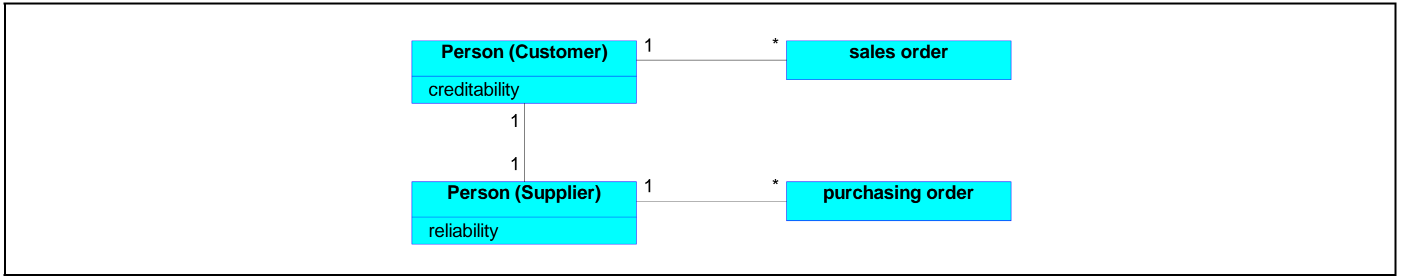

Figure A-4: Aspects Example

The following figure shows the most important elements of a UML state diagram:

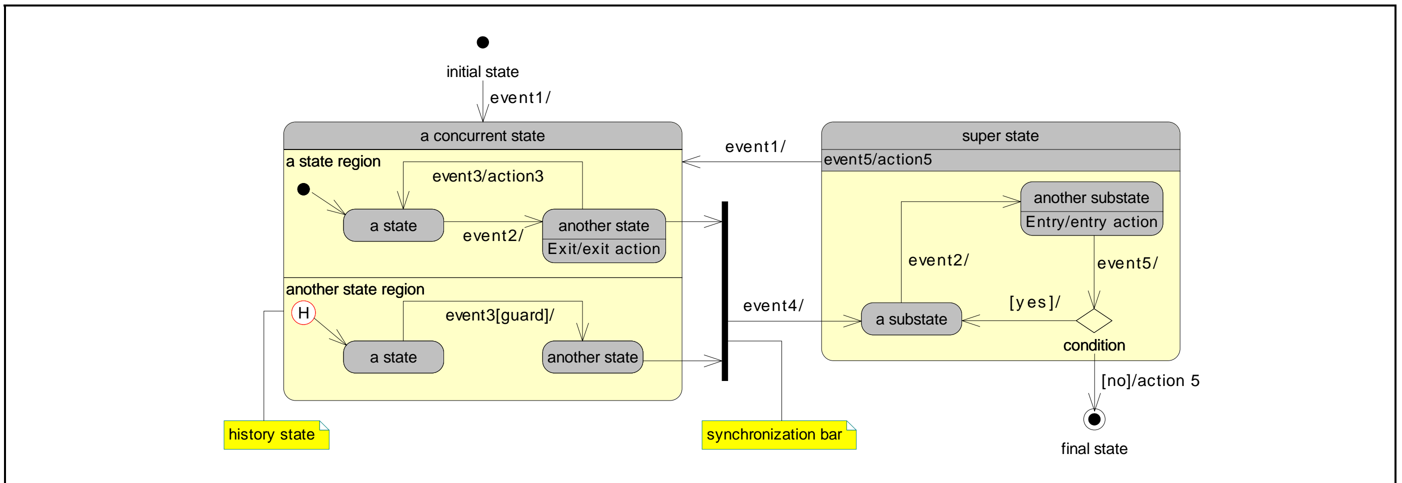

Figure A-5: State Diagram Notation

The following figure shows the most important elements of a UML sequence diagram:

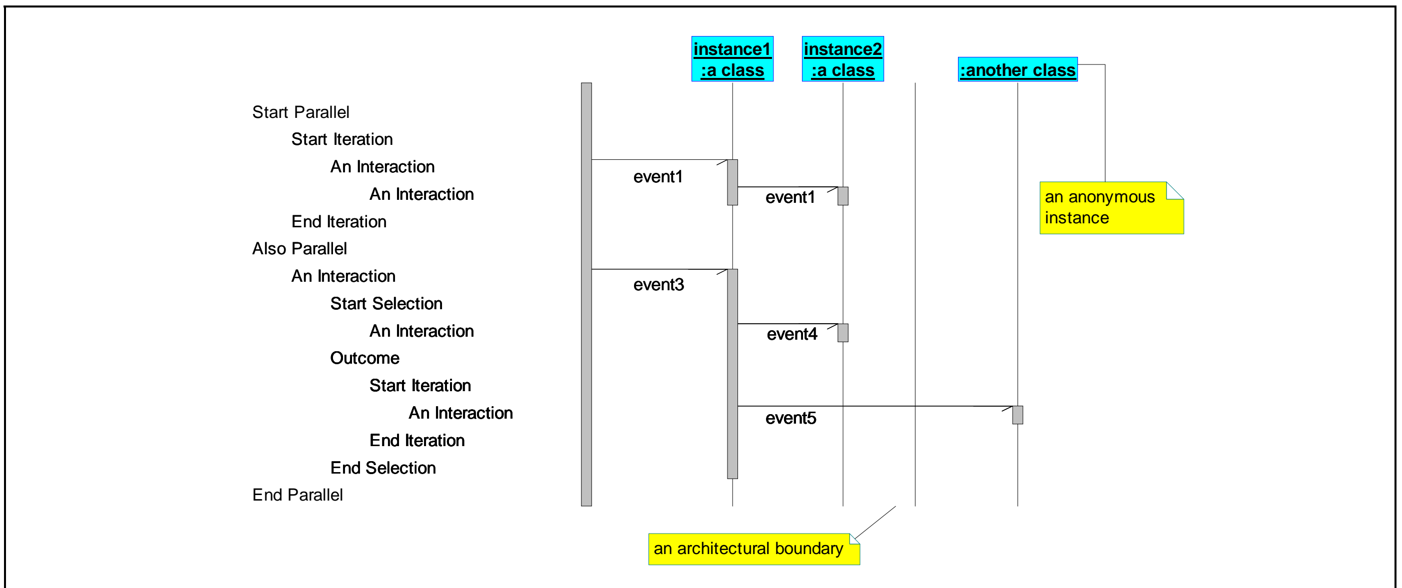

Figure A-6: Sequence Diagram Notation

Appendix B Stereotypes

The following UML stereotypes were used in the GENERIS model:

Classes and Packages of Classes:

- <system> system
- <field> controlled by the system, but external to it.
- <pe> physical element in the rail yard

Events:

- <c> command from an actor to the interlocking system
- <s> status indication from the interlocking system to an actor
- <dv> detected value sent by a physical element to the interlocking system
- <pe> steering value sent by the interlocking system to a physical element
- <ic> interlocking system internal command
- <ee> external event influencing the status of a track element
-

The following picture illustrates the general usage of stereotypes for events:

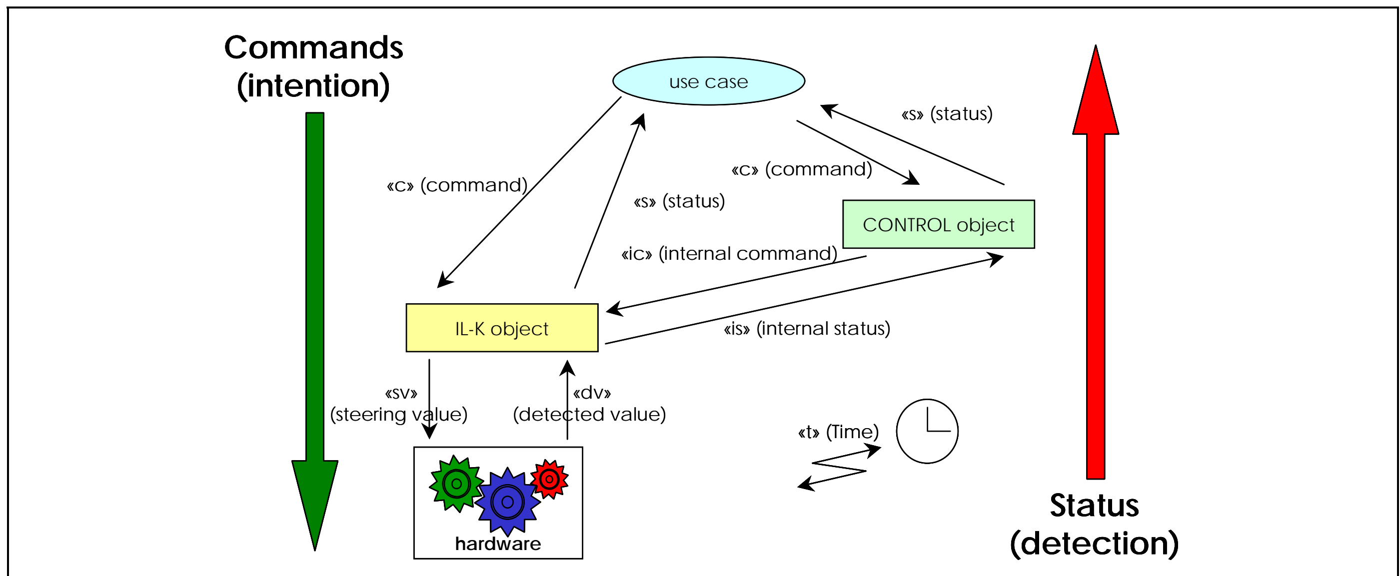

Figure B-7: Event Stereotypes