FP7 Project 2007-Grant agreement n°: 218575

Project Acronym: **INESS**

Project Title: **INtegrated European Signalling System**

Instrument: Large-scale integrating project
Thematic Priority: Transport

Document Title: INESS_WS D_Deliverable D.4.1_Documented strategy for Verification and Validation_Report

Due date of deliverable	M6
Actual submission date	M6

Deliverable ID:	**D.D.4.1**
Deliverable Title:	Documented strategy for Verification and Validation
WP Related:	WP D.4
Responsible partner:	UIC
Task/Deliverable leader Name:	Jeroen Ketema (LaQuSo)
Contributors:	UIC, Railsafe, York, Southampton, LaQuSo (TU/e)

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

Project coordinator: Paolo De Cicco
Project coordinator organisation: UIC

Revision: TAB 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:	Report
Document Name:	INESS_WS D_Deliverable D.4.1_Documented strategy for Verification and Validation_Report_Ver2009-05-27
Revision:	TAB finalised
Revision Date:	2009-05-27
Author:	UIC, Railsafe, York, Southampton, LaQuSo (TU/e)
Dissemination level:	**CO**

Approvals

	Name	Company	Date	Visa
WP leader	Khalid AGROU	UIC		
WS leader	Wendi MENNEN	ProRail		
Project Manager	Emmanuel BUYSENE	UIC		
Steering Board				

Document history

Revision	Date	Modification	Author
Initial	2009-03-13	-	Jeroen KETEMA/LaQuSo
First	2009-03-31	Accomodate W. Mennen's review	Jeroen KETEMA/LaQuSo
PMC finalised	2009-04-22	Accomodate comments by PCM	Jeroen KETEMA/LaQuSo
TAB finalised	2009-05-27	Accomodate comments by TAB	Jeroen KETEMA/LaQuSo

TABLE OF CONTENTS

Chapter 1. EXECUTIVE SUMMARY 4
Chapter 2. INTRODUCTION 5
Chapter 3. VALIDATION AND VERIFICATION STRATEGY 8
Chapter 4. CONCLUSION 21

GLOSSARY

ACP Algebra of Communicating Processes, process algebra

CENELEC European Committee for Electrotechnical Standardization, standards body

DOORS Dynamic Object Oriented Requirements System, requirements management tool

EGL Epsilon Generation Language, language for model-to-text transformation

ERTMS European Railway Traffic Management System, railway signalling standard

ETL Epsilon Transformation Language, language for model-to-model transformation

LaQuSo Laboratory for Quality Software, partnership of Eindhoven University of Technology and the University of Twente (among other Dutch universities)

LPS Linear Process Specifications, process specifications satisfying a special format

LTS Labelled Transition System, transition system in which transitions are labelled

PBES parameterised boolean equation system, equation system consisting of boolean expressions

Railsafe Railsafe Consulting Ltd., railway consultancy firm

Southampton University of Southampton, university

UIC International Union of Railways, worldwide international organisation of the railway sector

UML Unified Modelling Language, general-purpose modelling language in the field of software engineering

xUML executable UML, a flavour of UML that is formal enough to be executable

York University of York, university

Chapter 1 — EXECUTIVE SUMMARY

The objective of work package D.4 (WP D.4) is to validate and verify a core set of functional requirements for interlocking systems, as captured in structured natural language from the participating railways. The current document lays down a strategy to achieve this aim

The core set of functional requirements will be modelled in UML (Task D.3.3, UIC) and it is this model that will be validated and verified. The aim of validation is then to gain confidence that (a) the model is a formalised expression of the functional requirements captured in natural language, and that (b) the captured functional requirements are optimal from the perspective of the functionality desired by the participating railways.

The aim of verification of the model is to show that the model satisfies certain requirements that are not directly expressed as functional requirements. These requirements include safety requirements such as the impossibility of trains colliding.

Validation will be performed by associating test cases — also intended for the later testing and commissioning of INESS conformant interlocking systems — with the textual requirements (Task D.4.2, Railsafe). The behaviour of the model on a given test case will be simulated. The validation of the model will be based on the simulation outcomes of these test cases.

Verification will be performed by giving rigorous, mathematical arguments. These arguments will either apply to specific track layouts or to all possible track layouts.

Both validation and verification are complex, laborious tasks. In order to perform these tasks quickly and accurately, computer programs will aid in the validation and verification. These computer programs are available, but need to be adapted to be able to interact with UML and to be able to deal with the size of the validation and verification, which is a major undertaking (Task D.4.5, universities).

Chapter 2 — INTRODUCTION

2.1 Objective

The objective of work package D.4 (WP D.4), as formulated in the INESS Description of Work, is to validate and verify a common kernel of the functional requirements of interlocking systems, as captured from the participating railways. Below, a strategy is elaborated that can be used to achieve this aim. Product validation and verification is explicitly not part of the strategy and outside the scope of the work package.

The functional requirements have been (or will be) captured and collected in structured natural language in a DOORS database. A core set of these functional requirements is going to be modelled in Executable UML (xUML) (Task D.3.3, UIC). Henceforth, we will refer to this xUML model as the "Interlocking Model". The Interlocking Model is one of the inputs of the validation and verification activities of WP D.4. Since this model will not be available until late in the project, the model originating from the Euro-Interlocking project will serve as a temporary replacement. Henceforth, we will refer to this model as the "GENERIS Model".

The Interlocking Model will be constructed by a modelling engineer from the informal, textual functional requirements as collected in the DOORS database. This construction process will involve a certain degree of interpretation and formalisation by the modelling engineer. The Interlocking Model itself can be thought of as a generic interlocking; it describes the classes of the concrete objects (track, route, etc.) that play a role in an interlocking system. An instantiation of the Interlocking Model, i.e., a collection of objects that are instances of the classes defined by the Interlocking Model, can be thought of as the specification of a concrete interlocking. It is important to note that there are (infinitely) many ways to instantiate the Interlocking Model.

2.2 Definition of validation and verification

We briefly describe what we mean by validation and verification, and we give a short summary of our strategy, as described in Sections 3.1 and 3.2.

Validation of a model refers to the process of assessing that (a) the model is a correct representation of the textual functional requirements, and that (b) the textual functional requirements are actually those desired.

Validation of the Interlocking Model will be carried out by associating test cases with the textual requirements in the DOORS database, modelling the test cases in UML, simulating the behaviour of the Interlocking Model on the test cases, and evaluating the observed behaviour. The test cases will be synthesised by examining all operationally necessary sequences of functions that occur during the passage of trains. The tests will predominately take into account normal operating modes, however common and credible equipment failures will also be tested.

Verification refers to the use of formal, mathematical techniques in proving that a model has certain properties. The main differences with validation (via testing) are: (a) Verification uses rigorous mathematical arguments to make statements about the model; (b) Verification is an exhaustive technique, that is, the entire model is verified, unlike in testing where the behaviour of the model on certain probable scenarios is evaluated; (c) The properties that are verified do not necessarily express functional requirements of separate components, but they can express properties of the global, temporal or concurrent behaviour of the model.

We will apply the following three verification techniques: model refinement, model checking and theorem proving. The use of three different techniques is expected to produce results that will cover more aspects of the Interlocking Model than would be possible using only one type of formal verification. The nature of these techniques, their differences, and their pros and cons are described in Section 3.2.3 on verification techniques.

2.3 Tools

To perform the validation and verification quickly and accurately, we will use tools (computer programs). We do not intend to develop these tools within the project, but to use, and possibly adapt, existing ones. We summarise the kind of tools we shall employ in the project:

- simulators (for validation purposes);
- transformation tools (for transforming the Interlocking Model into the input languages of other tools);
- model checkers (for exhaustive verification of example track layouts); and
- theorem provers (for verifying the internal consistency of the Interlocking Model and for proving general properties about the model).

In order to be able to use existing tools, it is necessary to transform the Interlocking Model into the different input languages of these tools. Given the size of the Interlocking Model, and given the fact that the Interlocking Model will change in the course of the project, we need to automate these transformations. We expect to be able to develop such an automated transformation tool by adapting existing transformation tools that have been developed for similar purposes. Combining the transformation tool with other existing tools would provide a first step towards a fully automated verification process.

Although there is some overlap between the different verification tools, the purpose of using several tools is not so much to compare their answers on the same question, but rather to be able to answer different questions with different tools. In the cases where the same question is answered by different tools, it will be interesting to see which tool is most efficient. We do not expect that one of our tools will outperform the other tools on all types of verification. The tools can only produce conflicting outcomes on the same question if their representations of the Interlocking Model differ. If conflicting results are found, we will use them to correct the model representations. In summary, by using several verification tools, we believe we are able to perform a more efficient and more complete verification.

2.4 Expected results

Validation. The validation will deliver a semi-formal implementation independent model of the necessary test sequences. This will be in a notation that can be readily applied to different technologies and that can be translated to propriety test environments. The notation will also facilitate future automation of testing of INESS conformant interlocking systems. More concretely, the above includes (a) a set of generic test cases, (b) synthesised test sequences (test plans), and (c) possibly test cases as UML sequence diagrams.

Verification. We expect, at the end of the project, to be able to answer the following questions:

- Does the xUML of the Interlocking Model contain flaws (inconsistencies)?
- Does the Interlocking Model contain behaviour that is clearly unwanted (deadlocks, livelocks, etc.)?
- Does the Interlocking Model ensure that all relevant safety requirements are satisfied?

Since the Interlocking Model will not be available until late in the project, we will mainly focus on developing methodologies and tools to be able to answer the above questions quickly and accurately. To this end we employ the GENERIS model from the Euro-Interlocking project.

2.5 Risks

Interlocking Model

The Interlocking Model that needs to be validated and verified will only be delivered late in the project (Deliverable D.D.3.3). Hence, if the Interlocking Model is not delivered in time, WP D.4 may not be able to successfully complete its validation and verification activities on the Interlocking Model.

To mitigate the above risk, we will work with the GENERIS Model stemming from the Euro-Interlocking project. The Interlocking Model to be produced by WP D.3 is supposed to be similar in style (i.e., type of xUML, modelling conventions, etc.) to the GENERIS Model. To mitigate the effect of changes in the style of modelling, we plan to regularly consult with the modelling engineer responsible for the Interlocking Model.

Safety requirements

The verification of the Interlocking Model depends on the availability of safety requirements in a formal format. However, the informal requirements on which these are to be based (Task D.4.4), i.e. those of the CENELEC Phase 3 documents from the Euro-Interlocking project, are too high-level and unspecific to be of any use.

To mitigate the above problem, we will consult with the railways and Railsafe and possibly other partners involved in INESS. If this does not yield any safety requirements, we plan to use safety requirements available in the literature on formal verification of interlocking systems.

Chapter 3 — VALIDATION AND VERIFICATION STRATEGY

3.1 Validation strategy

3.1.1 Introduction

The key objective behind the validation activity is to examine by means of tests if the requirements specified in the DOORS database are correct and complete for the European railway infrastructure managers and the industry. This will be achieved by using the xUML based simulation as a vehicle for functional testing (its starting point assumes the Interlocking Model is a broadly complete and correct model of the specified interlocking function). By examining how the system behaves operationally, it will be possible to determine if there are faults in the "system". This could either be that the requirements themselves are incomplete or incorrectly specified in the database or, that Interlocking Model and/or simulation system itself is incorrect. By testing in this way, the validation exercise has a secondary benefit of determining if the Interlocking Model correctly models the specified and intended interlocking behaviour and whether the model is therefore "correct to requirements".

The validation activity will also deliver to the supply industry and to the railways a functional test specification that permits and supports validation of future, real INESS interlockings. The specification will be written in a formalised and systematic manner that can be used and understood by any organisation to validate in future developments and deployments of INESS. The notation and syntax that will be used to express the test requirements will be chosen to be compatible with a wide range of different technologies for possible future automation of testing.

3.1.2 Starting point

During the functional requirements capturing phase, the railways are encouraged to express their requirements in a standard way, avoiding country-specific expression of the same functionality. When this is necessary, a function can be split into several requirements. In particular, this rigorous approach has been applied to certain modules that will play an important role in the validation process: the Commands and Statuses.

On the other hand, the GENERIS Model, a xUML model of a conventional interlocking, not taking into account of the ERTMS environment, is already available, as one of the Euro-Interlocking project deliverables. The model has been simulated with a simulation tool (Cassandra), and it is planned to reuse this tool for running the test cases and visualisation.

This legacy has led to the decision to perform validation according to the "black box" approach: the system is viewed as a box where we do not take in consideration the internal events and processes. Rather, we will stimulate it, and will interpret any output generated by the system.

3.1.3 Test cases

We will express the test cases using a syntax and vocabulary as close as possible to the one used in the DOORS functional requirements database. The intention is to produce a set of generic test cases in which each test sequence will be applicable to any particular track layout, that will stimulate the model. For instance, the various stimuli must be as close as possible to the already existing commands. The outcome of the test case will be captured from any already available statuses, signals, movements etc. In other terms, the information contained in the Command and Statuses module will be utilised.

The configuration in which the system will be stimulated will be synthesised from the requirements database, where a variation on the conditions to be satisfied will be introduced for each eligible functionality. Sometimes, inexplicit conditions may be needed prior to run the test case; this will be taken in account by means of a "pre-condition" that will sit in a column by itself.

These various considerations led to a structure made of eight columns (seven for the test case plus one for the countries to which the test case is applicable):

1. Identifier of the test case;
2. Identifier of the functional requirement being tested;
3. The creating preconditions: sometimes, it is needed to create a particular context before running a test sequence;
4. The combination of conditions required to perform the test of the considered requirement;
5. The command to be launched to run the test, i.e. the stimulus;
6. The test case result;
7. A comment to give explanation for special cases;
8. The countries to which this scenario is applicable.

Each of these test cases will be linked to the key-functional requirement it is testing.

From that, we will be in a position to provide a set of so-called "generic" test cases, applicable to a big number of railways without modification, but nevertheless able to test specific configuration of specific railways as their conditions will remain within the database. If the test cases are generic enough, any railway will be able to run a test sequence on a layout of his choice.

It should be noted that only "positive testing" will be carried out, i.e. only testing of normal operational scenarios and conditions. We will write test cases only for relevant requirements in the light of normal system operation and possible (logical) failures. This is why the role of the railways will be important: only a domain expert can imagine a track configuration from a test case expressed in generic terms.

A sufficiently deep analysis of the requirements database should, along with test cases drafting, permit identification of the best test case sequence. We can expect that in certain circumstances, the particular situation where the system stands after running a test case will actually give us the proper context to test another requirement, whatever can be the result. This type of consideration will be taken in account and synthesised in the test plan.

3.1.4 UML

Once a sufficient number of these test cases have been stabilised with regard to all relevant functionality, they will be modelled in UML in the form of sequence diagrams, possibly supported by other diagram types such as state transition where appropriate. This method suits the black box approach explained before.

The purpose of this work is to deliver, for the benefit of future INESS product developers and testers, a semi-formal implementation-independent model of the necessary test sequence in a notation that can be readily applied to different technologies with ease and has the capability of translations to propriety test environments and to facilitate future automation of tests.

The notation will also serve as a mechanism of effectively and unambiguously communicating the test requirements to manual testers.

3.1.5 Validation process

The validation process will:

- Derive a set of generic test cases;
- Synthesise the test sequences (test plan);
- Produce the corresponding UML sequence diagrams.

Only the common core functions will be tested. When a test case results in setting the system in the initial conditions of another one, we will group them in a particular sequence in order to improve the efficiency.

In terms of quantity, we do not consider it mandatory to have exactly one test case module for one functional requirements module. Given the fact that some complex functional requirements modules (such as route) can refer to more than one other module, we will package them into one test case module.

3.2 Verification strategy

3.2.1 Introduction

As explained in the main introduction, the purpose of verification is to determine if the validated Interlocking Model (and hence, the functional requirements) ensure certain properties. Verifying that the Interlocking Model indeed satisfies these properties will be done using several different methods, each of which requires dedicated tool support. Rather than developing these tools ourselves, we shall use the following tools: the mCRL2 toolkit, the Rodin proof tools with UML-B front end, and FDR2 together with ProBe, ProB and PAT. To be able to use these tools, the Interlocking Model needs to be translated to their input languages.

We first explain which properties will be verified. Then we explain our verification techniques, our approach to the problem of translating the Interlocking Model into the input languages of the different verification tools, and we briefly discuss the different functionalities of the different tools. Finally, we explain how we intend to address the actual verification and how the results are translated back to the Interlocking Model.

3.2.2 Properties

We will verify certain structural properties of the Interlocking Model and its internal consistency (including deadlock and livelock freedom). This guarantees a kind of consistency in the form of absence of commonly undesirable problems, making sure that the system does not seize up unexpectedly, refusing to make any further useful progress. Some of the tools we use will also guarantee that components are deterministic, where this is desirable. We will also verify certain functional properties, derived from safety requirements. For example, one of these safety requirements may say that, under certain circumstances, no two trains may occupy the same track circuit. This is known as a conditional invariant, describing certain working behaviour. Exceptional circumstances, beyond the control of the signalling system, may occur, and we will analyse the causes for such exceptions, and strategies for re-establishing the invariants.

3.2.3 Verification techniques

We will essentially use three verification techniques: model refinement, model checking, and theorem proving. We give a short description of each of these. The use of three different techniques is expected to produce results that will cover more aspects of the Interlocking Model than would be possible using only one type of formal verification.

Model refinement. Model refinement can be described as checking the consistency between two specifications that are expressed as models. Often the specifications to be verified are expressed within a single model (referred to as internal consistency). For example, in formal models it is usual to express static properties (e.g. safety properties like deadlocks) as "invariants" and verify that a model of the behaviour always satisfies these invariants. The two specifications involved in the verification are the static invariants and the behavioural model. Where one model *refines* another one by providing more detail, the refined model should be consistent in some sense with the original one. Checking this consistency is a form of verification.

The value of verification via model refinement is that, if we described something in several different but consistent ways, it is more likely to be correct than if we only described it once. However, more than this, we can choose different ways to describe particular aspects more clearly. For example, we are more likely to express a safety property correctly if it is isolated in an invariant than when it is hidden in a behavioural model. Similarly, refinement introduces aspects of a specification in small, manageable steps so that we can deal with them one at a time. Hence model refinement enables us to build a specification out of diverse views whilst ensuring that the views are consistent with each other.

Model checking. In model checking, one checks whether a formal model satisfies some formal property. The formal model is a mathematically precise description of all possible behaviours of a system, and the formal property is expressed in a logical language, such as temporal logic or μ-calculus. By an exhaustive inspection of the formal model, it is determined if a certain desired property (no deadlock, a safety requirement, etc.) is satisfied by all system behaviours. It is not feasible to do such an exhaustive inspection by hand; for this we shall employ dedicated tools called model checkers. In simple terms, a *model checker* is a computer program which takes a formal model and a formal property as input, and returns Yes if the property is satisfied in the model (i.e., by all behaviours), and No otherwise. If a property is found not to hold, a model checker can also return diagnostic information. Such information is given in the form of an "error trace" which is a sequence of inputs and system reactions that lead to violation of the property. Error traces can be very useful as feedback for correcting the model.

One well-known problem in model checking is that as new aspects are added, the formal model can become so large that it exceeds the capacity of currently available computing machinery. In order to deal with this so-called "state space explosion problem", model checkers often incorporate quotienting techniques, or symbolic techniques that avoid generating the full state space.

Another important point to be made is that the formal model used in model checking represents a concrete instantiation of a system. In the current context, this means that only particular instances (viz. track layouts) of the generic Interlocking Model can be model checked, and all model checking results are relative to some specific track layout. In the model checking process, it is therefore important to consider instantiations of the Interlocking Model that capture interesting, realistic and potentially problematic features of interlocking systems.

Theorem proving. The idea of theorem proving is to formulate a certain desired property of interlockings as a mathematical theorem about the Interlocking Model. By proving the theorem it is established that the property is satisfied in every possible instantiation of the Interlocking Model, and thus by every interlocking that is built according to the functional requirements. It is convenient to use dedicated tools to assist in the search for a mathematical proof; such tools are called *theorem provers*. Compared with the fully automated approach of model checking, theorem proving requires a substantial amount of human effort. On the other hand, theorem proving allows the verification of generic properties and does not suffer from the state space explosion problem.

Generic verification through theorem proving involves:

1. an axiomatic specification of the generic properties of instances;

2. a generic model of the behaviour of such systems;
3. theorems expressing the desired safety properties (invariants).

An attempt is then made to prove that (3) follows from (1) and (2).

The main problem with generic proof is that the theorem proving can be difficult. A good theorem proving tool is needed that will deduce what needs to be proved (i.e. the proof obligations that are steps along the way to fully proving the theorem) and automatically proving them where possible. Even with a good tool, it is often necessary to choose ways to express the model that will assist the proof. If a proof does not succeed it is not known whether this is because it is false or because the prover was not strong enough. However, despite these difficulties, proof is very valuable in modelling. When a proof does not succeed, the proof obligation can be examined. With experience this provides insight into the model and it can then be improved or corrected in order to facilitate the proof.

3.2.4 Transformation of the Interlocking Model

Before actual verification can be carried out, the Interlocking Model needs to be transformed into a mathematically based language suitable for formal verification, a formal specification language. It is understood that the result of the transformation is validated against the original model.

A weakness of this approach is that transformation could change the meaning of the model in unintended ways. This could result in false negative results or in false positive results during the verification. False negative results will be detected when the problem is analysed (either by raising a query with the owners of the Interlocking Model or by testing the model via its executable animation). False positive results are more of a concern because they could lead to a problem going undetected. For this reason and because transformation by hand is not feasible and prone to errors, the transformation will be rigorously specified and implemented via a model transformation technology. Note that the use of several disparate verification techniques also helps to mitigate the danger of false positives.

We intend to develop a transformation tool, which will account for a substantial part of the work. We envision a *domain-specific language* and *model transformation* approach. This will entail:

- Incremental and iterative construction of the modelling language (xUML) used to capture the requirements.
- Explicit rule-based encoding of transformations from the model to a suite of formal specification languages.

The advantages of this approach are several:

- We are able to control the scope and complexity of the language used for requirements modelling, and the scope of the languages used for formal verification;
- We are able to more easily identify those parts of the existing xUML requirements that are difficult (or impossible) to verify, and thus de-risk the overall verification problem by breaking it up into smaller parts;
- We make the transformation from the Interlocking Model to the formal specifications used for verification, as explicit and as transparent as possible, thus enabling validation of the transformation itself (something that is very important for increasing our confidence in the verification results);

- Through the approach to defining model transformations that we envision applying, we enable reuse of the transformations – at least in part – so that transformation rules used to generate formal specifications in one language can also be used to generate formal specifications in a second, different language. Reuse will be achieved by defining the transformation rules on the abstract syntax of the requirements modelling language.

To implement the model transformations, we envisage using the Epsilon Model Management Framework[1], developed at York. Epsilon provides a number of task-specific model management languages, along with Eclipse-based development tools. Two important languages are the Epsilon Transformation Language (ETL) and the Epsilon Generation Language (EGL). ETL is used for model-to-model transformation, wherein a model (e.g., in xUML) needs to be transformed to a different model in a new language. EGL is used for model-to-text transformation (or code generation), wherein a model (e.g., in xUML) needs to be transformed to a textual representation. Both languages are pertinent, as we may need to use model-to-model transformation to produce an internal model that is then easier to map to a formal specification; we will need to use model-to-text transformation, since many of the formal verification tools we intend to use take textual files as input.

We have already developed an initial definition of a small subset of xUML, which supports many of the basic features of xUML (except the action language). Further increments will add the action language. The definition has been implemented in Eclipse/EMF. The implementation allows us to construct and manipulate xUML-like models. This language will be incrementally extended as we proceed until the full extent of xUML that is used in the existing xUML model is supported. This process will allow us to incrementally restructure the existing xUML model to make it more amenable to verification in our selected tools.

The language definition is expressed as a *metamodel*. A metamodel is (effectively) analogous to a context-free grammar for a programming language, in that it describes the abstract syntax of a modelling language (whereas a context-free grammar describes the abstract syntax of a text-based language). The metamodel can be used as the basis for definitions of model transformation rules. The initial metamodel that we have defined, for a subset of xUML, is shown in Fig. 3.1. Naturally, this metamodel corresponds to the UML metamodel of state machines.

The metamodel defines the basic concepts of state machines (e.g., States, Transitions, Constraints, Behaviours) and their inter-relationships. This metamodel is derived from the UML 2.x metamodel, and as such includes several concepts that may not be useful for verification purposes in the future. It is straightforward to remove these concepts if they prove to be unnecessary.

Figure 3.1 is a visual representation of the metamodel; when implementing it Eclipse/EMF, we actually made use of a textual language Emfatic, developed by IBM for specifying the metamodel. Emfatic can thereafter be used to generate visual representations.

Hereafter we can build a transformation from xUML to a formal specification language. We intend to do this in two stages: first, transform xUML models into a model of a formal specification (e.g., a CSP, Circus, mCRL2, or UML-B model), then generate a textual representation of the formal model. This two-stage process allows us to potentially generate several different textual representations from the same formal model, and potentially simplifies the different stages. The first stage involves model-to-model transformation, implemented using ETL. The second stage involves model-to-text transformation, implemented using EGL.

The formal specification languages we intend to target are determined by the tools we will employ for verification. These tools are described in Section 3.2.5.

mCRL2 (LaQuSo)

LaQuSo intends to use mCRL2 as formal specification language. To be able to study the transformation to mCRL2 and to also enable the study of different aspects of the verification, a prototype transformation tool

[1]http://www.eclipse.org/gmt/epsilon

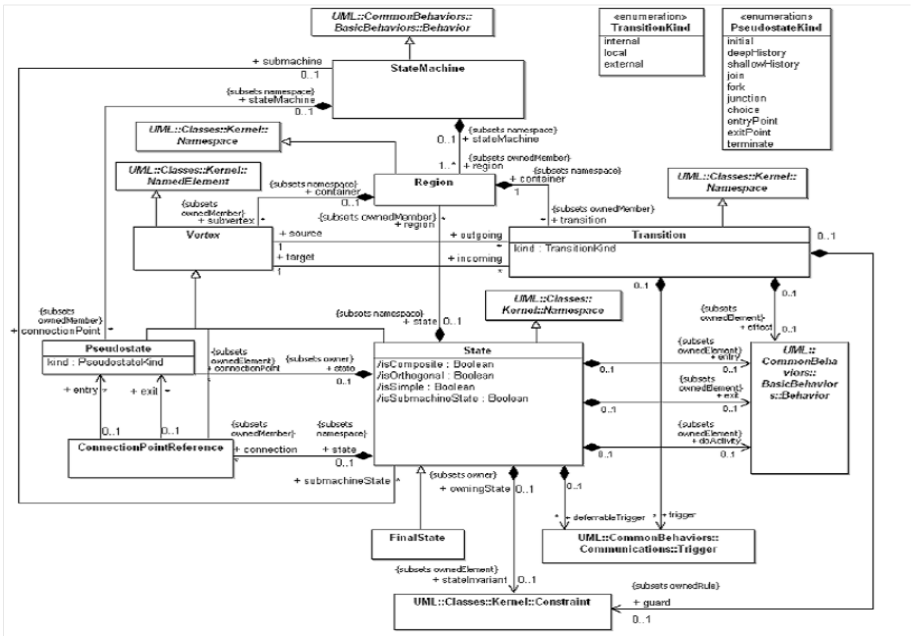

UML::CommonBehaviors::BasicBehaviors::Behavior
StateMachine
«enumeration» TransitionKind
internal
local
external
«enumeration» PseudostateKind
initial
deepHistory
shallowHistory
join
fork
junction
choice
entryPoint
exitPoint
terminate
UML::Classes::Kernel::Namespace
UML::Classes::Kernel::NamedElement
Region
Vertex
Transition
kind : TransitionKind
Pseudostate
kind : PseudostateKind
State
/isComposite : Boolean
/isOrthogonal : Boolean
/isSimple : Boolean
/isSubmachineState : Boolean
ConnectionPointReference
FinalState
UML::CommonBehaviors::Communications::Trigger
UML::Classes::Kernel::Constraint
UML::CommonBehaviors::BasicBehaviors::Behavior

Figure 3.1: A metamodel of abstract syntax of xUML-like state machines

will be built — part of which is already available. This prototype tool will be replaced with the technology developed by the University of York once available.

A transformation of the Interlocking Model into mCRL2 implicitly requires the choice of a formal semantics of xUML as used in the definition of the Interlocking Model. To lay the transformation on firm ground, a formal semantics for the subset of xUML state diagrams used in this project shall be provided. This will enable us to make formal statements about the soundness of the transformation and will enhance the confidence in the obtained verification results. Moreover, the insight gained by the semantics and the soundness proof will help establish a formal and intuitive link between the verification results (e.g., obtained counter-examples in mCRL2) and the original model (e.g., concrete runs of state diagrams).

UML-B (Southampton)

Southampton intends to use UML-B as formal specification language, in such a way that the model builds from an abstract model, gradually incorporating more detail in layers of refinement. Ways will be explored to transform the Interlocking Model into UML-B by manually modelling an equivalent model in UML-B and employing the technology developed by York once available.

The transformation will be done in stages first by using a small pilot study consisting of part of the Interlocking Model. The Rodin verification tools will be used to verify this pilot in an iterative manner as more detail is added. The verification tools are expected to raise questions of the Interlocking Model that will guide the translation. Out of this process the following products will emerge.

- Definition of the translation rules for mapping xUML into UML-B
- Guidelines for verifying the interlocking model using the Rodin tools

UML-B is different from xUML in some significant ways. The Interlocking Model is contained in a single level (no refinement). Also, xUML has a richer notation for transition firing than UML-B does (although, it is

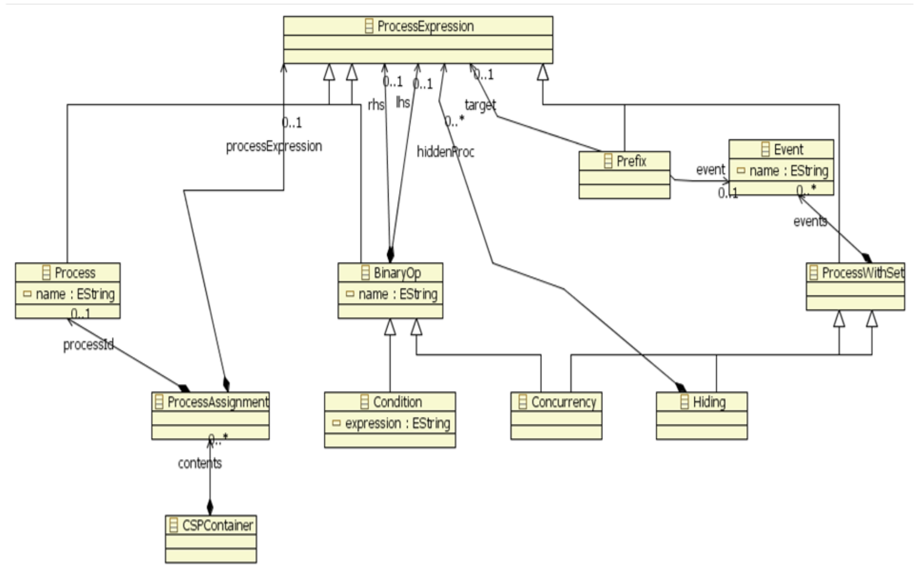


Figure 3.2: A metamodel of CSP

closer to UML-B than standard UML in this respect). However, UML-B does contain diagrammatic notations that essentially map onto the main diagrammatic notations used in the Interlocking Model (Class diagrams and State machines). The GENERIS Model also utilises inheritance and derived variables in a way that may have some correspondence with refinement. An initial pilot study to translate a small part of the Interlocking Model into UML-B was reasonably successful. The translated model was fully proven (although there were only very simple constraints to be proved in such a small model, since safety constraints were not yet available). Nevertheless the exercise demonstrated that a better understanding of the model can be gained by re-expressing it in UML-B and the proof obligations even from this simple model raised certain questions about assumption that have been made in the Interlocking Model model.

CSP, Circus, PROMELA (York)

York will be the main responsible for construction of the modelling language and the encoding of the transformations. Moreover, York intends to use CSP, Circus, CSP || B, and PROMELA as formal specification languages. There is already a prototype of a simple transformation from a subset of xUML to CSP, following the above structure. The transformation requires a metamodel of CSP (an example is shown in Fig. 3.2). Transformation rules from xUML to CSP are defined and also text generation rules.

3.2.5 Verification tools

FDR2

The transformation into CSP will allow the use of a variety of tools to verify various properties of the xUML models. The best-known tool is FDR2, a commercial product developed by Formal Systems (Europe) Ltd. This is usually described as a model checker, but is really an explicit-state checker, in that it converts two CSP process expressions into labelled transition systems (LTSs), and then determines whether one of the processes is a refinement (in a technical sense) of the other. FDR2 applies various state-space compression algorithms to the process LTSs in order to reduce the size of the state-space that must be explored during a refinement check. A successful check guarantees that all the behaviours of one model (the implementation) are permitted by the other (the specification). If we take an xUML model as our implementation, then we can pose different functional properties as specifications and check to see if they are refined by the xUML model. If they are, then the xUML model has those properties.

There are four other useful tools for analysing CSP models. The first is ProBe, an animator for CSP. The second tool is the similarly named ProB tool, developed by the Universities of Southampton and Dusseldorf. It includes animation facilities and support for analysis of CSP processes both through refinement checking, and LTL model-checking. The third is the Adelaide Refinement Checker, developed by the University of Adelaide, which uses different verification technology (ordered binary decision diagrams) to reduce the state explosion problem without requiring the use of FDR2's state-space compression algorithms. The fourth is the Process Analysis Toolkit (PAT), from the National University of Singapore. PAT is able to perform refinement checking, LTL model-checking, and simulation of CSP processes. The PAT process language extends CSP with support for mutable shared variables and asynchronous message passing.

mCRL2

The name mCRL2[2] refers to a formal specification language together with an associated tool set. The formal specification language mCRL2 is based on the process algebra ACP (Algebra of Communicating Processes). An mCRL2 specification thus defines a formal model of system behaviour, also called a process. Apart from process algebraic constructs, mCRL2 allows user defined data types such as lists and arrays to be part of a process specification, as well as the specification of real-time processes.

The property language of mCRL2 is a logical language which can be described as regular modal μ-calculus extended with data. Some of the features of this language include first-order quantification over data, modalities indexed by regular expressions over actions, timing constraints (for stating that some timed action takes place at a certain time), and least and greatest fixed point operators. This language subsumes propositional dynamic logic (PDL) and computation tree logic (CTL), and is well suited for expressing a wide range of properties of concurrent systems such as invariants, fairness and liveness. The regular modalities are suitable for specifying particular sequences of (un)wanted actions.

The tool set associated with mCRL2 can be divided into the following types of tools:

transformation tools that carry out transformations from (a) mCRL2 specifications to linear process specifications (LPSs), from (b) LPSs to labelled transition systems (LTSs), and from (c) LPSs and property formulas to parameterised boolean equation systems (PBESs);

simulators that take LPSs and LTSs as input;

visualisers of the state space (both 2D and 3D);

reduction tools for simplifying LPSs, and reducing the state space of LTSs;

model comparison tools for comparing two LTSs with respect to various types of equivalences (among which trace equivalence and several notions of bisimulation); and

symbolic model checking tools for checking formal properties on PBES representations (which allows model checking infinite state spaces).

One particular strength of mCRL2 is its use of symbolic techniques (on LPSs and PBESs). These symbolic techniques allow reasoning about very large, even infinite, state spaces. Moreover, mCRL2 allows for defining algebraic data types and compared with other process algebras, it accommodates more flexible types of communication.

[2]http://www.mcrl2.org

UML-B modelling tool

UML-B is a modelling notation that contains similar diagrammatic notations to UML for class and state machine modelling. UML-B, however, is based on the formal modelling language, Event-B, and is tailored to have a close correspondence with Event-B. This results in a restricted and more specialised notation than UML. For example, instead of methods, classes have guarded events that fire spontaneously whenever their guards (predicates) are true. There is no concept of calling in UML-B. However, events can influence each other by manipulating the variables in each others guards.

The UML-B modelling tool is a plug-in to the Rodin platform, which contains verification and validation tools for Event-B models. Hence, UML-B models are incrementally translated into Event-B where they are checked for syntax and type correctness and any errors are fed back to the UML-B diagrams. The resulting Event-B is also analysed for proof obligations and where possible, proved automatically. When the prover is unsuccessful manual interaction with the prover is necessary either to assist the proof or to analyse why the proof can not be discharged. Other verification and validation tools in the Rodin platform that will be used include a model checker, disprover (that looks for counterexamples to a proof obligation) and animators.

The Rodin platform and tools, including UML-B, were developed under the EU FP6 project, Rodin[3]. Development continues under the EU FP7 project, Deploy[4]. Southamptons work on INESS, using UML-B in such a full-scale comprehensive and specific modelling task is likely to lead to the discovery of improvements that can be made in the UML-B notation and the Rodin verification tools. In this way, the two FP7 projects will benefit each other.

A strength of the UML-B approach using Rodin proof tools is that the model is verified generically for all layout instances. (As long as they satisfy the general layout properties expressed in the model). Other approaches such as model checking can find many problems relatively quickly, but they always rely on exploring a particular example instantiation and are therefore limited in that there could be other untested examples that fail.

A second strength of the UML-B approach is that Rodin also contains model checking and animation tools. This allows the modeller to utilise a combination of approaches selecting the most appropriate for particular purposes. Drawing on the model-checkers ability to quickly explore the model to gain confidence in parallel with investigating proof obligations. A particular use of the model-checker has been incorporated as a, disprover feature in the Rodin toolset. The disprover can be used to try to find a counter-example to a proof obligation that has not been automatically proved by the Rodin proof tools, Counter-examples illustrate why the model is inconsistent, for example, providing a specific case where a safety property is not obeyed.

A third strength of UML-B is that (because of its Event-B basis) it is a refinement based modelling notation. This means that models are gradually built up in layers, adding more detail (data and behaviour) at each layer. Each layer must be proven to be consistent with the previous layer. The strength of this approach is that it allows the model to concentrate on one concern at a time building detail in small understandable steps.

3.2.6 Verification process

LaQuSo

LaQuSo plans to use the model checker mCRL2 to verify properties of the Interlocking Model.

As already mentioned, it is of great importance to validate the translation of the Interlocking Model into a formal model, in this case, an mCRL2 process specification. To gain confidence in the translation we will use various simulation and visualisation tools supplied by mCRL2. We plan to simulate test cases on the

[3]http://rodin.cs.ncl.ac.uk
[4]http://www.deploy-project.eu

formal model and compare its behaviour with the functional requirements of the Interlocking Model. Using the visualisation tools, we will investigate structural properties of the state space such as expected symmetries.

Due to the complexity of the GENERIS Model from the Euro-Interlocking project (and the expected Interlocking Model) LaQuSo will start by considering smaller models of interlockings mostly created within the Euro-Interlocking project. Once sufficient confidence is gained in the correctness of the xUML to mCRL2 translation, larger parts of the GENERIS Model will be considered. The ultimate goal is to verify realistic example track layouts that instantiate the Interlocking Model. Railways and their sector organisations will be consulted for assistance and input in choosing suitable track layouts that capture as much as possible of the complex behaviour found in the GENERIS Model.

The exact formulation of the formal properties to be verified will depend on the variables used in the model description (such as the names of action labels and state variables). We must therefore first fix a vocabulary for the formal model before formal properties can be defined. The transformation from xUML to mCRL2 will be documented such that model checking results can be related back to the Interlocking Model. In particular, if error traces are found, we will translate these back into UML such that the error trace can be simulated in the xUML model. The model checking results will also provide feedback for the transformation tools.

Southampton

Southampton will use the Rodin verification tools to:

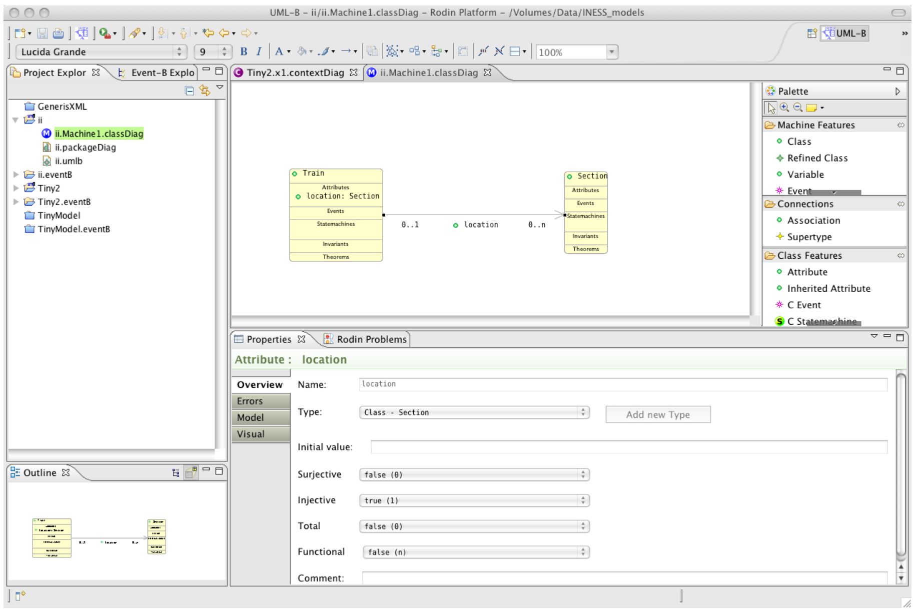
1. demonstrate that the UML-B model is a valid interpretation of the Interlocking Model in xUML;
2. verify the internal consistency of the model;
3. verify that the model satisfies the safety requirements.

The Rodin animation tools may also be used to explore the UML-B model in comparison with the Interlocking Model to test whether the two models behave in similar ways.

The safety properties should be described in terms of states of the objects in our models. For example, if we model the location of trains on track sections a safety property might be that under certain circumstances a track section should have no more than one train located on it at any point in time. Once meaningful safety properties have been described in natural language we will interpret them in the terms of our models. For UML-B this will be as Event-B predicates using the variables and values from the UML-B translated model. For example, we might have an association called location from a class Train to a class Section. The safety property about only one train being in a section would then be that location is injective. Fig 3.3 shows this property expressed in UML-B (The association, location is selected so that the properties view shows its property values. The property injective is set to true and this is annotated on the diagram as the 1 in 0..1 at the Trains end of the association.

This property will generate an invariant with corresponding proof obligation for each event in the UML-B model. (i.e. every event including initialisation must leave the system in a state where this property is true). The model checker may also be used as a quick but less comprehensive check to find a sequence of events that leads to a violation of this safety property.

Since the UML-B model will incorporate layers of refinement, gradually building in more detail, the safety properties will also be specified in corresponding levels of detail. Continuing with the example of Fig 3.3, we might at the next refinement, introduce the concept of signals which probably play a role in ensuring the safety property that location is injective. Hence we might, now express lower level safety properties about the behaviour of signals which we could not do at the previous level. The refinement must be consistent with the previous level, so the prover will require that the refinement maintains the previous safety property



Attribute : location

Name: location

Type: Class - Section

Initial value:

Surjective: false (0)

Injective: true (1)

Total: false (0)

Functional: false (n)

Comment:

Figure 3.3: A property expressed in UML-B

The verified UML-B model will describe the generic properties of track layouts. However, it is also interesting to verify that particular example layouts obey the generic properties that have been expressed in the UML-B model. If they do not, this could either mean that the example is not a valid one (and it should be rejected from use or corrected) or it could mean that the generic properties are not correctly expressed.

In order to test the particular example layouts the UML-B models will be instantiated with the layouts and the Rodin model checker (which has a feature specifically for this purpose) will be used to test the layouts.

- The consistency of the generic UML-B model (throughout its refinement levels) will be proven using the Rodin proof tools. This could discover flaws in the xUML interlocking model and/or flaws in its translation to UML-B.

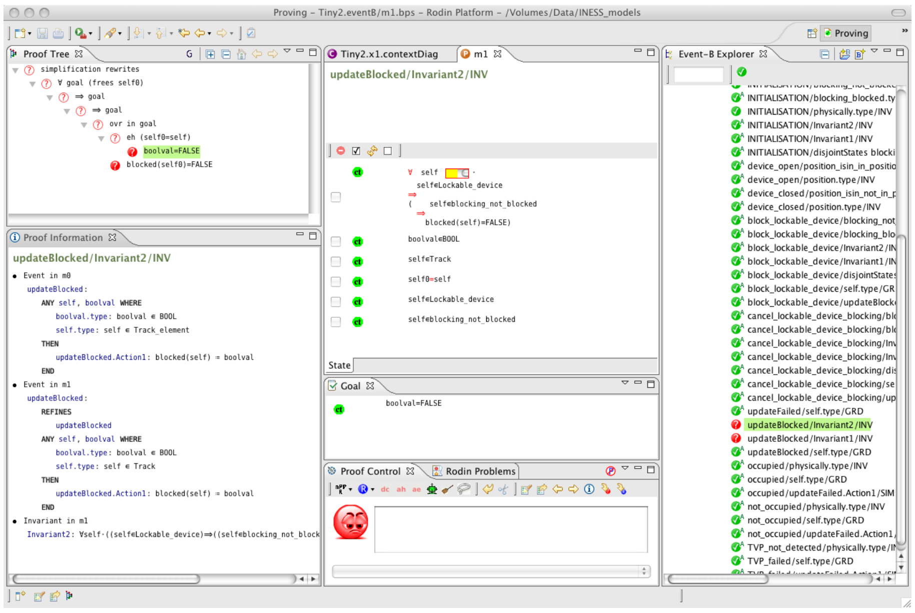
- Safety properties will be translated into invariants embedded in the UML-B model. The Rodin verification tools (prover and/or disprover) will be used to check whether the model obeys the safety properties.

- Note that the main aim here is to prove that the model obeys the safety properties. However, this can only be achieved if the model is consistent throughout. The principal of the prover is that nothing is proved until all proof obligations have been discharged.

A screen shot showing the Rodin prover in use is given in Fig 3.4. The top left panel shows the steps already taken by the automatic prover, the bottom left panel shows information relating to the proof obligation. The middle panels show (from top to bottom) relevant hypotheses found by the prover, the current goal to be proved, a user interaction panel. The right hand panel shows a list of proof obligations which have been proved (if green) or not proved (if red).

York

York will take the following steps:



```
simplification rewrites
  ∀ goal (frees self0)
    ⇒ goal
      ⇒ goal
        ovr in goal
          eh (self0=self)
            boolval=FALSE
          blocked(self0)=FALSE
```

updateBlocked/Invariant2/INV

```
• Event in m0
  updateBlocked:
    ANY self, boolval WHERE
      boolval.type: boolval ∈ BOOL
      self.type: self ∈ Track_element
    THEN
      updateBlocked.Action1: blocked(self) ≔ boolval
    END
• Event in m1
  updateBlocked:
    REFINES
      updateBlocked
    ANY self, boolval WHERE
      boolval.type: boolval ∈ BOOL
      self.type: self ∈ Track
    THEN
      updateBlocked.Action1: blocked(self) ≔ boolval
    END
• Invariant in m1
  Invariant2: ∀self·((self∈Lockable_device)⇒((self∈blocking_not_block
```

updateBlocked/Invariant2/INV

```
∀ self  ·
  self∈Lockable_device
⇒
(    self∈blocking_not_blocked
  ⇒
   blocked(self)=FALSE)
boolval∈BOOL
self∈Track
self0=self
self∈Lockable_device
self∈blocking_not_blocked
```

```
boolval=FALSE
```

```
INITIALISATION/blocking_blocked.ty
INITIALISATION/physically.type/INV
INITIALISATION/Invariant2/INV
INITIALISATION/Invariant1/INV
INITIALISATION/disjointStates blocki
device_open/position_isin_in_positio
device_open/position.type/INV
device_closed/position_isin_not_in_p
device_closed/position.type/INV
block_lockable_device/blocking_not
block_lockable_device/blocking_blo
block_lockable_device/Invariant2/IN
block_lockable_device/Invariant1/IN
block_lockable_device/disjointStates
block_lockable_device/self.type/GR
block_lockable_device/updateBlock
cancel_lockable_device_blocking/blo
cancel_lockable_device_blocking/blo
cancel_lockable_device_blocking/Inv
cancel_lockable_device_blocking/Inv
cancel_lockable_device_blocking/dis
cancel_lockable_device_blocking/se
cancel_lockable_device_blocking/up
updateFailed/self.type/GRD
updateBlocked/Invariant2/INV
updateBlocked/Invariant1/INV
updateBlocked/self.type/GRD
occupied/physically.type/INV
occupied/self.type/GRD
occupied/updateFailed.Action1/SIM
not_occupied/physically.type/INV
not_occupied/self.type/GRD
not_occupied/updateFailed.Action1
TVP_not_detected/physically.type/IN
TVP_failed/self.type/GRD
```

Figure 3.4: A screen shot of the Rodin prover

1. Identify an initial subset of xUML (likely consisting of states, transitions, events, guards, actions/activities), and encode this subset as a domain-specific language in Eclipse/EMF. This will enable the manipulation of requirements models using standards-compliant model management and model transformation tools. Moreover, encoding of models in Eclipse/EMF will allow for lightweight model validation and consistency checking on the model directly, without any transformation to alternative formats. Necessarily, this validation will be incomplete, but it may be possible to cover some functional properties without transformation;

2. Generate sample models that are approximately consistent with those currently available (e.g., the GENERIS Model);

3. Select an initial formal specification language to use for verification of functional properties. Candidates are CSP, Circus, CSP || B and PROMELA, all of which will support specification and verification of a wide range of functional and safety properties;

4. Generate formal specifications in appropriate input formats to use with tools for CSP (e.g., the FDR2 model checker), Circus, CSP || B and PROMELA (e.g., the SPIN model checker). Verify properties against the model;

5. When properties are verified against the translated models (using model checking tools), the final step is to make a relationship between the formal verification results and the original xUML models. For instance, if a property has a false result during verification, this step consists on generating a representation of the false result that can be understood in terms of the original xUML model.

6. Iterate the above steps until we have achieved comprehensive coverage of most, if not all, of the xUML concepts and semantics that are used in the existing requirements model.

Chapter 4 — CONCLUSION

We have defined a validation and verification strategy for the Interlocking Model developed as part of the INESS project. In the next subsection we summarise the verification strategy; the validation strategy essentially consist of developing test cases, as explained in 3.1.3 and the Description of Work.

4.1 Verification strategy

To perform verification quickly and accurately different verification tools will be used. Each of these tools has its own specific strengths and thus the tools will be complementary:

- model checking (LaQuSo and York) will work automatically on fixed track layouts;
- theorem proving (Southampton) will work interactively on all possible track layouts.

By using several different verification tools we can cover a larger range of questions, and by comparing outcomes on the same questions we increase our confidence in the correctness of the Interlocking Model and its formal representation.

In order to verify the xUML model of the functional requirements, we need to transform xUML to the input languages of the different verification tools. This is the expertise of York and we will employ their model-based transformation technology to make a reusable and uniform front-end to our verification tools based on xUML. In essence, the model-based transformations will provide some unifying structure that allows us to systematically use several overlapping yet complementary verification tools.

We next summarise the verification strategy per university.

LaQuSo

LaQuSo plans to take the following steps:

1. Our first goal is to obtain a prototype mCRL2 model of a (generic) interlocking as quickly as possible. For this, we use as input parts of the GENERIS model. We shall on the one hand try to perform translations by hand, and on the other hand work on a prototype translator that is able to handle the xUML constructions used in the GENERIS model.
2. As soon as we have an mCRL2 model of an interlocking we shall start verifying safety properties on small examples of track layouts, mainly to gain experience. We gradually increase the functionality and size of the examples of track layouts that we consider.
3. In the mean time we shall work together with York to get a generic translation from xUML to mCRL2.
4. When our translator is capable of translating the entire GENERIS model, we shall attempt to verify track layouts obtained by consulting with railways and their sector organisations.
5. Finally, we shall try to make a generic translation and try to check generic safety requirements (using parametrised model checking).

Southampton

Southampton envisions the following steps to be performed on the complete model. Southampton will investigate and demonstrate the feasibility of these processes on some parts of the model. This will allow other partners or subsequent work to complete the steps on the full model. The following is an outline plan.

1. Tiny Pilot Study: A small subsection of the GENERIS Model will be hand translated into UML-B and the Rodin verification tools will be used to verify the model. The purpose of this pilot is to,
   (a) learn about the xUML modelling notation and the style of modelling used,
   (b) develop some initial intuition about translation into UML-B and whether the UML-B notation is suitable or needs extending,
   (c) demonstrate the feasibility and usefulness of verification using the Rodin tools.
2. Translation of GENERIS Model into UML-B

   A more substantial part of the GENERIS model will be translated into UML-B. The prototype translation will embody some example safety requirements. The purpose of this prototype translation is to formulate the rules of translation. The Rodin verification tools will be used to an extent necessary to discover whether the translation is valid and feasible.Any queries that are not obvious translation mistakes will be raised within the work-package group for consideration.A definition of the translation will be produced.
3. Animation of GENERIS UML-B model

   The translated model will be animated and tested in parallel with the interlocking model. The purpose of this stage is to gain confidence that the UML-B model is a valid translation of the Interlocking Model.
4. Verification of safety properties in GENERIS UML-B model

   The formalised safety properties will be embodied into the UML-B model. The Rodin verification tools will be used to verify that the UML-B model satisfies the safety properties
5. Verification of specific interlocking layouts

   The UML-B model will be adapted into a form suitable for verifying interlocking layouts. At least one example layout will be specified and the Rodin verification tools will be used to verify that it meets the generic properties of interlocking layouts.
6. Translation of Interlocking Model into UML-B

   The Interlocking Model will be translated into UML-B using the rules developed in step 2. The extent and completeness of the translation will depend on the feasibility of translation discovered in step 2. The Rodin verification tools will be used to check the consistency of the resulting UML-B model. Any queries that are not obvious translation mistakes will be raised within the work-package group for consideration. (If feasible, this translation will use automated transformation tools so that it is easily repeatable.)
7. Verification of safety properties in UML-B model

   The Rodin verification tools will be used to verify that the ERTMS UML-B model satisfies the safety properties. This stage will rely on reuse of techniques developed in step 4.

York

We will iteratively and incrementally extend the metamodel for xUML, and will develop metamodels for our chosen verification languages CSP, Circus, CSP || B and PROMELA. Based on this, we will develop a library of transformations for these languages using Epsilon. Moreover, when formally analysing the models using models checking tools, we will tackle the issue of making a relationship between the formal verification results and the original xUML models. Therefore, we will also build a graphical front end for the xUML language which may also allow visual feedback of failures in model validation to be displayed.