# Rail to Digital automated up to autonomous train operation

## D30.4 – ERTMS L3 trackside with moving block formalisation

Due date of deliverable: May 28, 2026
Actual submission date: March 23, 2026

Leader/Responsible of this Deliverable: Arne Borälv (TRV)

Reviewed: Yes

| Document status | | |
|---|---|---|
| Revision | Date | Description |
| 01 | Sept 24, 2025 | First draft. |
| 02 | Jan 23, 2026 | Version for WP30 review. |
| 03 | Jan 26, 2026 | Updated version for WP30 review. |
| 04 | Jan 28, 2026 | Updated version for WP30 review. |
| 05 | Feb 16, 2026 | Added performance observations section, various edits. |
| 06 | Feb 18, 2026 | Editorial edits, added figures 4 and 9, updated executive summary. |
| 07 | Mar 8, 2026 | Updates based on review comments by DLR, SNCF. |
| 08 | Mar 20, 2026 | Updates following WP30 review meeting. Updated 1.4, 2.2, 9.3, 9.4, and executive summary. |
| 09 | Mar 23, 2026 | Accepted change markers, minor edits from WP30 review. |
| 10 | Mar 23, 2026 | Added 8.1.7 and small addition to 9.3. Minor language edits. |
| 11 | Apr 20, 2026 | Addressed TMT review comments. |



| Project funded from the European Union's Horizon Europe research and innovation programme | | |
|---|---|---|
| Dissemination Level | | |
| PU | Public | X |
| SEN | Sensitive – limited under the conditions of the Grant Agreement | |

Start date: 01/12/2022 Duration: 42 months

## ACKNOWLEDGEMENTS

This project has received funding from the Europe's Rail Joint Undertaking (ERJU) under the Grant Agreement no. 101102001. The JU receives support from the European Union's Horizon Europe research and innovation programme and the Europe's Rail JU members other than the Union.

## REPORT CONTRIBUTORS

| Name | Company | Details of Contribution |
|---|---|---|
| Arne Borälv | Trafikverket | Author |
| Daniel Schwencke | DLR | Reviewer |
| Thomas Jousselin | SNCF | Reviewer |
| Abdul Rasheeq | DB InfraGO | Reviewer |
| Stalin Castelino | DB InfraGO | Reviewer |
| Sai Aditya Bommakanti | DB InfraGO | Reviewer |

## EXECUTIVE SUMMARY

This deliverable (D30.4) accompanies the video demonstrator D30.3 and provides a technical description of the modelling techniques, abstraction choices, and validation activities developed within Europe's Rail Joint Undertaking project R2DATO (WP30, Task 30.2). Together, the two deliverables present a software-based prototype intended to support early, model-based validation of requirements for evolving railway signalling systems-of-systems.

The long-term evolution of ETCS-based trackside systems is driven by harmonisation across Europe, replacement of legacy solutions, shifts in system structure and interfaces, and the introduction of new technologies such as automatic train operation and moving block principles. These drivers do not simply add functionality but alter the technical and operational premises on which earlier generations were engineered.

Each new generation must therefore confront fundamental questions: Which legacy requirements remain essential, and which reflect past design constraints? Which architectural, migration, and technology options should be pursued — not only for the next deployment step, but in anticipation of further evolution? Without structured modelling support, requirements from one generation risk becoming implicit constraints for the next, limiting architectural freedom and obscuring opportunities for simplification or improvement.

This creates a clear need for abstraction-driven, model-based methods that enable architectural alternatives, migration strategies, and new technology requirements to be assessed objectively — before detailed design commitments are fixed.

The core contribution of this work is a modelling methodology and abstraction level for early, repeatable validation of signalling requirements in evolving system-of-systems contexts. The approach builds on a pre-existing object model methodology previously applied in fixed-block interlocking contexts and extends it by introducing ontology-driven modelling and mechanically generated object model interfaces that support both executable and formal analysis.

At its foundation is an ontology-based framework. A domain-independent upper ontology defines global, architectural, and design concepts, while a domain-specific lower ontology introduces concepts for ETCS-based trackside system-of-systems architectures. From the combined ontology, object model interfaces are generated mechanically, exposing a simple operational view — object types with explicitly typed attributes — while internalising the structural complexity required to support modularity, reuse, and controlled abstraction (inheritance, mixins, and specialisation). This interface forms a stable semantic basis for expressing requirements, behavioural models, test cases, and formal properties consistently and without reinterpretation.

Domain specialisation focuses on abstractions central to requirements-level signalling analysis, including train-centric state representations, path-based movement constraints, forbidden paths, non-nominal situations, and join/split contexts. Subsystem interaction is modelled using a message-based abstraction described in this report; its end-to-end implementation is intentionally partial.

The demonstrator has been instantiated using real infrastructure data provided by Trafikverket. RDF-based configuration data are translated into an LCF topology and used to instantiate the object model. Validation combines executable testing of automatically generated movement scenarios with targeted formal verification of selected property classes, including configuration consistency, ontology-level invariants, lifecycle and typing constraints, authority-overlap rules, forbidden-path constraints, and correct evolution of train-centric state (e.g., sweeping of unknown areas and join/split contexts). These activities were applied uniformly across all available infrastructure data sets, including the largest examples, demonstrating that the abstraction and validation mechanisms remain tractable at realistic scale.

The validation scope is intentionally limited. The work does not provide complete behavioural models, protocol verification, or operational performance evidence. Instead, it demonstrates that carefully chosen ontology-driven abstractions, combined with object model generation and integrated executable and formal validation, can support early, repeatable, and scalable reasoning about evolving signalling systems-of-systems.

The abstraction realised in the demonstrator is deliberately general. More constrained architectural variants, such as route-based ETCS configurations or hybrid train detection schemes, can plausibly be represented as refinements within the same framework by introducing additional constraints in the lower ontology and behavioural models, while preserving the upper ontology, object model generation, lifecycle semantics, and validation infrastructure. The most general abstraction is defined once; architectural variants are obtained by constraining it rather than redefining it.

Taken together, D30.3 and D30.4 present a coherent proposal for a scalable modelling methodology that enables earlier, more systematic validation of signalling requirements while maintaining a clear separation between abstraction design, prototype realisation, and future architectural evolution. The same abstraction framework also provides a structured basis for automated compliance checking of requirements, supporting the longer-term objective of automated authorisation.

## ABBREVIATIONS AND ACRONYMS

| | |
|---|---|
| AB | Authority Base |
| AC | Architecture Concept |
| API | Application Programming Interface |
| AP | Authority Path |
| BT | Behaviour Tree |
| CBTC | Communication-Based Train Control |
| CUT | Central Unit Train |
| DAG | Directed Acyclic Graph |
| DC | Design Concept |
| DS | Directed Segment |
| ERA | European Union Agency for Railways |
| ETCS | European Train Control System |
| FM | Formal Methods |
| GO | Global Object |
| HLL | High-Level Language |
| IXL | Interlocking |
| LCF | Layout Configuration Format |
| MSFOL | Many-Sorted First Order Logic |
| NN | Non-Nominal |
| OC | Object Controller |
| OM | Object Model |
| PO | Proof Obligation |
| R2DATO | Rail to Digital & Automated up to Autonomous Train Operation |
| RBC | Radio Block Center |
| RDF | Resource Description Framework |
| sHLL | Statements for HLL |
| SoS | System of Systems |
| SWEPT | Swept Path |
| TCG | Test Case Generation |
| TD2.3 | Technology Demonstrator 2.3: Moving Block |
| TD2.7 | Technology Demonstrator 2.7: Formal Methods and Standardisation |
| TLP | Train Location Path |
| TMS | Traffic Management System |
| TPR | Train Position Report |
| TRL | Technology Readiness Level |
| TTD | Trackside Train Detection |
| V&V | Verification and Validation |
| WP | Work Package |

# TABLE OF CONTENTS

## LIST OF FIGURES



## LIST OF TABLES

# 1 INTRODUCTION AND SCOPE

This document constitutes Deliverable 30.4 “ERTMS L3 trackside with moving block formalisation” in the framework of WP30 of Flagship Project 2 (R2DATO).

This document is the work of Task 30.2 in WP30.

## 1.1 PURPOSE OF THE DELIVERABLE (RELATION TO D30.3)

This deliverable (D30.4) provides a written technical description of the modelling techniques, abstractions, and validation activities underlying the software-based prototype demonstrator presented in deliverable D30.3. Together, these two complementary deliverables document work performed within the Europe’s Rail Joint Undertaking project R2DATO on early, model-based validation of requirements for evolving railway signalling system architectures.

While D30.3 presents the demonstrator primarily through a conceptual narrative and visual explanation, the purpose of D30.4 is to serve as a **technical reference**. It describes the modelling framework in more detail, clarifies the abstraction choices made, and documents the scope of implementation and validation achieved. The report is intended for readers with technical interest in railway signalling systems, modelling methodologies, and validation and verification (V&V) techniques.

This report does not aim to provide a reference design, a production-ready system specification, or a complete implementation of a signalling system. Instead, it documents a **prototype demonstrator** and the modelling principles it embodies.

## 1.2 PROBLEM CONTEXT AND VALIDATION CHALLENGES

Railway signalling systems, and ETCS trackside systems in particular, are subject to long-term evolution driven by multiple factors, including harmonisation across Europe, replacement of legacy solutions, evolving system architectures and interfaces, and the introduction of new technologies. These developments increasingly lead to system-of-systems contexts, in which multiple interacting subsystems must be considered together rather than in isolation.

In this report, the term “ETCS trackside system” is used in a modelling sense to denote the trackside system-of-systems context within which an RBC operates. Depending on architectural variant, this context may include separate or integrated interlocking (IXL) functionality, object controllers, and interfaces to traffic management systems. The modelling approach is intentionally architecture-neutral and allows different separations or integrations of RBC and IXL components to be represented within the same conceptual framework.

In such contexts, requirements are often specified at a high level and across organisational and technical boundaries. Experience from prior projects has shown that issues such as inconsistency, ambiguity, and incompleteness in requirements are frequently discovered only in later development phases, when design and implementation decisions have already been made. This late discovery increases cost, complexity, and risk.

At the same time, modelling approaches used in early phases often face a difficult trade-off. Informal models may be expressive and easy to use, but lack the precision needed for rigorous validation. Highly formal models may offer strong verification capabilities but require premature design commitments or are difficult to scale to complex, evolving architectures.

## 1.3 Scope and Non-Goals of the Demonstrator

The work documented in this report addresses the problem of early requirements validation, with a particular focus on identifying a suitable abstraction level for analysing evolving railway signalling systems-of-systems. The scope of the demonstrator and this report includes:

- **Definition of an ontology-based modelling framework for railway signalling systems**, comprising an upper ontology and a domain-specific lower ontology, including concepts for global structure, domain modelling, and system architecture.
- **Use of a generated object model as a common semantic foundation** for expressing requirements, behavioural models, test cases, and safety properties.
- **Conceptual modelling and instantiation of architectural concepts (AC)**, such as containers, factories, communication links, and message types, to support system-of-systems structuring and allocation of domain objects, without prescribing a complete system architecture.
- **Description of modelling abstractions for ETCS trackside**, including train-centric representations, path-based constraints, forbidden paths, and explicit modelling of non-nominal situations.
- **Conceptual modelling of behaviour and interaction**, including message-based interaction between subsystems and a primitive execution model.
- **Instantiation of the modelling framework using real infrastructure data examples** and execution of selected validation activities.

The following aspects are explicitly out of scope for this deliverable:

- Definition of a complete signalling system architecture or reference design.
- Full behavioural modelling of all subsystems and interactions.
- Exhaustive validation of behavioural properties or operational performance.
- Demonstration of production-ready implementations or interfaces.

Several parts of the modelling framework described in this report are specified at a conceptual level and are only partially realised in the demonstrator implementation. This applies in particular to behavioural models and the message-based interaction model. The report makes this distinction explicit throughout.

## 1.4 Relation to Prior Work

The modelling approach described in this report builds on a pre-existing object model–based methodology that has previously been applied in railway signalling contexts, including fixed-block interlocking systems [11][18], where it was used to separate conceptual structure from behavioural logic and validation activities. In those earlier applications, the object model served as a shared, typed vocabulary: requirements and behavioural models were expressed over the same object and attribute concepts, enabling systematic executable testing and formal analysis without repeated reinterpretation between artefacts.

The work in Shift2Rail TD2.7 provides relevant background and motivation in two complementary ways. First, TD2.7 produced methodological guidance emphasising early, model-based reasoning and the value of prior-to-construction analysis for safety-critical signalling systems. In particular, [8]

proposed procurement-oriented methodologies aimed at reducing ambiguity in tender requirements and enabling more automated compliance verification through semi-formal and formal models, with an emphasis on reuse, modularity, and reduced dependence on manual interpretation.

Second, TD2.7's analyses of moving-block requirements and configuration data (notably [3], [4], and [5], [6]) exposed practical challenges in defining clear domain concepts, separating safety-related assumptions from architectural choices, and validating configuration-dependent behaviour in a scalable way. These studies demonstrated both the usefulness of ontology-based modelling and the difficulty of applying formal analysis when domain abstractions are not sufficiently precise or modular. The insights gained from this work directly informed the abstraction level adopted in the present report.

R2DATO generalises and operationalises these insights for evolving signalling systems-of-systems by introducing an ontology-based modelling framework and mechanically generated object model interfaces. The ontology provides a modular and reusable conceptual definition of the system domain; the generated interfaces expose a stable operational view—object types with typed attributes—supporting executable validation and formal verification over the same semantic foundation. In this sense, the contribution of D30.3/D30.4 is not to restate prior methodological guidance, but to provide a concrete abstraction and tool-supported workflow that can be exercised on realistic infrastructure data and incrementally extended as architectures evolve.

The relation to the earlier object model methodology is discussed in more detail in Section 2.2, and the operational realisation through generated object model interfaces is described in Section 4.6.

## 1.5 Structure of the Report

The remainder of this report is structured as follows:

- Chapter 2 introduces the proposed abstraction level and discusses the trade-off between expressivity and verifiability.
- Chapter 3 describes the ontology-based modelling framework, including the upper and lower ontologies.
- Chapter 4 presents the object model methodology, attribute semantics, and behavioural separation.
- Chapter 5 discusses behavioural modelling and interaction at a conceptual level.
- Chapter 6 describes instantiation from infrastructure data and visualisation.
- Chapter 7 summarises the validation activities performed.
- Chapter 8 discusses maturity, limitations, reuse potential, and challenges.
- Chapter 9 concludes the report and outlines possible future work.
- Chapters 10–13 provide appendices containing selected technical details, generated interface excerpts, formal model representations, and illustrative proof obligations that complement the main text.

## 1.6 Technology Readiness Level

The demonstrator achieves TRL 3 by providing an experimental proof of concept for an ontology-based object model methodology intended to support early requirements validation for evolving

railway signalling systems-of-systems. The work demonstrates feasibility of the proposed abstraction and modelling discipline through executable and formal analysis, including instantiation using real infrastructure data and validation of selected representative scenarios.

The achieved TRL reflects validation of a methodological and abstraction-level concept under realistic conditions, rather than readiness for operational deployment.

# 2 PROPOSED MODELLING APPROACH AND LEVEL OF ABSTRACTION

This chapter introduces the modelling approach proposed in this work and clarifies the level of abstraction at which it operates. Rather than presenting a concrete system design, the chapter focuses on the conceptual choices that shape how evolving railway signalling systems-of-systems are represented, analysed, and validated at an early stage.

The discussion motivates the approach by examining the tension between expressivity and verifiability in early-phase modelling, relates the work to prior object-model-based methodologies, and explains how the approach is extended to support dynamic behaviour and system-of-systems contexts. The chapter then positions the ontology as the primary carrier of abstraction and explains how this abstraction is realised operationally through object model generation. Finally, the relation to the generic formal methods process is discussed, situating the proposed approach within a broader validation workflow.

## 2.1 MOTIVATION: EXPRESSIVITY VS VERIFIABILITY

Modelling approaches for railway signalling systems must navigate a fundamental tension between **expressivity** and **verifiability**. On the one hand, models must be expressive enough to capture dynamic system behaviour, operational rules, and interactions across multiple subsystems in a system-of-systems context. On the other hand, models intended to support rigorous validation and verification (V&V) must have sufficiently precise semantics to enable systematic analysis, including executable testing and formal verification.

Experience from earlier projects has shown that informal or semi-formal models—such as diagrams, textual descriptions, or loosely specified behavioural representations—are often effective for communication and early exploration but provide limited support for rigorous validation. Conversely, highly formal models can offer strong verification guarantees, but frequently require early commitment to detailed architectural or design choices, or become difficult to scale as architectures evolve and system contexts become more heterogeneous.

The work documented in this report adopts the position that effective early requirements validation depends primarily on the **choice of abstraction level**. In this report, abstraction refers to the deliberate selection and structuring of system concepts and relationships that are represented explicitly in the model, while omitting details that are not relevant for the validation task at hand. Rather than maximising either expressivity or formality, the aim is to establish a modelling abstraction in which the relevant system concepts and their relationships are defined explicitly and independently of specific architectural realisations or behavioural logic. Required behaviour can then be formulated over this stable conceptual structure.

By separating conceptual structure from behavioural rules—and from concrete architectural configurations—the approach seeks to enable automated and objective analysis of requirements without prematurely committing to particular implementation designs. This separation also supports controlled variation of architectural assumptions while preserving a consistent validation framework

## 2.2 RELATION TO PRIOR OBJECT MODEL METHODOLOGY (FIXED BLOCK)

The abstraction strategy outlined in Section 2.1 builds on an **object model–based methodology** previously developed and applied in railway signalling contexts, including fixed-block interlocking systems and the Shift2Rail TD2.7 Alex level crossing case study.

The methodology was designed precisely to separate conceptual system structure from behavioural logic and validation activities. The conceptual structure is defined by an object model that provides a shared, typed interface consisting of explicitly defined object types and their attributes. Requirements and behavioural models are then expressed consistently in terms of this object model, clarifying the roles of engineering artefacts and V&V activities (see Table 1).

Because both behavioural models and requirements are formulated over the same typed object model, validation can be automated: consistency checks, executable tests, and formal properties are defined directly over a shared semantic foundation, eliminating the need for reinterpretation between artefacts.

| Artefact | Role in the methodology |
|---|---|
| **Object model** | Shared vocabulary and typed interface |
| **Requirements** | Statements formulated using that vocabulary |
| **Behavioural model** | Executable logic operating over that vocabulary |
| **Validation & Verification** | Automated checking of behaviour against requirements expressed in the same vocabulary |

**Table 1 Role of the object model as a shared semantic foundation**

In earlier applications, the conceptual system domain was described using a precise object model that served as the common basis for:

- requirements specification,
- behavioural descriptions,
- executable test cases, and
- formal safety properties.

A key strength of the methodology is the separation between:

- what the system is (conceptual structure),
- what it is required to do (requirements), and
- how it is implemented (behavioural realisation).

By defining objects, attributes, and relations explicitly at the conceptual level, the object model provides a stable vocabulary in which requirements and behaviour can be expressed consistently. This supports early detection of inconsistencies, reduces ambiguity, and limits manual reinterpretation across validation activities.

The original methodology was developed in relatively constrained interlocking contexts, characterised by fixed-block signalling principles, limited architectural variability, and a primary focus on individual subsystems rather than full system-of-systems interactions.

The work in R2DATO aims to preserve the core benefits of this methodology while extending its applicability to more dynamic, configurable and heterogeneous signalling architectures.

In addition to interlocking-related applications, the methodology was used in Shift2Rail TD2.7 in the **Alex level crossing case study**. There, it served as a common foundation for defining tender

requirements [16] against which software models developed along multiple development tracks were verified. V&V activities included formal verification of safety-labelled tender requirements and execution-based testing using test cases generated from an object model–based test case generation (TCG) model. Relevant publications from TD2.7 [9][10] report on the modelling approach and validation results.

## 2.3 Extending the Methodology to Systems-of-Systems

To support evolving ETCS trackside systems and related signalling architectures, the object model methodology must be **generalised along several dimensions**. In particular, the modelling approach must accommodate:

- **dynamic object types**, in contrast to the purely static object models used in the original methodology;
- **configurable system architectures**, in which subsystems, interfaces, and responsibilities may change over time;
- **explicit system-of-systems interaction**, with controlled modelling of information exchange between subsystems; and
- **richer abstractions of train movement**, such as path-based representations that go beyond fixed-block assumptions.

Rather than addressing these challenges by increasing model complexity indiscriminately, the demonstrator adopts a **principled extension** of the original methodology. The object model remains the central semantic reference, but its expressive power is increased in a controlled manner through:

- an **ontology-based definition** of the conceptual system domain, separating generic (upper ontology) and domain-specific (lower ontology) concepts;
- **explicit modelling of object lifecycles and attribute semantics**; and
- a **clear separation between the object model and behavioural models** that operate on instantiated objects.

This extension seeks to retain as much as possible of the verifiability benefits of the original methodology while enabling analysis of more dynamic and system-of-systems scenarios.

## 2.4 The Proposed Level of Abstraction

Taken together, the modelling approach described in this report can be understood as proposing a pragmatic level of abstraction for reasoning about evolving railway signalling systems-of-systems.

### 2.4.1 Ontology as the Conceptual Level of Abstraction

The proposed level of abstraction in this work is defined by the **ontology**. The ontology determines which concepts are represented, how they are related, and which distinctions are considered relevant for analysis. In doing so, it fixes the modelling scope and expressivity and makes explicit the abstraction choices that underpin all subsequent modelling, validation, and verification activities.

At this level, abstraction decisions include, for example:

- how track topology and train movement (and related movement constraints) are represented;
- which objects are considered static or dynamic;

- how non-nominal situations are captured; and
- which aspects of system behaviour are intentionally left out of scope.

These decisions are not incidental. They directly affect what properties can be formulated and which kinds of analysis are feasible. By making such choices explicit in the ontology, the modelling approach avoids implicit assumptions being scattered across behavioural models, test cases, or verification artefacts.

In this sense, the ontology serves as the **primary carrier of abstraction**. It reflects a shared conceptual view of the system domain, is designed to be understandable to multiple stakeholders, and is sufficiently precise to support rigorous analysis. The abstraction level is therefore chosen consciously at the ontology level, rather than emerging implicitly from implementation or tooling constraints.

### 2.4.2 Object Model as the Operational Realisation

The **object model** is generated from the ontology and provides the **operational realisation** of the chosen abstraction.

Object model generation absorbs the structural complexity introduced in the ontology – such as inheritance, attribute sharing, and refinement mechanisms[1] – and exposes a simple and uniform modelling interface. The resulting object model consists only of object types with explicitly typed attributes that can be read and written, allowing modelling and validation activities to operate without requiring awareness of the underlying ontology structure.

These interfaces are produced for the target languages used in the demonstrator—Python, used to instantiate concrete object models and to carry out executable, test-based analysis, and HLL for formal analysis—and present a fully resolved view of the ontology's concepts. They are suitable for instantiation, behavioural modelling, and validation. In this process, the internal ontology structure is intentionally hidden from users of the object model, who instead interact with a uniform and homogeneous interface consisting of object types with explicitly typed attributes.

From a methodological perspective, object model generation is primarily a **technology mapping step**. The conceptual structure and abstraction decisions are already fixed in the ontology; the object model realises these decisions in a form that can be executed, analysed, and verified. This separation clarifies responsibilities: changes to abstraction scope or modelling assumptions are made at the ontology level, while the object model ensures consistent and traceable propagation of those changes into executable and formal artefacts.

By distinguishing clearly between ontology (as abstraction) and object model (as realisation), the modelling approach supports both conceptual clarity and practical applicability. This distinction underpins the integration of ontology definition (Chapter 3), object model generation (Chapter 4), behavioural and property modelling (Chapter 5), and model instantiation and validation (Chapters 6–7).

## 2.5 Relation to the Generic Formal Methods Process

Earlier work in Shift2Rail TD2.7 describes a **generic process characterising formal methods (FMs) application**, intended to be independent of purpose, lifecycle phase, notations, tools, or

[1] The structural complexity required in the ontology to support modularity, reuse, and disciplined abstraction.

degree of formality. The process is deliberately non-prescriptive and comprises a set of generic activities—Define Ontology, Define Properties, Create Model, Perform V&V, Produce Output, and Improve (see Figure 1)—that may be instantiated with different emphasis and effort depending on context and reuse.

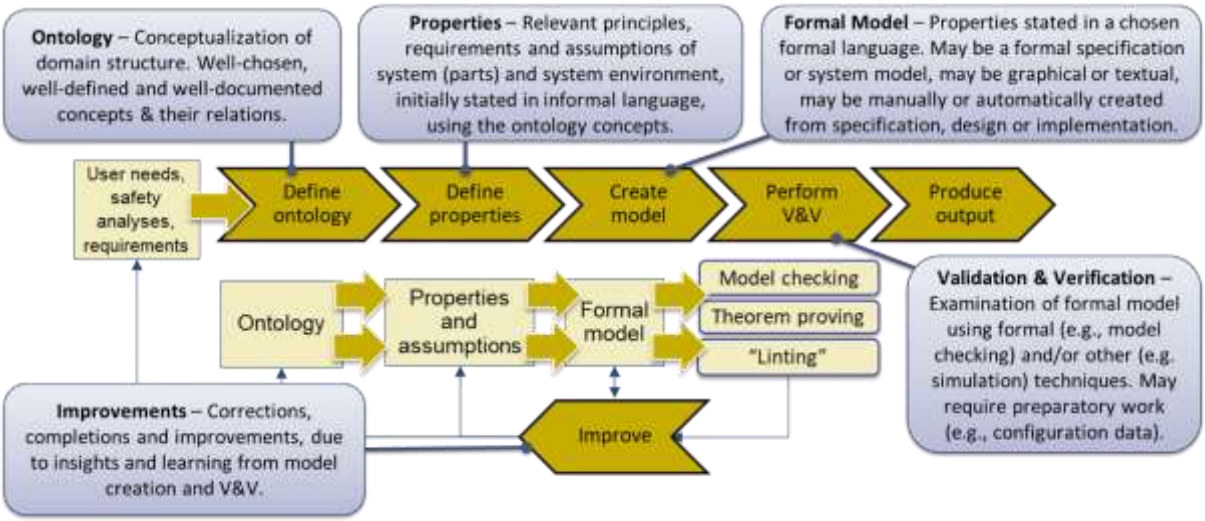



**Figure 1: Generic FMs process (with brief explanation of its steps)**

The modelling approach used in the demonstrator is best understood as providing a **common, validation-oriented infrastructure** that supports several of these generic activities, without privileging a particular validation technique. In particular, the demonstrator provides mechanisms to define an ontology and to mechanically transform it into a sufficiently detailed modelling interface that can be used to express both properties and required behaviour, enabling different forms of validation and analysis as appropriate to context.

The **Define Ontology** activity is supported through the explicit definition of an ontology using well-defined class, mixin, and specialisation constructs. As described in Section 2.4.1, the ontology establishes the conceptual system domain and fixes the chosen level of abstraction, reflecting a shared conceptual view and making modelling scope and assumptions explicit.

Based on the defined ontology, **system properties and principles** may be formulated in terms of the object model interface, which provides a common semantic foundation across validation activities. Such properties may be expressed and evaluated in different contexts, including executable test-based validation using Python, formal analysis using the generated HLL model, or other analysis techniques as appropriate. Properties may also be stated at the ontology level to capture generic consistency principles independent of a particular model instance.

The **Create Model** activity is supported through automatic object model generation, which realises the ontology-defined abstraction in concrete modelling interfaces. Object model generation resolves inheritance, mixins, and specialisations and produces executable and formal representations that can be instantiated with concrete data and exercised using different validation techniques.

The demonstrator further provides **partial support for the Perform V&V activity**. This includes executable validation—such as scenario-based testing, simulation, and graphical visualisation—as well as formal checking of selected classes of properties, including type correctness and generic consistency, through automatically generated proof obligations discharged against the HLL model.

Iterative refinement of ontology definitions, modelling choices, and validation artefacts—corresponding to the **Improve** activity—occurred throughout the work and is reflected in the design principles and empirical findings discussed in Chapter 8.

In this light, the demonstrator can be seen as a concrete instantiation of the **System Inception** use case described in the FMs guidebook [7]: supporting early-phase reasoning and validation before requirements are finalised, using a shared conceptual foundation and executable models to explore assumptions, scenarios, and properties, while remaining neutral with respect to the specific validation techniques applied.

## 2.6 What This Abstraction Enables

Within its intended scope, the proposed abstraction enables:

- early, model-based validation of requirements prior to construction;
- consistent expression of requirements, behavioural models, and test cases using a shared object model;
- executable, test-based validation supported by runtime checking of modelling constraints;
- formal analysis of selected properties using generated formal models;
- analysis of dynamic behaviour, including creation and deletion of runtime entities; and
- exploration of architectural and migration options in system-of-systems contexts.

Together, these capabilities support a shift of validation activities towards earlier development phases, where changes are less costly and design freedom is greater.

## 2.7 What This Abstraction Does Not Aim to Do

Equally important are the deliberate limitations of the approach. At the proposed abstraction level, the modelling framework does **not** aim to:

- define complete behavioural models for all subsystems;
- prescribe specific deployment architectures or allocation of functions;
- provide production-ready implementations or interfaces;
- replace detailed design, implementation, and performance engineering activities; or
- serve as a stand-alone operational model of a trackside system (of systems).

Several aspects of the modelling framework – particularly behavioural modelling and message-based interaction – are described conceptually and are only partially realised in the demonstrator implementation. These limitations are conscious design choices, reflecting the prototype nature of the demonstrator and the focus on modelling techniques rather than complete system realization.

## 2.8 Reading Guide for the Remainder of the Report

The remainder of this report should be read with the abstraction level defined in this chapter in mind:

- Chapter 3 describes the ontology-based modelling framework that defines the conceptual system domain.

- Chapter 4 explains how the object model is generated and used as a common semantic foundation for modelling and analysis.
- Chapter 5 discusses behavioural modelling and interaction at a conceptual level, including partial realisation.
- Chapters 6 and 7 present instantiation from infrastructure data and the validation activities performed.
- Chapters 8 and 9 discuss maturity, limitations, reuse potential, and future work.

Readers primarily interested in results and implications may focus on Chapters 6–9, while Chapters 3–5 provide the methodological and technical foundations underlying those results.

Throughout the report, distinctions are made between **conceptual models**, **implemented mechanisms**, and **validated results**, to ensure clarity regarding scope and maturity.

# 3 ONTOLOGY FRAMEWORK

This chapter describes the ontology framework that underpins the modelling approach presented in this report. The ontology defines the conceptual system domain, including the types of objects, relations, and modelling mechanisms that are considered relevant at the chosen abstraction level.

The chapter introduces the structure of the ontology, including the distinction between a domain-independent upper ontology and a domain-specific lower ontology, and explains how classes, attributes, mixins, and specialisation are used to support modularity, reuse, and constraint-preserving refinement. The focus is on the conceptual structure and modelling principles embodied in the ontology; implementation details and generated object model interfaces are covered in subsequent chapters and appendices.

## 3.1 OVERVIEW AND DESIGN INTENT

The demonstrator is built using an **ontology framework** that was developed – from scratch – for prototype-oriented exploration of modelling techniques for early requirements validation. This chapter describes the ontology framework, as designed and implemented. The ontology framework supports definition of upper and lower ontologies, and generation of object-model interfaces. Design choices and trade-offs motivated by verifiability and expressivity are documented here as part of the modelling approach; findings that emerged through experimentation are discussed separately in Chapter 8.

The framework supports definition of the **conceptual system domain** as an ontology and automated generation of a corresponding object model. A central architectural decision is to divide the ontology into:

- an **upper ontology**, defining a *universe of general types* and modelling mechanisms that provide a structured, reusable foundation for dynamic railway signalling contexts, agnostic to specific signalling principles while compatible with formal analysis; and
- a **lower ontology**, specialising the upper ontology with domain-specific concepts for ETCS trackside and the modelling of train movement–related requirements.

From the ontology, the framework generates a generic object model interface in two target languages:

- **Python**, used to instantiate models and support executable validation (test-based V&V); and
- **HLL**, used to represent a formal model in a many-sorted first-order logic setting, supporting formal verification of selected properties.

The framework also includes mechanisms that support systematic reasoning about ontology correctness and validity of instantiated models.

The framework was developed as a prototype and focused on exploring ideas and modelling mechanisms rather than delivering a product-grade toolchain.

## 3.2 UPPER ONTOLOGY: UNIVERSE OF TYPES

The upper ontology defines a set of general types and modelling mechanisms intended to provide a structured, reusable foundation for modelling dynamic railway signalling contexts. The upper ontology is deliberately agnostic to specific signalling principles, while introducing the concepts

required to express requirements and behaviour for scenarios that go beyond fixed-block assumptions, including dynamic objects and path-based movement abstractions.

A key input to the upper ontology is the **Layout Configuration Format (LCF)**, which provides a formal, graph-based representation of track topology and related static configuration data used in fixed-block trackside contexts. The upper ontology extends this basis to support additional abstractions relevant for moving block analysis constructs and system-of-systems modelling.

In the demonstrator, LCF is used as an intermediate representation of infrastructure data (translated from RDF, see Section 6.1) prior to model instantiation. This choice is aligned with risk management considerations. As a formally defined and publicly available language with precise semantics, LCF provides an unambiguous representation of the translated data, reducing the likelihood of misinterpretation and inconsistencies in subsequent processing. The published LCF specification includes explicit correctness requirements, enabling systematic structural and semantic validation of the translated models and supporting early identification of modelling inconsistencies and data quality issues. LCF has been applied in operational contexts and is supported by SIL 4–qualified tools for sign-off verification. In addition, conversion tools targeting LCF exist, and further translators can be developed with reasonable effort. LCF was designed as a general-purpose intermediate format for fixed-block representations, making it well suited as a stable interchange layer between heterogeneous data sources and downstream analysis and verification environments.

### 3.2.1 LCF: Graph-Based Representation of Track Topology and Static Data

LCF is a language for specifying static configuration data for railway trackside systems, supporting both generic and specific data. In LCF, the infrastructure topology (a “railyard”) is represented as a graph of nodes and edges, together with objects located at nodes. Consistency and correctness requirements for LCF instances are defined in the publicly available LCF specification [9].

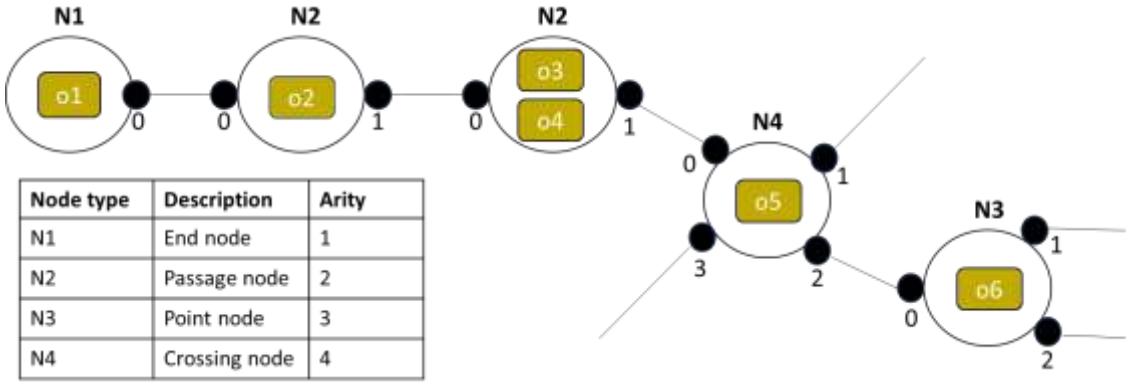
A railyard in LCF uses the following core constructs:

- **Graph representation:** a topology described as a graph of nodes and edges.
- **Nodes and objects:** each node may contain zero or more objects.
- **Node connectors:** each node type defines a set of connectors, numbered consecutively from 0.
- **Edges:** connect node connectors and are uniquely identified by pairs of endpoints.
- **Passage relations:** each node type defines permissible passages (how a train may traverse the node) as allowed connector combinations.

LCF defines **paths** as sequences of edges respecting passage relations, and **areas** as subgraphs defined as sets of edges and/or nodes. LCF also supports tabular data representations to facilitate compact review of configuration data.

Figure 2 illustrates a partial railyard graph and conventions normally used with LCF:

- For node types with arity greater than 2, connectors are numbered in clockwise order.
- For node types with arity 3 (points), the common leg is assigned connector index 0.




| Node type | Description | Arity |
|---|---|---|
| N1 | End node | 1 |
| N2 | Passage node | 2 |
| N3 | Point node | 3 |
| N4 | Crossing node | 4 |

**Figure 2: Illustration of partial railyard graph**

### 3.2.2 Extensions Introduced in the Upper Ontology

The upper ontology extends the fixed-block foundations of LCF in several directions to support the abstraction level described in Chapter 2:

- **Dynamic objects:** the modelling framework supports object types whose instances may be created and deleted at runtime (e.g., dynamic authorities, train-related runtime entities).
- **Richer path abstractions:** paths may begin and end at arbitrary positions, supporting moving block–style representations rather than only block-delimited extents.
- **Configurable architectures and environments:** concepts are introduced to represent architectural components and their interaction, supporting system-of-systems modelling and evolution.
- **Time-related modelling:** a timer concept supports representation of real-time measurement.

The intent is to provide a reusable general foundation, applicable not only to ETCS-related scenarios but, in principle, also to similar signalling domains such as CBTC.

### 3.2.3 Base Type Categories

At a high level, the upper ontology defines a constrained universe of types organised into the following categories:

- **Global Objects (GO):** track topology primitives used as the common basis for referencing infrastructure and defining extents of areas and paths.
- **Architecture Concepts (AC):** modelling of system architecture elements such as containers and communication links, and the allocation of objects to factories (i.e., dedicated lifecycle-management contexts for object creation and deletion; see Sections 4.3 and 6.4).
- **Design Concepts (DC):** modelling constructs used by behavioural models and design-level representations, including topology-related and non-topology-related objects.
- **Non-Nominal (NN) concept:** explicit modelling concept for representing non-nominal situations detected during runtime.

These categories provide the general conceptual foundation that is specialised in the lower ontology.

### 3.2.4 Naming convention

In the upper ontology, type names prefixed with U_ denote *upper-ontology interface types*. These types define abstract roles and structural contracts that are independent of any specific domain or deployment. They are not intended to be instantiated directly, but to be specialised by lower-ontology types or used as structural anchors in the object model.

This convention is used consistently across the upper ontology for architectural, topological, and interaction-related concepts (e.g. U_Node, U_Edge, U_Container, U_Part, U_Factory, U_Link). The prefix is a naming aid rather than a semantic modifier; the semantics of each type are defined by its attributes, mixins, and specialisations.

The inheritance structure of architecture concepts in the upper ontology is illustrated in Appendix A, which provides a consolidated view of scoping, factories, containers, and communication links.

## 3.3 Ontology Structure and Modelling Mechanisms

This section describes the **ontology structuring mechanisms** used in the framework. The ontology is represented as a class structure with **single inheritance**, complemented by two orthogonal mechanisms: **mixins** (attribute import) and **attribute specialisation** (restriction).

### 3.3.1 Classes as the Structural Backbone

The ontology is organised as a class hierarchy rooted in a single root class. Each class inherits from exactly one parent, reflecting a clear *is-a* relationship. Only **leaf classes**—that is, classes without subclasses—are instantiable.

In the upper ontology, leaf classes serve as **interface types** intended to be inherited and specialised by the lower ontology.

### 3.3.2 Attributes and Attribute Categories

Classes are characterised by **attributes**, which represent named relations defined on objects. An attribute may relate an object to either:

- a **ground type value** (Boolean, integer, enumerated type, or object reference), or
- a **set value**, defined as a (finite) set of tuples whose elements are ground-type values, with a specified signature.

Attributes are **globally unique by name**, avoiding ambiguous overloading and ensuring a uniform object model interface.

Each attribute is assigned an **attribute category** that constrains its behaviour over time:

- **Static**: fixed and not allowed to change during runtime.
- **Dynamic**: may vary over time, reflecting evolving system state.
- **Incarnation**: fixed during the lifetime (incarnation) of a dynamic object; after deletion and possible re-creation (“reincarnation”), the attribute may take a different value.

These categories support a constrained modelling style that helps maintain verifiability while allowing controlled expressivity.

In addition to this basic form, the ontology also supports **attributes with arguments**, parameterised by one or more ground-type values. Such an attribute represents a family of related values indexed

by a tuple of ground values. For example, in the demonstrator an attribute may be defined whose value is a **directed segment**, and which takes as argument a **node and a node connector index**. This allows the attribute to denote, in a typed and explicit way, the directed segment that begins or ends at a given node index.

Attributes with arguments are treated uniformly with other attributes: their arguments and return values are explicitly typed, they are assigned an attribute category (static, dynamic, or incarnation), and they are subject to the same consistency and lifecycle principles. This mechanism provides a compact and expressive way to represent indexed relations without introducing additional object types or auxiliary structures.

### 3.3.3 Reuse and Refinement of Attributes

While classes define the structural backbone of the ontology and attributes define typed relations, additional mechanisms are needed to support reuse and refinement without compromising clarity or analysability. In the demonstrator, this is achieved through a combination of **mixins** and **specialisation**, which together enable controlled sharing and refinement of attributes.

Mixins are used as a typed mechanism for sharing a common set of attributes (relations) across multiple classes, without introducing the semantic complexity associated with multiple inheritance. A mixin defines a coherent set of relations that can be used by different classes, effectively providing a shared interface across multiple object types. This allows common modelling concepts—such as lifecycle attributes, architectural relations, or behavioural interfaces—to be introduced once and reused consistently across the ontology. At the same time, maintaining a single-inheritance class hierarchy preserves a clear and analysable type structure, which is important for object model generation and for a range of validation and analysis activities.

Specialisation is used as a well-defined mechanism for refining attributes and constraints introduced by classes or mixins. A specialisation may restrict value ranges, narrow types, or strengthen constraints associated with an attribute, but it cannot introduce new attributes or relax existing constraints. In this way, specialisation is **monotonic**: it only makes modelling assumptions more precise. This ensures that refinements remain compatible with their more general definitions and allows domain-specific concepts to be introduced by tightening generic ones rather than redefining them.

In practice, specialisation involves one or more of the following forms of refinement:

- restricting the attribute category (e.g. from dynamic to incarnation or static);
- restricting the value range of an attribute;
- restricting cardinality for set-valued attributes;
- restricting the type of referenced objects; or
- defining constant or function-based attribute values.

Mixins and specialisations are explicitly named entities whose identity is preserved in the generated object model. This structure is reflected consistently across representations, appearing as sorts in the formal HLL model and as accessible type information in the Python object model. Together with single inheritance, these mechanisms form a **directed acyclic graph (DAG)**.

Together, single inheritance, mixins, and specialisation provide a **modular and expressive ontology structure** while preserving a clear, analysable type system suitable for object model generation and for systematic validation and analysis using different techniques.

### 3.3.4 Standard Principles and Core Object Semantics

To avoid null values and union types, the framework assumes that each leaf type has a dedicated **nullary object**, used as a type-safe placeholder in lieu of NULL. Nullary objects enable uniform typing and explicit semantics for representing "absence".

A core set of attributes is introduced uniformly for all objects, including:

- **nullary** — identifies nullary objects;
- **dynamic** — determines whether the type supports runtime creation and deletion; and
- **active** — indicates whether an object is currently considered to exist in the system. For static objects this value is constant, while for dynamic objects it may change over time.

These standard principles underpin both executable and formal checking of generic consistency requirements and provide a common semantic foundation for validation activities.

Selected excerpts from the ontology definition language and generated inheritance diagrams are provided in Appendix A.

## 3.4 LOWER ONTOLOGY: DOMAIN-SPECIFIC CONCEPTS

### 3.4.1 Purpose and Scope of the Lower Ontology

The purpose of the lower ontology is to define the **domain-specific concepts required to model a concrete system domain within the chosen abstraction level**, building systematically on the general concepts provided by the upper ontology. All concepts introduced in the lower ontology are defined by inheriting from, and thereby instantiating, the base types of the upper ontology. This ensures that every domain-specific concept is categorised explicitly within the upper ontology's conceptual framework.

In this way, the lower ontology provides a **complete conceptual vocabulary for its intended scope**. Domain-specific object types are assigned appropriate roles by inheriting from upper ontology base types (e.g., Global Objects, Architecture Concepts, Design Concepts) and, where needed, by importing shared interfaces through mixins and refining behaviour or constraints through specialisation. This structuring ensures that lower ontology concepts remain compatible with the modelling mechanisms, lifecycle semantics, and validation discipline established by the upper ontology.

The lower ontology is therefore not an ad-hoc collection of examples, but a **coherent specialisation of the general modelling framework**, capturing the concepts necessary to express domain-specific requirements, constraints, and interaction patterns at the selected abstraction level.

In the demonstrator, the lower ontology defines concepts relevant to **ETCS trackside systems**. The scope used is deliberately bounded and is not intended to define a complete or normative ETCS specification. Instead, it constitutes a **representative and sufficiently rich subset of concepts** chosen to exercise the modelling approach and to support early validation of requirements involving dynamic train movement, safety constraints, and system interaction.

A central aspect of the lower ontology is a **train-centric modelling abstraction**. A train concept is related to a set of paths with dynamic extent, enabling requirements and constraints to be expressed using **moving block principles**, where both the existence and extent of paths may change at runtime. These dynamic paths are defined in relation to static concepts in the trackside system of

systems. Additional dynamic-extent paths include, for example, *unknown paths* and *forbidden paths*, which are managed by the RBC but commanded by a separate traffic management system.

The demonstrator further assumes a **regularised interface model** for trackside objects, where each object type is assumed to reside on a dedicated wayside object controller. All objects of the same type—such as points, single-slip track crossings, or level crossings—therefore share a common interface for information exchange with ETCS trackside, supporting uniform modelling and validation of subsystem interaction. The interfaces considered here are at the level of logical system modelling and are intentionally independent of implementation-specific interface specifications. While initiatives such as EULYNX address harmonisation of technical subsystem interfaces, the demonstrator focuses on abstraction and behavioural consistency at modelling level rather than on compliance with a specific engineering standard.

In addition to train-centric and interaction-related concepts, the lower ontology includes a set of **typical static trackside objects and structures**, such as:

- node types of degree 1, 2, 3, and 4 (corresponding to end nodes, passage nodes, point nodes, and track-crossing nodes with different passage relations);
- temporary and permanent areas (e.g. point-lock areas and shunting areas);
- level-crossing areas (areas defined as sets of nodes);
- paths with static extent;
- balise groups; and
- signals and boards.

Together, these concepts provide a **coherent domain-specific specialisation of the general types defined in the upper ontology**, sufficient to express and validate representative ETCS trackside requirements at the chosen abstraction level.

### 3.4.2 Domain-Specific Concepts and Patterns

A recurring domain-specific pattern addressed by the lower ontology concerns **information exchange between ETCS trackside and wayside object controllers** in the trackside topology. In the demonstrator, each wayside object—such as a signal, point, or level crossing—is abstracted as being controlled by a dedicated subsystem, represented by a separate object controller container. This abstraction is used to model the trackside system as a system of interacting subsystems, while keeping the interfaces explicit and amenable to validation.

To support this pattern, the lower ontology defines concepts that enable a **static interface** between ETCS trackside (represented by an RBC container) and each class of wayside object controller. The interface is considered *static* in the sense that the relevant object types, interface attributes, and directions of information exchange are known a priori at the chosen abstraction level. The intent is not to mirror existing implementation-level interfaces or to anticipate future ones, but to define **interface abstractions that promote verifiability** and support early requirements validation in a system-of-systems context.

In the demonstrator, information exchange between subsystems is modelled uniformly using **message objects** exchanged between containers. A message is a dynamic object allocated to a factory associated with a communication link. When a message is sent, the corresponding message object is activated in the link's factory; once the message has been received and processed, the object is deactivated. As a consequence of this execution model, **all message types are defined**

**statically**, including the attributes they carry, while individual message instances are created and consumed dynamically during execution.

As part of the lower ontology, message types are defined for interfaces between ETCS trackside and wayside object controllers. These message types are attribute-based, such that each message modifies exactly one attribute value. This supports precise and fine-grained modelling of information exchange and avoids ambiguity in message semantics. The concrete structure of message types and their payload patterns is described in Section 5.3.1.

The ontology framework provides a **uniform interface abstraction** to represent such static interfaces. Conceptually, an interface is defined by:

1. a **generic interface specification**, consisting of a set of attributes labelled as *input* or *output*; and
2. a **master object**, belonging to a master container (e.g. ETCS trackside), with an associated slave container (e.g. a wayside object controller).

The relation between the master object and the interface is not modelled as an is-a relation, nor as a single has-a relation. Instead, the ontology supports a *has-attributes-of* relationship, allowing a class to declare the attributes that constitute the semantic contract of an interface without prematurely introducing a concrete interface structure. From this declaration, the ontology framework can generate a **first-class interface entity** as a refinement step, associated with the master object and its container. In principle, this refinement can itself be subjected to objective verification, ensuring that the generated interface faithfully reflects the declared attribute contract.

At runtime, the master and slave containers exchange messages to modify the interface attributes. A key modelling principle is that the **master object maintains the current value of all interface attributes**, both inputs and outputs. Incoming messages modify the corresponding attribute values unconditionally when processed. How a slave container reacts to changes in output attributes is intentionally outside the scope of the interface abstraction and belongs to the slave's internal behaviour model.

This interface pattern is inspired by the *relay-based interface* used in the tender specification for the Alex level crossing system in Shift2Rail TD2.7. Each such interface can be depicted graphically as a box with incoming and outgoing arrows, one per attribute, providing a clear and intuitive representation of the information exchange contract (see Figure 3). At the chosen abstraction level, all interface attributes are of ground type, further supporting precise semantics and verifiable modelling.

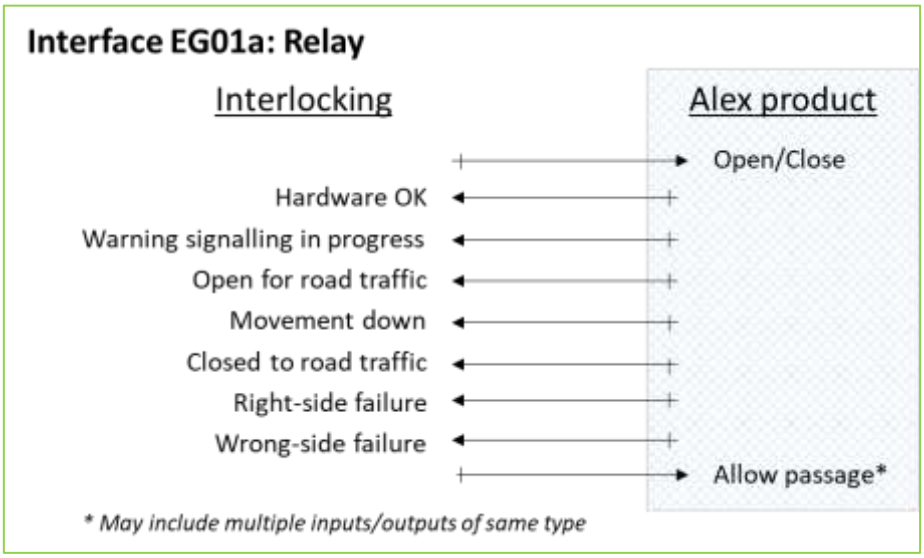



**Figure 3: Relay interface for level crossing Alex**

## 3.5 TRAIN-CENTRIC MODELLING ABSTRACTIONS

### 3.5.1 Train-Centric Abstraction and Dynamic Path Concepts

The lower ontology defines a train-centric abstraction in which a train managed by an RBC is treated as the primary dynamic entity. Rather than modelling signalling behaviour primarily in terms of fixed blocks or routes, the abstraction centers on the train and a set of associated path objects whose existence and extent may change at runtime.

For each managed train, the following dynamic path concepts are defined:

- **Train Location Path (TLP)** – represents the train location as perceived by ETCS trackside.
- **Authority Base (AB)** – defines the spatial region within which movement authority may be granted.
- **Authority Path (AP)** – defines the currently authorised movement within the Authority Base (forward or backward relative to the train's orientation).
- **SWEPT path** – represents the path confirmed as traversed by the train, derived from successive Train Position Reports (TPRs).

These path types coexist around the managed train object and represent different aspects of state, permission, and confirmed movement. Their conceptual relationship is illustrated in Figure 4.

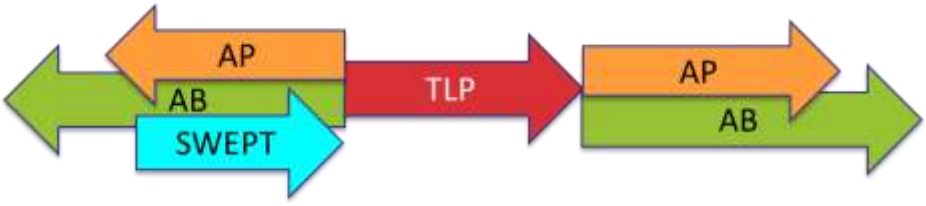



**Figure 4: Conceptual relationship between paths in the train-centric abstraction**

Within this abstraction:

- the TLP captures where the train is believed to be located,
- the AB defines the spatial envelope within which authority may be structured,
- the AP represents the currently usable portion of that authority, and
- the SWEPT path records confirmed progression of the train along the topology.

All path objects are defined relative to static trackside topology concepts introduced in the upper ontology (e.g., directed segments). This ensures that dynamic train movement is always analysed against a consistent and shared infrastructure representation.

The train-centric abstraction is designed to apply independently of whether the physical train has an operational ETCS onboard system. A managed train is represented as a dynamic object, and upon its creation a dynamic interface for it is established either:

- to an ETCS onboard subsystem, or
- to a Traffic Management System (TMS), if the train is managed indirectly.

The abstraction is inspired by concepts introduced in the moving block specification [2] (Shift2Rail TD2.3), but the demonstrator defines distinct ontology concepts and terminology tailored to the modelling approach adopted here.

An operational illustration of these concepts in use is provided later in Chapter 7 (Figure 10), where generated test cases exercise the train-centric abstraction to sweep an unknown area using dynamic path evolution.

### 3.5.2 Forbidden Paths as a Safety Modelling Pattern

The concept of **forbidden paths** was introduced in the lower ontology to address a modelling gap identified during requirements analysis of moving-block–based signalling principles in Shift2Rail. Existing abstractions were found to be insufficient to express certain classes of safety constraints—such as flank protection, handover protection, and operational movement restrictions—in a uniform and analysable way, particularly in a path-based, train-centric context.

A forbidden path represents a section of track that is explicitly forbidden for train movement, either in one direction or in both directions. **A violation of a forbidden path constraint detected at runtime is treated as a non-nominal situation.** In such cases, the modelling approach mandates creation of a corresponding **non-nominal (NN) object**, which explicitly records the violation and persists until the underlying cause has been removed or resolved.

By linking forbidden paths to non-nominal objects in this way, the modelling framework distinguishes between *the definition of safety constraints* and *the handling of their violation*. Forbidden paths define what is not allowed, while NN objects provide a structured mechanism to represent and reason about abnormal system states that arise when such constraints are violated.

This path-based formulation enables a compact and versatile representation of safety constraints that would traditionally be distributed across route logic, interlocking tables, or special-case rules. By expressing such constraints uniformly as forbidden paths, the modelling approach avoids embedding safety logic in procedural behaviour and instead captures it declaratively at the conceptual level.

Forbidden paths are used to express several distinct but related use cases, including:

- **flank protection**, preventing conflicting movements adjacent to an authorised train path;

- **handover protection**, restricting movement toward a boundary while coordination with an adjacent control area is in progress; and
- **operational restrictions**, such as temporary or permanent prohibitions imposed by traffic management.

Although inspired by interlocking principles, forbidden paths are not a standard signalling concept. Rather, they constitute a **domain-specific abstraction introduced to improve expressivity and analysability of requirements** in a moving-block–oriented modelling framework. Their path-based nature makes them well suited for integration with train-centric abstractions and for systematic validation using the object model discipline described in subsequent chapters.

### 3.5.3 Non-Nominal Objects and Exceptional Situations

In addition to modelling nominal system behaviour, the ontology framework explicitly supports representation of **non-nominal situations** through the use of *non-nominal (NN) objects*. Non-nominal situations arise when assumptions or constraints that normally govern system operation are violated, or when the system enters a degraded or exceptional mode that requires special handling.

Rather than treating such situations as transient events or implicit error conditions, the modelling approach represents them as **first-class dynamic objects**. An NN object is created when a non-nominal condition is detected at runtime and persists until the underlying cause has been resolved or explicitly cleared. This persistence enables non-nominal situations to be reasoned about consistently over time, rather than being reduced to momentary alarms or flags.

Non-nominal objects provide a uniform mechanism to capture a wide range of exceptional situations, including—but not limited to—violations of forbidden path constraints, inconsistent or conflicting input from subsystems, loss of confidence in train location information, or override actions initiated by traffic management. By representing such situations explicitly, the ontology supports modelling of system behaviour that depends not only on the current configuration and permissions, but also on the presence of unresolved abnormal conditions.

Conceptually, NN objects separate **the detection of abnormal situations** from **the response to them**. The ontology defines what constitutes a non-nominal condition and how it is represented, while the specific behavioural responses—such as restricting movement, issuing commands, or requiring operator intervention—are defined elsewhere using behavioural models that operate on the object model interface.

This explicit treatment of non-nominal situations is essential for realistic requirements validation. It enables requirements to be formulated that distinguish clearly between nominal and degraded operation, supports validation of recovery and override scenarios, and ensures that exceptional conditions are neither ignored nor handled implicitly. As such, NN objects play a key role in maintaining clarity, consistency, and analysability in modelling of safety-critical trackside systems.

### 3.5.4 Split and Join

In ETCS-based operation, train coupling (join) and decoupling (split) are associated with exceptional but expected situations. These include, for example, temporary loss of train integrity during splitting and authorised overlap of movement permissions when trains are coupled. Treating such situations as ordinary movement scenarios would blur the distinction between expected operational exceptions and genuinely hazardous behaviour.

To address this, the lower ontology introduces dedicated path types — **JOIN** and **SPLIT** — that explicitly compartmentalise these operations. These path types represent spatial regions and movement contexts within which coupling or decoupling behaviour is permitted and expected.

This compartmentalisation provides two key benefits:

- It allows requirements to distinguish clearly between expected exceptional behaviour (e.g. temporary integrity loss during a controlled split) and unexpected or unsafe behaviour requiring mitigation.
- It enables precise constraints on authority overlap. For example, an Authority Base (AB) may overlap a Train Location Path (TLP) only within a JOIN path, and only for the specific trains involved in the joining operation.

By making these operational contexts explicit in the ontology, the modelling framework supports clearer safety semantics and more systematic validation of train-centric exceptional behaviour. JOIN and SPLIT paths therefore serve not merely as modelling artefacts, but as abstraction mechanisms that preserve safety intent while maintaining analysability.

**In summary section 3.5, the train-centric abstractions, forbidden paths, and non-nominal objects together provide a compact and expressive conceptual basis for modelling both nominal operation and exceptional situations within the chosen abstraction level.**

## 3.6 Summary of the Ontology Framework

This chapter has described the ontology framework that underpins the modelling approach used in the demonstrator. The framework provides a structured way to define the conceptual system domain, separating **general, domain-independent concepts** in an upper ontology from **domain-specific specialisations** in a lower ontology.

The upper ontology establishes a stable universe of types and modelling principles, including object categories, attribute semantics, lifecycle concepts, and interaction abstractions. It fixes the chosen abstraction level explicitly and ensures that modelling assumptions are made visible and consistent across validation activities. The lower ontology specialises these general types to define a coherent set of concepts appropriate for the intended application domain, while remaining compatible with the structural constraints imposed by the upper ontology.

Together, the upper and lower ontologies provide a complete conceptual vocabulary for expressing requirements, constraints, and interaction patterns within the selected scope. The ontology framework supports reuse and extensibility based on the mechanisms of single inheritance, mixins, and specialisation, enabling domain-specific modelling without compromising clarity or analysability.

Importantly, the ontology framework itself remains **independent of behavioural logic and implementation details**. Its role is to define *what exists* in the system and *how concepts relate*, not *how behaviour is realised*. The realisation of this conceptual structure as a concrete modelling interface is addressed in the next chapter.

Chapter 4 therefore builds on this foundation and describes how the ontology is mechanically transformed into an **object model**, providing a precise and uniform interface for instantiation, behavioural modelling, and validation.

# 4 OBJECT MODEL METHODOLOGY

This chapter describes how the conceptual ontology introduced in Chapter 3 is **realised as a concrete modelling interface**. While Chapter 3 focuses on *what concepts exist* and *how they are structured at the conceptual level*, the present chapter addresses *how these concepts are made operational* through object model generation.

The object model (OM) provides a precise and uniform interface—consisting of object types and typed attributes—that is used consistently to instantiate system models, express behavioural models, and perform validation activities. The discussion in this chapter therefore concentrates on the **mechanics and discipline of object model generation and use**, rather than on the domain semantics of the ontology itself.

In this report, the object model (OM) is treated as a modelling interface. Properties and test cases are expressed using the OM interface, while verification and execution are performed on the corresponding formal or executable models generated from it.

## 4.1 ONTOLOGY DEFINITION AND OBJECT MODEL GENERATION CONTEXT

### 4.1.1 Ontology Definition Language

The ontology is defined using a **Python-based definition approach**, in which classes, mixins, specialisations, and attributes are created via generic constructor functions provided by the ontology framework. Although ontology definitions have a declarative style, they are expressed as ordinary Python statements and are therefore subject to ordering constraints (for example, parent classes and mixins must be defined before they are referenced).

The framework uses Python dataclass constructs to represent ontology elements in a structured form, enabling uniform processing, inspection, and transformation of ontology definitions. Conventions are applied to separate the upper ontology from the lower ontology and to group related parts of the ontology, for instance to support analysis and visualisation of selected subsets of the class structure.

This approach allows ontology definitions to remain explicit, programmable, and easily extensible, while serving as the input to subsequent object model generation steps described below.

### 4.1.2 Object Model Generation

Given an ontology defined as described above, object model generation is a processing step that produces the **modelling interface** for each leaf type in the ontology. Informally, the generation step “pushes down” the relevant attribute definitions from the class structure to leaf types, applying mixins and attribute specialisations and checking consistency along the way. The result is a set of leaf-type interfaces where each leaf type:

- has a fully resolved set of attributes (as specialised for that leaf type), and
- is associated with a set of sorts (class sorts, mixin sorts, and specialisation sorts) used in the formal representation.

The generated interface treats each attribute as a **global function** with the same name as the attribute. This yields a uniform “read” interface where attribute names are **globally unique**, avoiding name collisions and overloaded function semantics.

Type checking during generation applies the following integration rules in order:

1. attributes inherited from the parent class are added to the class,
2. attributes defined locally in the class are added,
3. attributes imported from mixins are added, and
4. attribute specializations are applied (requiring the attribute to already exist).

This order ensures that the attribute set of a class is well-defined and that specialization is only applied to declared attributes.

## 4.2 Generated Object Model Interfaces

This section describes the object model interfaces generated from the ontology and used throughout the demonstrator for executable and formal validation. Object model generation transforms the ontology-defined abstraction into concrete, language-specific interfaces while preserving its typing discipline, attribute semantics, and structural assumptions.

Two object model interfaces are generated in parallel: an executable interface in Python, used for instantiation, test case execution, and scenario-based validation; and a formal interface in HLL, used for specification-level reasoning and formal verification of selected properties (see Figure 5). Although realised in different languages, both interfaces expose the same conceptual object model—object types with explicitly typed attributes—and are generated mechanically from the ontology.

Illustrative excerpts from the generated interfaces are provided in Appendix B.

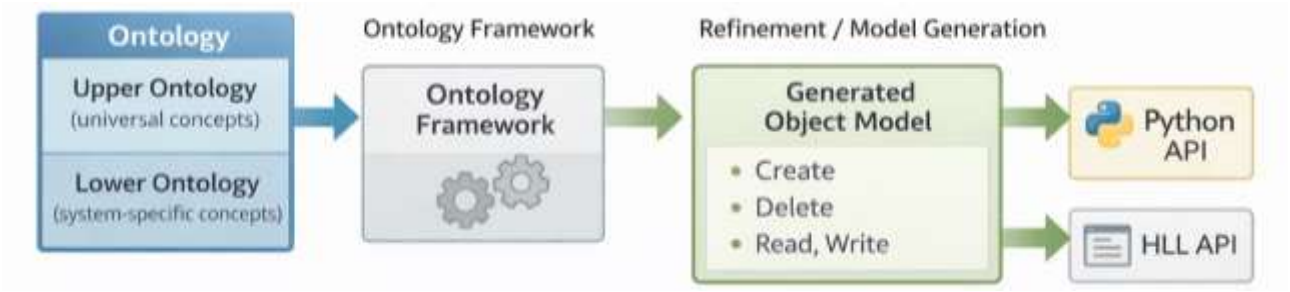



**Figure 5: Generation of object model interface in two languages: Python and HLL**

### 4.2.1 Executable Object Model Interface (Python)

The demonstrator generates a Python interface for the object model to support model instantiation and executable test-based analysis. The generated Python interface includes:

- **global functions** for reading attribute values,
- **utility functions** to query sort hierarchy information, and
- a Python class representation for each leaf type, providing an object constructor and typed attribute storage.

In the generated Python classes, attributes are represented using *__slots__* to improve memory usage and to reduce accidental creation of misspelled attribute names (which would otherwise silently create new fields). This supports robustness during prototyping and during automated instantiation from data.

In addition to generated code, the Python interface relies on a small set of **manually defined helper functions**, used in two main ways:

1. **function-defined attributes**: some attribute definitions are specialised to be functions (e.g., derived attributes); and
2. **generic upper-ontology mechanisms**: e.g., common lifecycle operations in factories, timer operations, and reusable utilities.

The net effect is that the object model interface becomes a stable "API surface" for all executable modelling: instantiation, behavioural modelling, and test-driven V&V.

### 4.2.2 Formal Object Model Interface (HLL)

The framework also generates an interface in HLL for formal verification. The goal is to provide a formal representation that is homogeneous with the instantiated Python model, while respecting the constraints and idioms of the HLL modelling language.

At a high level, the generated HLL interface includes:

- separate representations of **static** and **dynamic** attributes (e.g., separate structs),
- global functions for reading attribute values (for attributes introduced locally and via mixins),
- an explicit **sort hierarchy** for classes, mixins, and specialisations, and
- a representation of enumerated types as sorts, consistent with the type system used in HLL.

A key aspect is that the formal model preserves the ontology's typing discipline by generating explicit sort relationships, including equivalence classes where multiple sorts map to the same leaf-type set.

The HLL interface is complemented by generated HLL "instance data" describing the instantiated model (see Section 4.5), enabling formal verification of selected properties over the specific instantiation.

## 4.3 Attribute Semantics and Object Lifecycle

The modelling framework relies on explicit attribute semantics and explicit representation of object existence and lifecycle. The object model distinguishes:

- **static** objects: exist permanently and are always active (except nullary objects),
- **dynamic** objects: may be activated and deactivated at runtime, and
- **nullary** objects: one per leaf type, used in lieu of NULL values.

Attributes are explicitly typed and assigned an attribute category:

- **static** (unchanging during execution),
- **dynamic** (may change over time), or
- **incarnation** (fixed while a dynamic object is active; may differ across reincarnations).

The demonstrator enforces these semantics through runtime checks in the Python model (e.g., preventing writes to static attributes during execution) and by treating the same constraints as assumptions or proof obligations in the HLL formal model, depending on expressivity.

In the object model, object existence is represented explicitly by an *active* attribute:

- active is specialised to constant True for static non-nullary objects,
- active may change for dynamic objects, reflecting runtime activation/deactivation,

- active is always False for nullary objects.

This explicit existence modelling supports both executable V&V (runtime checking) and formal reasoning about “current system state”.

The attribute semantics and lifecycle discipline described above are complemented by a small set of higher-level consistency principles that apply uniformly across all models built using the framework.

## 4.4 Higher-Level Ontology Consistency Principles

In addition to defining types and relations, the modelling framework incorporates a small set of higher-level consistency principles that apply uniformly to all models built using the framework. These principles express fundamental assumptions about object existence, lifecycle discipline, and attribute semantics. They are independent of any particular lower ontology or behavioural model and therefore provide a common semantic foundation for validation activities.

At a conceptual level, the following principles apply:

- **Existence discipline:** For each instantiable object type, a corresponding nullary object exists and represents the absence of a concrete instance. Nullary objects are never considered to exist in the system.
- **Static versus dynamic objects:** Each object is classified as either static or dynamic. Static objects exist permanently and cannot be created or deleted at runtime. Dynamic objects may be activated and deactivated during execution.
- **Explicit object existence:** Object existence is represented explicitly. Static non-nullary objects are always active, dynamic objects may change their active state over time, and nullary objects are never active.
- **Attribute type and category correctness:** Attribute values always conform to their declared type and value range. Static attributes do not change over time, while dynamic attributes may evolve. Attributes classified as incarnation attributes remain constant for the lifetime of a dynamic object.
- **Reference consistency:** Active objects do not reference inactive objects, except where explicitly permitted by specialised modelling constructs (such as dormant attributes).

These principles are treated as **invariants of the modelling framework**. Compliance can be checked dynamically during executable instantiation and execution, where violations result in immediate errors, and statically through formal analysis of generated models, where applicable. Together, these checks ensure that instantiated object models and behavioural models adhere to the declared ontology semantics.

## 4.5 Consistency Checking and Proof Obligations

Given the attribute semantics and lifecycle discipline described above, the framework provides generic consistency checking and proof-obligation generation mechanisms to support systematic validation of ontology definitions and instantiated object models.

The framework supports generic consistency checks and the generation of proof obligations used to validate the ontology structure and certain classes of modelling constraints. These checks can be applied:

- dynamically during executable model instantiation and runtime manipulation; and
- statically by generating formal properties for model checking in HLL, where applicable.

This provides a disciplined quality-assurance mechanism for ontology-based modelling and helps ensure that generated object models and instantiated system data adhere to declared semantics.

## 4.6 Behaviour Separation and Modelling Discipline

A foundational design choice is to separate the **object model** (data model and modelling interface) from **behavioural models** (rules, algorithms, and system logic).

- The **object model** defines the vocabulary: object types and typed attributes.
- **Requirements** and **tests** are written in terms of the object model vocabulary.
- **Behavioural models** implement or simulate system behaviour by manipulating instantiated objects through the object model interface.

In other words, requirements and behaviour are expressed **in terms of** the object model; verification typically means verifying **behaviour** (executable or formal) **against requirements**, while both share the same modelling interface.

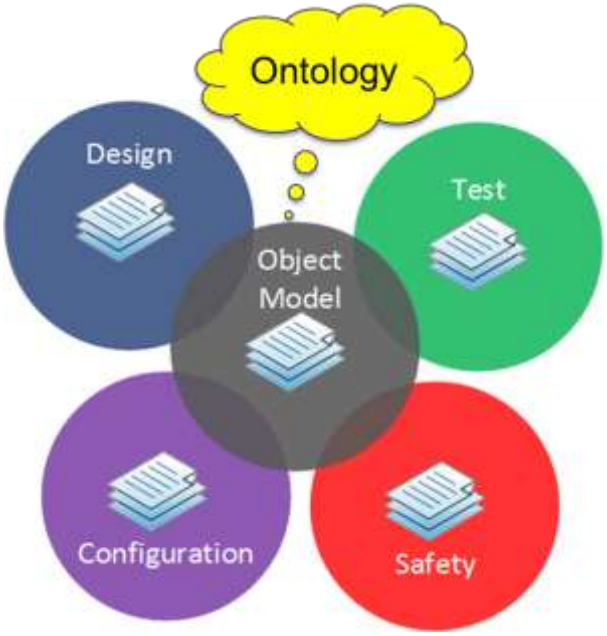



**Figure 6: Object model as shared semantic foundation for engineering artefacts**

### 4.6.1 Required Behaviour (Not Reference Design)

In many procurement and stakeholder contexts, the primary goal is not to maintain a complete reference design, but to define and validate **required behaviour** in a precise, analysable form. The demonstrator therefore focuses on modelling required behaviour in a way that supports systematic V&V and reduces reliance on manual judgement.

Once required behaviour is expressed precisely and validated early, it becomes more feasible to compare design options, supplier proposals, or reference implementations against a shared foundation.

Behavioural formalisms (such as behaviour trees) are discussed in Chapter 5; Chapter 4 focuses on the interface discipline enabling such behavioural models to be defined and analysed.

## 4.7 Message Objects and Interaction Abstraction (Conceptual)

To support system-of-systems modelling, the demonstrator adopts a message-based interaction abstraction in which information exchange between subsystems is modelled explicitly using **message objects**.

At the conceptual level:

- interaction between containers (subsystems) is represented as explicit creation, transfer, and consumption of message objects;
- message objects are allocated to factories associated with communication links;
- message types are defined partly by ontology generation and partly by explicit configuration (e.g., ETCS-standardised RBC–onboard message families and additional message types for subscriptions and events).

This message abstraction is intended to provide a disciplined, high-fidelity basis for modelling interaction, supporting both executable simulation and formal reasoning about allowed exchanges.

**Status in the demonstrator:** this interaction model is treated as a conceptual abstraction and is only partially realised in the demonstrator implementation. This report therefore distinguishes:

- the intended modelling discipline and semantics, and
- the extent to which the implementation currently realises the discipline.

## 4.8 Analysis Models for Validation

The demonstrator produces two closely related analysis models used for validation:

- an executable model in Python (supporting instantiation and test-based validation), and
- a formal model in HLL (supporting formal verification of selected properties).

These models are designed to be as homogeneous as possible, given the differences between languages. The HLL formal model is generated based on the instantiated Python model: after instantiation, a conversion step generates the “specific data” portion of the HLL model corresponding to the concrete instantiated objects and attribute values.

The two models differ in representation details (for example, how attribute storage and update semantics are encoded), but aim to preserve the same conceptual interface and typing discipline.

### 4.8.1 Sets as Characteristic Functions

A recurring representation technique in the HLL model is to encode sets as **characteristic functions**. In the simplest case, a set of objects of type T is represented by a function:

- $T \rightarrow bool$

returning True when the object is a member of the set.

In the general case used in the demonstrator, sets contain tuples with a signature of ground types (booleans, integers, enums, and object types). The characteristic function is then defined over the tuple domain, returning True for members and False otherwise.

This technique supports uniform modelling of relations and membership tests in a way that integrates naturally with formal verification tools. The trade-off is that the set contents are represented implicitly, which can be less intuitive to inspect than explicit enumeration; however, it often improves compatibility and tractability for formal reasoning.

# 5 BEHAVIOURAL MODELLING (CONCEPTUAL AND PARTIAL REALISATION)

This chapter describes the approach to behavioural modelling considered in the demonstrator. The intent is to support validation of **required behaviour** in a form that can be analysed early, and that can remain independent of detailed implementation design choices. The demonstrator therefore treats behavioural models as separate artefacts that operate on instantiated object model data through the generated object model interface (Chapter 4).

The behavioural modelling work in the demonstrator is **conceptual and partially realised**: the demonstrator defines foundational modelling mechanisms and explores candidate formalisms and execution concepts but does not yet provide complete behavioural models of all relevant subsystems or full, validated interaction semantics for all message exchanges.

## 5.1 CONCEPTUAL MESSAGE-BASED INTERACTION AND SCHEDULING MODEL

The demonstrator adopts a **conceptual message-based interaction model** to represent information exchange between subsystems in a system-of-systems context. This model is intended to support reasoning about interaction semantics, timing assumptions, and validation scenarios at an abstraction level suitable for early requirements analysis. Its purpose is to define *what* kinds of interactions are meaningful and constrained, rather than to prescribe a concrete implementation.

Interaction is modelled as **explicit message exchange between containers**. Messages are treated as **dynamic objects** allocated to link-specific factories. Conceptually, sending a message corresponds to activating a message object on an outgoing link; receiving a message corresponds to deactivating that object on the incoming link. Message types are known statically, while individual message instances are created and consumed dynamically. This object-based treatment makes message existence, lifecycle, and ordering explicit and analysable.

Container execution is defined relative to a **global logical time model**. An execution of a container is an individual processing interval with a start time $T_0$ and an end time $T_1$, where the execution duration is $T_1 - T_0$. When a container begins execution at time $T_0$, it processes all incoming messages whose arrival time is strictly less than $T_0$.

During an execution, a container may emit messages to its environment. The actual transmission of those messages onto communication links, and their eventual arrival at receivers, is determined by the surrounding execution environment and a (potentially non-deterministic) **scheduler**. The model therefore does not assume that message arrival times are fixed at send time or as part of the container's execution semantics. However, for any given communication link from container $C_1$to container $C_2$, arrival times are required to be monotonic with respect to message creation order at $C_1$: if message $m_1$is created before $m_2$and both are eventually transmitted on the same link, then the arrival time of $m_1$is less than or equal to that of $m_2$. This constraint is a deliberate modelling decision. It abstracts from lower-level protocol reordering and enforces per-link causal consistency, ensuring that the logical time model does not introduce inversions where a later-created message overtakes an earlier one along the same channel (messages on different links remain temporally independent). This model captures causal ordering without assuming synchronous execution or instantaneous communication.

To support flexible communication patterns, the upper ontology introduces **half-link abstractions**, which decouple senders and receivers. Half links can be paired dynamically to realise one-to-one, one-to-many, or many-to-one communication, including radio-based interaction between trains and

trackside subsystems. This abstraction allows interaction topology to be expressed without committing to physical connectivity or fixed addressing schemes.

At the conceptual level, execution is coordinated by a **scheduler** that selects containers for execution based on pending message arrivals and execution policies (e.g. periodic or immediate execution). Each execution advances a global tick counter and may schedule future executions by producing messages or triggering time-based conditions. A formal realisation of the scheduler and message-transfer procedures is envisaged as part of future work, but was not required to validate the modelling abstractions exercised in the demonstrator.

The interaction and scheduling model described here is **only partially realised** in the demonstrator. Behavioural execution and validation activities focus primarily on a single container (the RBC), while other containers are instantiated structurally but remain behaviourally passive. Nevertheless, the conceptual model provides an important foundation for:

- expressing interaction-related requirements,
- reasoning about message ordering and causality,
- and guiding future extension toward multi-container execution and formal system-of-systems analysis.

Subsystems are modelled as containers connected either by regular links (with statically known sender and receiver) or by paired half links (e.g. RBC2TRAIN_OUT, RBC2TRAIN_IN) whose messages are transferred via an abstract medium such as radio communication.

Regular links are directed. Bidirectional communication is therefore represented by two link types (and corresponding link instances), one per direction; for example, communication between an RBC and the TMS uses both TMS2RBC and RBC2TMS. Object Controllers (OCs) are connected to an RBC via regular links in the same way.

Figure 7 illustrates the conceptual architecture, including regular links (yellow colour) and half links (pink); half links attach to a single container and use, by convention, suffix _in or _out to indicate their role (in paired half links);

Behavioural execution in the demonstrator scenarios is limited to the RBC container; other components are included to provide architectural and interaction context.

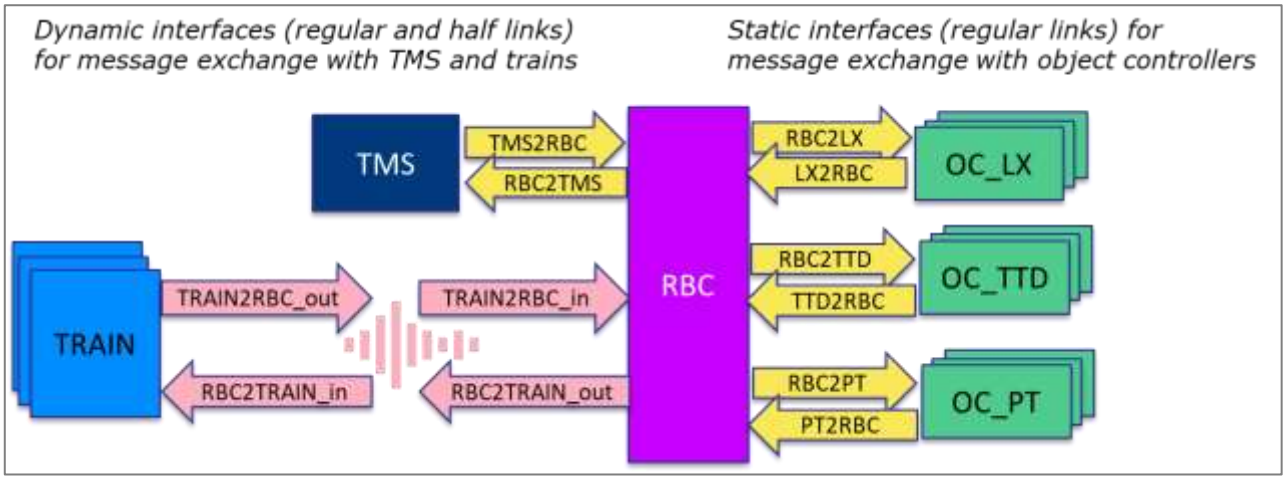



**Figure 7: Conceptual architecture and communication abstractions**

## 5.2 Behaviour Trees as a Behavioural Formalism

### 5.2.1 Conceptual Motivation

Behaviour trees (BTs) were explored as a candidate formalism for expressing required behaviour. The motivation was driven primarily by two properties:

- **Modularity.** Behaviour trees support decomposition into structured, reusable rule fragments. This is attractive in system-of-systems contexts where behaviour evolves, exceptions accumulate, and maintainability becomes a dominant quality attribute.
- **Implementation independence.** BTs are well-suited for representing "what the system shall do" without prescribing detailed implementation strategies. This aligns with stakeholder needs in procurement and requirements validation contexts, where a precise model of required behaviour is valuable even when a reference design is not a primary objective.

BTs are widely described in literature using standard node types and a "blackboard" data structure. However, available BT frameworks differ in execution semantics and data model conventions. During Task 30.2 exploration, no evaluated framework provided the level of alignment needed with the demonstrator's objectives – specifically, automated generation of a formal model suitable for formal verification.

This led to the assessment that, for the demonstrator objectives, BT exploration would require a modelling approach and supporting tooling that is compatible with:

- the ontology-derived object model interface as underlying data model, and
- automated creation of both executable and formal models.

### 5.2.2 Why the Object Model is a Suitable Data Model for Behaviour

A key idea in the demonstrator is that the object model provides a well-defined, typed interface for expressing:

- requirements and test cases, and
- behavioural models that are verified against those requirements.

Using the object model as the behavioural model's data model constrains behaviour authoring in a useful way: behavioural logic must interact with system state only through well-typed attributes and operations on instantiated objects. This supports consistency, reduces ambiguity, and improves traceability between requirements, behavioural models, and analysis results.

## 5.3 Interaction and Execution Model (Conceptual)

Behavioural models are defined per container (subsystem) and operate on objects allocated to the container's factory. Interaction between containers is modelled through explicit exchange of message objects on communication links (see Chapter 4). At the behavioural level, this establishes a primitive execution discipline in which a container's behaviour is driven by input messages and may produce output messages.

The demonstrator is designed to support **two complementary execution styles for behavioural models**, sharing the same underlying modelling semantics.

**Message-Based Interaction (Primitive Execution Model)**

In the message-based interaction model, which forms the **primitive semantic layer** of the demonstrator:

- the behavioural model processes relevant input message objects available to the container,
- updates the container's local objects as needed (via its factory), and
- creates output messages allocated to the appropriate link factories.

This interaction style aligns naturally with system-of-systems modelling, where explicit interfaces, communication constraints, and ordering of interactions are important for requirements validation and analysis. All other interaction styles are conceptually defined in relation to this primitive model.

**Vector-Based (Cycle-Based) Interaction (Derived Execution Model)**

On top of the message-based interaction layer, the demonstrator conceptually supports a cycle-based execution style as a *derived* abstraction. In this style, behavioural models operate on vectors of input and output variables, while all external interaction with the environment remains message-based.

Concretely, incoming primitive messages that arrive before a given execution time are first mapped—using a container-specific conversion function—into a local input state vector. The behavioural model then executes atomically on this vector, computing new values for internal and output variables. A corresponding conversion function subsequently maps the resulting output state vector back into a set of primitive messages, which are emitted on the container's outgoing links. Messages produced in the same execution cycle share a common creation context (e.g. tick and creation identifiers), reflecting their joint origin in a single behavioural step.

This execution style corresponds to well-known cyclic execution models used in PLC-based systems, synchronous programming languages, and digital twin or reference-design models. It is intended to support behavioural models that are naturally expressed as periodic state transformations, while preserving the explicit, message-based interaction semantics required for system-of-systems modelling and analysis.

In the demonstrator, message-based interaction constitutes the primitive execution layer. The cycle-based execution style is a conceptual extension that has not yet been fully implemented or validated but is supported by the modelling framework as a potential refinement for future work.

### 5.3.1 Conceptual Message Patterns

During the demonstrator work, several recurring *conceptual message patterns* were identified. These patterns describe the *intent* of message exchange rather than concrete message types or protocol elements and are presented here as a logical view rather than as part of the formal ontology.

The following conceptual patterns were observed:

- **Request** – a message that asks the receiver container to perform an action, without necessarily expecting a response.
- **Request–Response** – a request message for which a corresponding response message is expected within a bounded response time. The sender is assumed not to resend the request until this response window has elapsed.

- **State-Update (Notification)** – a message that reports a change in the state of an object to one or more subscribers. Subscription relationships are assumed to be configurable.
- **Non-Nominal (NN) Notification** – a message reporting detection of a non-nominal situation. In practice, this pattern is just a specialised form of state update.

These patterns apply uniformly across both static interfaces (e.g. between an RBC and wayside object controllers) and dynamic interfaces (e.g. train-centric runtime interactions). In static interfaces, requests with override semantics are used to report or enforce state changes at the attribute level, even if such changes may lead to non-nominal situations at the receiver.

During experimentation, several variants were prototyped to represent these patterns using ontology classes, mixins, and specialisations. The current position is **not** to encode these patterns as first-class ontology types. Instead, they are treated as a logical interpretation of message intent, while concrete message definitions are characterised by their payload structure.

From this perspective, a message is primarily characterised by its payload:

- **Empty payload** – no associated data.
- **Attribute payload** – a single ground-typed attribute value.
- **Structured payload** – an unordered collection of key/value pairs, where keys are enumerated values and values are ground types (boolean, enumeration, integer, or object reference).

This separation between *logical message intent* and *concrete payload representation* supports flexibility in modelling and avoids premature commitment to fixed interaction patterns, while remaining compatible with executable testing and formal validation.

## 5.4 Generic Runtime Support for Behavioural Models

Behavioural models in the demonstrator execute on top of a small set of generic runtime mechanisms provided by the object model framework. These mechanisms are independent of any specific behavioural formalism and are intended to provide a structured substrate for executable modelling and validation.

In particular, the framework provides:

- **Lifecycle management for dynamic objects**, including creation, configuration, activation, deactivation, and deletion, enforced uniformly through factory-based object pools;
- **Typed attribute access and update**, with runtime checks in the executable model to enforce attribute categories, type constraints, and reference consistency; and
- **Explicit message creation and allocation**, supporting message-based interaction between containers via link factories.

These mechanisms ensure that behavioural models operate within the semantic constraints defined by the ontology and object model, and that violations of modelling assumptions are detected early during execution. The framework deliberately avoids embedding behaviour-specific control logic, leaving behavioural semantics to be defined by the chosen modelling approach (e.g. rule-based logic, behaviour trees, or other formalisms discussed later).

## 5.5 Task-Structured Behavioural Responsibilities (Exploratory)

The behavioural modelling work in the demonstrator has explored a task-structured view of subsystem behaviour, centred around three high-level responsibilities:

- **Initialize**
- **Manage**
- **Shutdown**

These tasks are not intended as implementation phases, but as *conceptual behavioural modes* that correspond to distinct requirement contexts and validation concerns. The pattern can be natural for an ETCS trackside subsystems such as an RBC, but can apply more broadly to other safety-critical subsystems (e.g. interlockings).

**Initialize** captures all activities required to bring a subsystem from startup into a state where normal operation can begin. This task may be trivial or complex depending on context. For example, an RBC may complete initialization immediately when restarting from a previously saved and consistent state, whereas initialization may take longer if the control area is initially unknown. In such cases, requirements may involve confirmation procedures, use of trackside train detection where available, or controlled sweeping of unknown areas at restricted speed. For an interlocking-like subsystem, initialization may involve starting in a conservative safe state (e.g. all routes locked) and gradually releasing constraints once defined conditions are met.

**Manage** represents the normal operational mode of the subsystem. Most operational requirements apply in this task, governing movement authorities, path management, interaction with other subsystems, and handling of expected and unexpected events. Importantly, the demonstrator considers *entry into* and *exit from* the Manage task as requirement-relevant transitions: there are conditions under which normal operation may begin, and conditions under which it must be left.

**Shutdown** captures controlled termination of operation. This includes nominal shutdown (e.g. planned system stop or migration) as well as non-nominal termination (e.g. crash or forced restart). While the demonstrator does not model crash recovery behaviour in detail, the distinction between nominal and non-nominal endings of Manage is considered relevant for requirements analysis.

From a behavioural modelling perspective, these tasks map naturally to **subtrees in a behaviour tree** or similar hierarchical formalism. The demonstrator does not prescribe a specific behavioural language, but this structuring supports separation of concerns, clarifying which requirements apply in which task, and enabling validation of task transitions independently of detailed operational logic.

The task-structured pattern is therefore intended as *modelling guidance* rather than a complete behavioural specification. It provides a way to organise behavioural models and requirements systematically, to reason about subsystem lifecycle responsibilities, and to support early validation of conditions for entering, operating within, and exiting normal operation.

## 5.6 Status of Implementation and Validation

Behavioural modelling in the demonstrator is **not complete**, and this is an important limitation to state clearly:

- The demonstrator does not provide detailed behavioural models for all subsystems or full message-handling semantics across all interactions.

- Behaviour trees were explored as a promising formalism, but the work has focused on defining foundations and feasibility rather than delivering a complete BT-based subsystem model library.
- Validation performed to date is focused on feasibility, soundness of modelling mechanisms, and limited proof-of-concept analyses, rather than comprehensive behavioural verification of a complete system model.

This status reflects the prototype nature of the demonstrator: the work prioritised establishing a viable modelling interface, lifecycle semantics, and the ability to instantiate from real data and generate analysis models, before expanding behavioural model completeness.

# 6 OBJECT MODEL INSTANTIATION

This chapter describes how the ontology-based modelling framework and object model methodology are instantiated to produce a concrete, analysable system model. Instantiation combines infrastructure-derived data, domain-specific modelling concepts, and architectural configuration to form a unified object model suitable for executable and formal validation.

The chapter covers instantiation of global and domain objects originating from configuration data, normalisation of static extents, and instantiation of architectural elements that organise objects into a system-of-systems structure. It also outlines how dynamic objects are instantiated and managed at runtime, and how consistency checks are applied during the process.

The focus is on the modelling and instantiation mechanisms exercised in the demonstrator implementation, and on how these mechanisms support validation activities even when behavioural modelling and interaction semantics are only partially realised. The purpose is not to define a complete data-processing or deployment pipeline, but to document how a well-formed, validation-ready object model is constructed under explicit assumptions and abstraction choices.

## 6.1 INFRASTRUCTURE DATA AS INPUT: RDF AND LCF

The demonstrator uses real infrastructure data examples provided by Trafikverket as input. The data represents static configuration and infrastructure information for selected railway areas and is expressed in RDF[2] (Resource Description Framework).

RDF is a W3C-standardised data model and a foundational technology of the Semantic Web. It provides a format for representing information with formal, logic-based semantics, ensuring that meaning is precisely defined and machine-interpretable. RDF expresses knowledge as simple subject–predicate–object statements (triples), forming a directed graph of relationships between resources. Because of its formal semantics, RDF supports consistent data integration, validation, and automated reasoning across systems. It is well suited for describing both generic ontology concepts (such as classes and properties and their relationships) and concrete instance data that instantiate those concepts, making it a flexible foundation for semantic interoperability and knowledge representation.

To support systematic instantiation and analysis, the RDF data is translated into a **graph-based representation** in the Layout Configuration Format (LCF [14]). LCF provides a formal, well-defined representation of track topology and related static entities, and serves as a stable intermediate format between raw infrastructure data and the ontology-based object model.

The use of RDF and LCF allows the demonstrator to:

- decouple infrastructure data representation from the ontology definition,
- reuse existing, formally defined topology semantics,
- and maintain a clear separation between generic topology modelling and domain-specific ontology concepts.

[2] https://www.w3.org/RDF/.

## 6.2 Instantiation of Global and Trackside Objects

Following translation into LCF, the infrastructure data is used to instantiate the object model. This process results in the creation of:

- **Global Objects (GO)** representing track topology primitives, such as nodes, edges, and directed segments; and
- **domain-specific trackside objects** (Design Concepts), defined in the ontology and associated with the topology (e.g. node-resident objects and area definitions).

Architecture Concepts (e.g., containers, links, and message objects) are instantiated separately (see Section 6.3).

Instantiation is performed using generic functionality generated from the ontology. Object creation uniformly follows the object model's typing and lifecycle rules, ensuring that all instantiated objects—both global and domain-specific—conform to declared attribute semantics and generic consistency requirements by construction.

For efficiency and reuse, statically defined areas and paths are normalised by introducing shared extent-definition objects. This normalisation reduces duplication of extent information across objects and supports more efficient analysis, by limiting the size and redundancy of the model. For example, if two areas cover the same physical track section, both reference a common extent-definition object rather than duplicating the extent information. Static areas are represented as sets of paths whose combined extent corresponds to the area. Shared path definitions can therefore be reused across multiple areas, reducing redundancy in the model.

Table 2 provides an overview of anonymised RDF-based infrastructure data sets provided by Trafikverket, in terms of the number of directed segments (DS), edges (EDGE), and node objects of different degrees (N*). The data sets include both conventional signalling system areas as well as one area equipped with ETCS (Level 2).

| | DS | EDGE | N1 | N2 | N3 | N4_DS | N4_NS | N4_SS | Total |
|---|---|---|---|---|---|---|---|---|---|
| AS | 64 | 330 | 7 | 298 | 19 | | | | 718 |
| BB | 978 | 4029 | 140 | 3540 | 278 | 1 | | | 8966 |
| BS1 | 202 | 466 | 30 | 365 | 48 | 4 | 2 | 1 | 1118 |
| BS2 | 292 | 1278 | 45 | 1132 | 73 | 4 | 2 | 1 | 2827 |
| FA | 84 | 228 | 16 | 186 | 16 | 3 | 1 | 1 | 535 |

**Table 2 Global object counts in infrastructure data sets used for validation**

## 6.3 Instantiation Process

The process to instantiate the object model proceeds in stages. First, objects that originate from infrastructure data are instantiated, including Global Objects (GO) and Design Concepts (DC). Next, based on architectural configuration, Architecture Concepts (AC)—such as containers, factories, and links—are instantiated. Finally, additional dynamic objects are created as needed (for example, to represent trains), and objects are allocated to containers (subsystems) and communication links defined by the architecture.

A simplified illustration of this process is given in Figure 8: Infrastructure-derived objects (topology and domain concepts) are instantiated first, followed by architectural elements that organise these objects into a system-of-systems structure. Architectural groupings provide a stable frame of reference for allocation and interaction, while allowing dynamic creation and removal of runtime entities. Behavioural models are associated with instantiated containers in later stages.

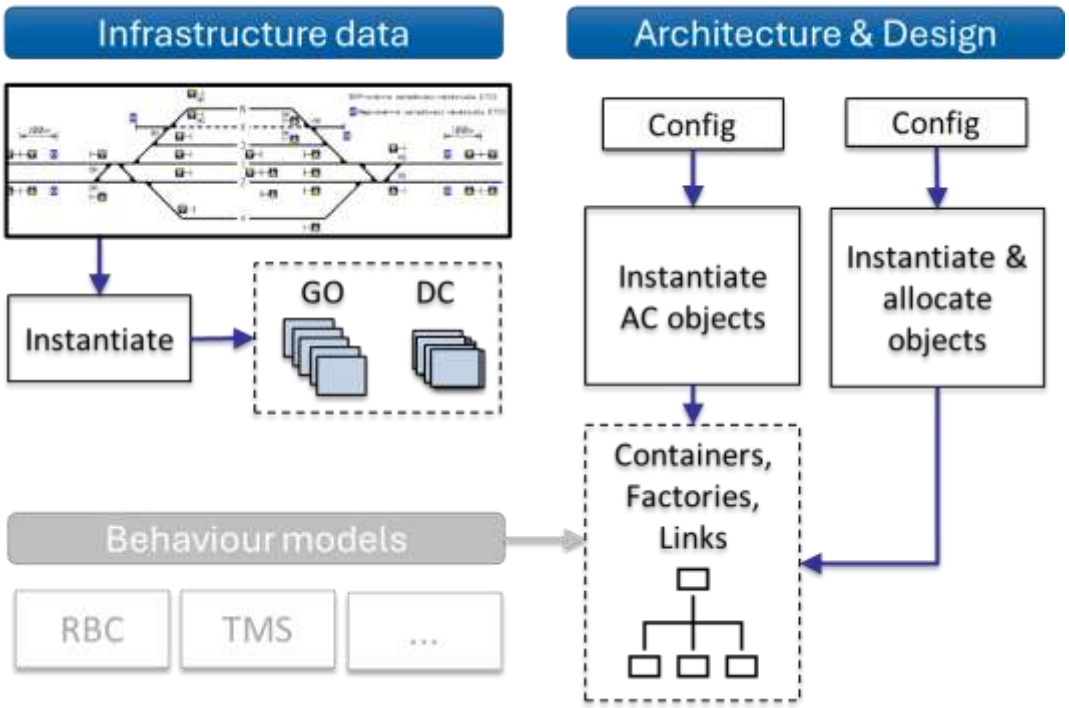



**Figure 8: Main steps in pipeline to instantiate the object model**

The process to instantiate the object model includes the following sequence of stages:

1. **Translation**

   RDF-based infrastructure data is translated into an LCF representation, providing a graph-based model of the railyard topology and associated static configuration elements.

2. **Instantiation of Global Objects**

   Global Objects (GO) representing topology primitives—nodes, edges, and directed segments—are instantiated (based on the LCF railyard graph).

3. **Instantiation of Design Concepts**

   Domain-specific static objects defined in the ontology (Design Concepts) are instantiated and associated with the topology. This includes node-resident objects, static areas, static paths, and other trackside concepts required for requirements modelling.

4. **Extent Normalisation**

   Static areas and paths are normalised by introducing shared extent-definition objects. This reduces duplication of extent information (to support reuse and improve efficiency).

5. **Instantiation of Architecture Concepts**

   Architecture Concepts (AC)—such as containers, factories, communication links, and message types—are instantiated separately based on architectural configuration. When the

AC objects have been instantiated, this step allocates the previously instantiated DC and GO objects over the system-of-systems structure (e.g., the TMS, RBCs, and object controllers), without modifying the underlying infrastructure data.

6. **Consistency checking and preparation for validation**

   Generic consistency and type checks are applied throughout instantiation and attribute writing, both during topology/domain instantiation and during architecture instantiation. These checks detect modelling errors early and ensure that the resulting object model conforms to declared ontology semantics.

   As a final step before execution-based validation, each factory is explicitly finalised. Finalisation marks the transition from model construction to execution: after a factory has been finalised, no new objects may be added to it (its structural contents are fixed). This enforces a clear separation between instantiation-time configuration and run-time behaviour and ensures that executable validation operates on a stable and well-formed object model.

The result of this process is a fully instantiated object model that combines infrastructure data, domain concepts, and architectural structure. This instantiated model serves as the common basis for executable validation (Python-based testing and scenario execution) and to generate the instantiated model for formal verification (HLL-based analysis) described in Chapter 7.

Appendix C illustrates how instantiated object data is represented in the generated formal model.

### 6.3.1 Instantiation of Architecture Concepts

The demonstrator instantiates Architecture Concepts (AC) based on explicit configuration, separated from the RDF-based infrastructure data. This reflects their role as modelling constructs that define *how* domain objects are organised and interact in a system-of-systems context, rather than *what* infrastructure exists. As a result, the same infrastructure instantiation can be combined with different architectural configurations, enabling exploration of alternative system-of-systems structures without modifying underlying data.

Architectural objects are further organised using two upper-ontology concepts: **U_Globals** and **U_Part**. *U_Globals* represents a singleton grouping for global objects and static domain elements that exist for the lifetime of a system-of-systems instance. *U_Part* represents a static architectural grouping used to organise dynamic architectural elements such as containers, factories, and communication links. While *U_Globals* and *U_Part* are themselves static, the architectural objects allocated to parts may be created, removed, or reconfigured during execution.

This separation provides a stable architectural frame of reference while allowing controlled evolution of system structure, and supports structured reasoning about lifecycle, ownership, and allocation of architectural elements in dynamic system-of-systems scenarios.

(Technically, AC objects may also be instantiated from explicit configuration data, but they are treated separately from infrastructure data in the demonstrator.)

## 6.4 Dynamic Objects: Instantiation and Runtime Lifecycle

In addition to static objects, the modelling framework supports **dynamic object types**, whose instances may appear and disappear over time. These objects represent runtime entities such as trains, movement authorities, swept paths, and markers for non-nominal situations.

It is important to distinguish between two phases in which dynamic objects are handled:

**Dynamic object instantiation (configuration-time)**

Before execution begins, the object model may be instantiated with an initial population of dynamic objects based on configuration decisions. This includes, for example:

- pre-allocated train objects,
- dynamic path and area objects that are expected to be used during execution, and
- architectural or bookkeeping objects required by the system-of-systems structure.

At this stage, objects are instantiated as **inactive**, allocated to factories, and prepared for later use. Their existence at this point reflects modelling assumptions and configuration choices, not runtime behaviour. This phase is part of the overall instantiation pipeline (steps 3 and 5) described in Section 6.3.

**Dynamic object creation and deletion during execution**

During execution, behavioural models may create and delete dynamic objects to represent evolving system state. Examples include:

- creation of train-centric objects (e.g. TLP, AB, AP) as trains are introduced and managed,
- extension, replacement, or removal of authorities and swept paths as trains move, and
- creation and removal of objects representing non-nominal situations.

Runtime creation and deletion of dynamic objects is governed by a **disciplined lifecycle model**:

- objects are instantiated in an inactive state and allocated to a factory,
- attributes are configured while the object is inactive, in accordance with declared typing and category semantics,
- objects transition to the active state only when all relevant consistency checks succeed, and
- objects may later be deactivated and eventually deleted, subject to lifecycle and reference constraints.

This explicit lifecycle modelling is a deliberate design choice. It allows temporary, incomplete states during configuration while ensuring that declared invariants hold whenever objects are active (i.e. considered to exist). The lifecycle discipline is enforced uniformly by generic functionality in the object model interface, providing immediate feedback during executable validation and a clear semantic basis for formal reasoning about object existence and state transitions.

The demonstrator includes generic mechanisms for managing dynamic object pools within factories, enforcing activation and deactivation rules, and detecting invalid references (see Section 8.1.6). While dynamic objects play a central role in scenario-based validation, the demonstrator does not attempt to provide complete operational behaviour models for all dynamic entities. Instead, dynamic object handling is used to exercise and validate the modelling abstractions, lifecycle semantics, and consistency constraints that are central to early requirements validation.

## 6.5 Visualisation of Instantiated Models

Visualisation support in the demonstrator operates on **instantiated object model state** and is used to inspect, explore, and gain confidence in the correctness of instantiated infrastructure data and dynamic modelling abstractions. While not part of the instantiation process itself, visualisation

depends directly on an instantiated system-of-systems model and is therefore described in this chapter.

The demonstrator provides visualisation of:

- instantiated track topology and node-resident objects,
- areas and paths with static or dynamic extent, and
- selected train-centric and movement-related modelling constructs.

Visualisation is used primarily for **exploratory inspection and early validation**. Typical uses include:

- checking that infrastructure data has been instantiated as intended,
- inspecting the spatial interpretation of areas and paths, and
- visually assessing the effects of dynamic operations such as path creation, extension, and sweeping.

By presenting instantiated objects in a graphical topology-aware view, visualisation makes modelling assumptions and abstraction choices tangible. This often enables rapid detection of modelling or configuration issues that are difficult to identify through textual traces or formal properties alone.

The visualisation mechanisms are deliberately decoupled from behavioural execution and verification logic. They complement, rather than replace, executable test-based validation and formal verification activities described in Chapter 7. In this sense, visualisation acts as a **diagnostic and understanding aid**, supporting early feedback during model instantiation and scenario-based exploration.

Examples of visualised infrastructure used in the demonstrator include selected sections of the Swedish railway network, instantiated from RDF-based data and rendered graphically. Figure 9 illustrates a small part of the topology of the largest dataset example provided by Trafikverket, rendered in a graphical viewer for LCF. The different colours indicate different types of track areas.

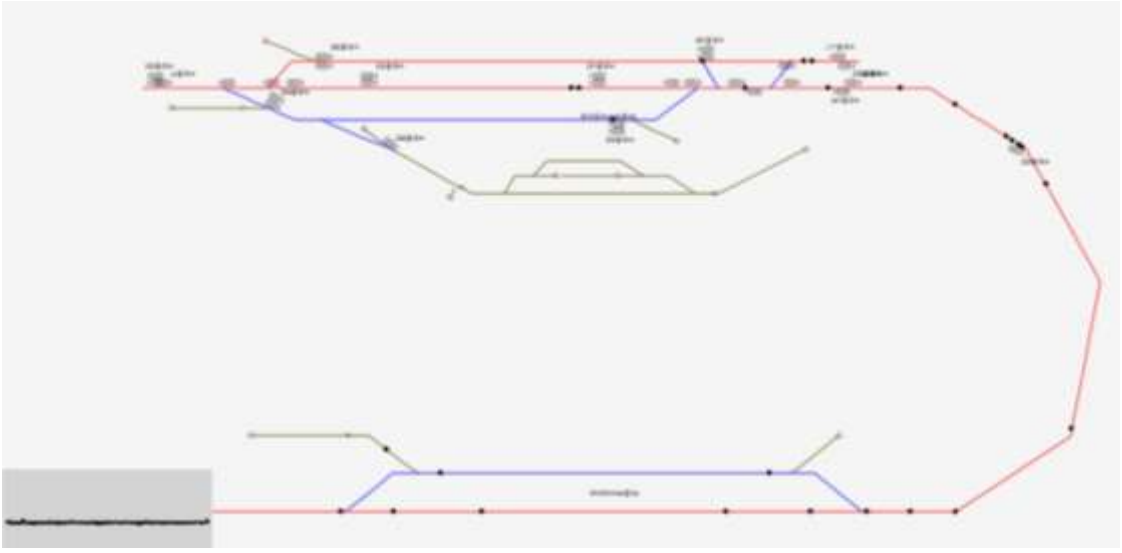


**Figure 9: Visualisation of infrastructure data**

## 6.6 Scope and Limitations of Instantiation

The instantiation performed in the demonstrator is deliberately scoped and reflects its role as a prototype intended to explore modelling abstractions and validation mechanisms rather than to deliver a complete data integration pipeline.

In particular:

- only a limited, but representative, set of real infrastructure data examples was instantiated;
- behavioural models operating on the instantiated data are partial and exploratory, serving to exercise modelling abstractions rather than to represent full subsystem logic; and
- message-based interaction is specified conceptually, with only partial realisation in the demonstrator implementation.

These scope choices were made consciously to keep the focus on abstraction design, object model instantiation, and early validation capabilities, rather than on completeness or operational coverage.

Within this scope, the instantiation activities nevertheless demonstrate that the proposed modelling approach:

- scales to realistic infrastructure data sizes,
- supports automated generation of both executable and formal representations from a common object model, and
- provides a practical and coherent basis for early, model-based requirements validation.

The instantiation mechanisms described in this chapter should therefore be understood as **proof-of-concept demonstrations of a modelling and validation approach**, rather than as production-ready data processing or system configuration solutions. They establish feasibility and soundness at the abstraction level targeted by the demonstrator.

## 6.7 Relation to Validation Activities

The instantiated object model provides the concrete basis for the validation activities described in Chapter 7. Static configuration and topology data are used to validate consistency and correctness properties, while dynamic object support enables limited exploration of runtime behaviour and non-nominal scenarios.

The separation between instantiation and behavioural modelling ensures that a significant portion of validation can be performed even in the absence of complete behavioural specifications.

The following chapter builds on this instantiated object model and describes how executable testing, formal verification, and scenario-based validation were applied within the defined scope to assess feasibility, consistency, and expressivity of the proposed modelling approach.

# 7 VALIDATION ACTIVITIES

This chapter describes the validation and verification (V&V) activities performed using the demonstrator. The emphasis is on what was validated, how validation was conducted, and the scope within which conclusions can be drawn. The activities reflect the prototype nature of the demonstrator and are intended to assess the feasibility, internal consistency, and analysability of the proposed abstraction and modelling methodology. The objective is not exhaustive behavioural verification or performance benchmarking, but evaluation of whether the ontology-based object-model approach provides a viable foundation for early, model-based requirements validation.

**Validation Summary**

Validation in the demonstrator, as presented in D30.3, combined three complementary mechanisms:

1. **Executable validation** through Python-based instantiation and runtime consistency checks.
2. **Formal verification** of selected consistency, typing, and lifecycle properties using HLL.
3. **Scenario-based validation** through automatically generated test cases executed over real infrastructure data provided by Trafikverket.

Scenario-based execution exercised central train-centric abstractions—including dynamic paths, authorities, forbidden paths, sweeping of unknown areas, and JOIN/SPLIT contexts—from the perspective of an ETCS trackside system (RBC).

The validation scope was intentionally limited: complete behavioural models, full interaction protocols, and operational performance claims were out of scope. Nevertheless, the activities demonstrate that the chosen abstraction supports meaningful automated validation prior to detailed design, while remaining tractable on realistic infrastructure data sets.

## 7.1 EXECUTABLE TEST-BASED VALIDATION

Executable validation is performed using the Python-based instantiation of the object model. This mode of validation focuses on:

- consistency and correctness of instantiated configuration and infrastructure data,
- enforcement of attribute semantics and lifecycle rules,
- detection of modelling errors during object creation, modification, and deletion, and
- exploration of dynamic behaviour through controlled, executable scenarios.

The generated Python object-model interface enforces typing rules, attribute categories, and lifecycle constraints at run time. For example:

- attempts to modify static attributes during execution result in immediate errors,
- references from active objects to inactive objects are rejected, and
- activation of dynamic objects is permitted only when declared type and consistency checks succeed.

These mechanisms provide immediate feedback during instantiation and scenario execution, allowing modelling errors to be detected before more complex behavioural assumptions are introduced.

Executable checks were applied both during initial instantiation of infrastructure data and during execution of automatically generated test cases. The purpose was to confirm that the abstraction behaves consistently under realistic data and dynamic evolution, and that its generic enforcement mechanisms operate predictably across data sets.

## 7.2 Formal Verification Activities

Formal verification is performed on the HLL representation generated from the instantiated object model. The formal model includes:

- the ontology-derived sort hierarchy,
- explicit representations of static and dynamic attributes, and
- the formal encoding of instantiated object data.

Verification focused on selected classes of properties, including:

- generic consistency principles defined in the upper ontology (e.g. nullary semantics, static/dynamic classification, object activity),
- type-correctness obligations arising from attribute specialisation and restriction, and
- selected domain-specific consistency properties related to static configuration data.

Some properties were formulated directly as HLL assertions. Others were generated automatically by the ontology framework based on declared attribute categories and specialisation rules. These generated proof obligations were then discharged using the model checker. Representative examples are provided in Appendix D.

A set of configuration-level consistency properties was defined and validated early in the work. Once established, these properties were retained and re-executed systematically throughout subsequent experiments. Because such properties are computationally inexpensive yet highly effective at detecting inconsistencies introduced through data changes or modelling refinements, they provided a stable correctness baseline for all further validation activities.

Together, these activities demonstrate that the ontology-driven abstraction yields analysable formal models suitable for early-phase validation, and that selected classes of requirements can be verified automatically prior to detailed behavioural design.

### 7.2.1 Use of Reachability Properties and Counterexamples

In addition to invariant-style consistency properties, bounded reachability analysis was used selectively as a diagnostic and exploratory technique. These analyses assessed whether specific train-centric states—such as combinations of authorities, path overlaps, or lifecycle transitions—were reachable within a bounded number of steps.

Counterexamples generated by the model checker were inspected using the textual debugger and used as witnesses to understand how particular states could arise. This proved valuable both for identifying modelling errors and for refining property formulations.

No graphical visualisation of formal counterexamples was implemented within this work. However, given the existing infrastructure for visualising executable traces, extending it to support visualisation of counterexample traces appears feasible and is identified as a potential future enhancement.

Reachability analysis also informed performance-related decisions. In particular, it exposed state-space growth caused by certain modelling encodings (e.g. alternative path-extent representations), directly influencing abstraction refinements discussed in Chapter 8.

**The formal verification activities demonstrate that the ontology-driven abstraction yields analysable formal models and that selected classes of requirements can be verified automatically at specification level, prior to detailed behavioural design.**

## 7.3 Scenario-Based Validation Using Generated Test Cases

In addition to static consistency checks and formal verification, the demonstrator supports **scenario-based executable validation** driven by automatically generated test cases. This validation mode exercises the modelling abstractions and lifecycle rules under realistic dynamic evolution, even in the absence of complete behavioural models for all subsystems.

### 7.3.1 Scope and Perspective

Scenario-based validation is conducted from the perspective of the ETCS trackside system (RBC). Although the demonstrator defines a broader system-of-systems structure using Architecture Concepts (containers, links, and message objects), executable behaviour in the scenarios described here is confined to the RBC container. Other architectural components are instantiated structurally but do not contribute active behaviour.

This scoping is deliberate. The purpose of these scenarios is to validate train-centric abstractions, path-based constraints, and lifecycle semantics as enforced by the RBC, independent of detailed implementations of surrounding subsystems.

### 7.3.2 Test Case Generation and Execution

Test case generation is driven by the instantiated object model using real infrastructure data imported from Trafikverket. Generic test-case templates are instantiated and executed to drive controlled state evolution, including:

- creation and deletion of dynamic objects,
- modification of path extents and authorities,
- enforcement of declared consistency and lifecycle constraints.

Execution traces are produced by the Python-based model and represent structured, movement-oriented scenarios rather than full operational behaviour. The intent is not to simulate a complete signalling system, but to exercise the abstraction under controlled yet realistic dynamic conditions.

### 7.3.3 Progressive Validation Objectives

Validation was carried out using a sequence of increasingly expressive test-case objectives, applied systematically across all imported infrastructure data examples. The progression included, e.g.:

- creation of basic train-centric objects (CUT, TLP, AB, AP),
- generation of forward and backward movement authorities and controlled train movement along main tracks,
- extension and renewal of authorities as trains approach the limits of their granted paths,
- handling of train reversal scenarios with simultaneous forward and backward authorities,

- creation and reduction of unknown areas using SWEPT paths derived from successive train-location updates,
- validation of release of flank protection as trains move through the topology, and
- validation of scenarios using JOIN and SPLIT paths for train coupling and decoupling.

These objectives were designed to exercise non-trivial interactions between dynamic paths, lifecycle rules, and protection constraints, rather than isolated attribute updates.

These scenarios exercise central modelling abstractions such as dynamic path extent, forbidden paths, swept paths, and lifecycle-dependent consistency constraints. They were applied uniformly to all infrastructure data sets used in the demonstrator.

This progression allowed modelling assumptions to be validated incrementally, with later scenarios building directly on confidence established in earlier, simpler cases.

### 7.3.4 Role of Visualisation

Execution traces produced by the generated test cases were visualised using an LCF-based topology viewer capable of displaying arbitrary path extents in terms of directed segments. Playback at representative speeds enabled intuitive inspection of train-centric behaviour and movement constraints under realistic movement scenarios.

Visual validation proved particularly effective for detecting modelling issues related to path extent, overlaps, and interactions between concurrent train-centric abstractions. An example trace was presented in deliverable D30.3. Figure 10 illustrates a representative scenario-based validation trace, showing incremental sweeping of an Unknown area using the train-centric abstractions defined in the lower ontology.

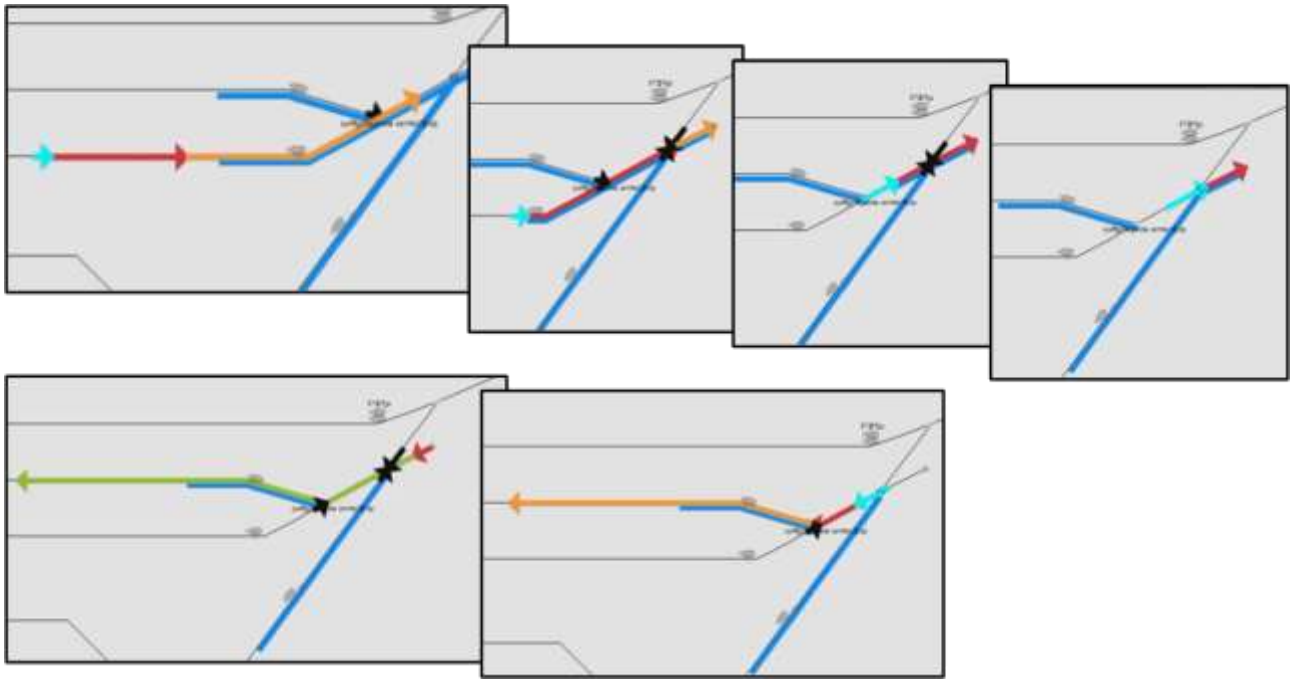


**Figure 10: Incremental sweeping of an Unknown area using train-centric abstractions**

Blue: Unknown area.
Red: Train Location Path (TLP).

Aqua: SWEPT path (confirmed train movement between successive TLP updates).
Orange: Movement Authority (AP).
Green: Authority Base (AB).
Black: Forbidden path.

Overall, scenario-based validation was used to assess **feasibility, expressivity, and internal coherence** of the modelling framework, rather than to verify complete operational correctness.

## 7.4 Scalability and Performance Observations

To provide concrete evidence regarding scalability, representative run-time measurements were collected for all Trafikverket infrastructure data sets described in Chapter 6 (see Table 1). The same data sets were used uniformly across instantiation, scenario-based validation, and formal verification activities. Table 2 summarises measurements obtained on a standard laptop platform (Lenovo, >3 years old). The reported figures correspond to four groups of activities, labelled A-D in the table:

**A – Model preparation and scenario generation (sum of subtasks):**

This group corresponds to these three subtasks (column "Total (A)" gives their total time):

- **Instantiation**: creation of Global Objects (GO), Design Concepts (DC), and Architecture Concepts (AC), including extent normalisation.
- **Simulation**: execution of a generic test case involving (i) sweeping of a shunting area using train-centric movement, and (ii) execution of a JOIN scenario.
- **Formal model generation**: export of the instantiated model to HLL.

**B – Configuration verification:**

Verification of invariant-style consistency properties over the instantiated model.

**C – Reachability analysis:**

Generation of a bounded counterexample for a representative reachability property involving train-centric state.

**D – Real-time simulation duration (50 km/h):**

Playback duration of the generated execution trace at simulated train speed. This duration is primarily determined by topology length and chosen speed and does not reflect computational overhead of instantiation or verification.

| Data set | DS | AC | Instantiate | Simulate | Formal model gen. | Total (A) | Config. verific. (B) | Reachability analysis (C) | Real-time simulation (50 km/h) (D) |
|---|---|---|---|---|---|---|---|---|---|
| **AS** | 64 | 238 | 2.5 s | 3.4 s | 0.4 s | **7 s** | 7 s | 12 s | 29m42s |
| **BB** | 978 | 2380 | 2m0s | 48.5 s | 3.9 s | **3m16s** | 8m43s | 8m55s | 211m34s |
| **BS1** | 202 | 302 | 3.8 s | 0.2 s | 0.6 s | **6.3 s** | 15 s | 24 s | 2m29s |
| **BS2** | 292 | 780 | 15.3 s | 15.4 s | 1.1 s | **35.4 s** | 41 s | 53 s | 48m4s |
| **FA** | 84 | 162 | 1.9 s | 1.2 s | 0.4 s | **4 s** | 7 s | 8 s | 8m22s |

**Table 3 Representative run-time measurements**

*DS = number of directed segments.*

*AC = number of instantiated Architecture Concept objects (containers + factories + links).*

**Interpretation**

The results show that:

- Instantiation and executable scenario generation remain within seconds to a few minutes even for the largest data set (978 directed segments).
- Formal configuration verification and bounded reachability analysis remain tractable at realistic scale.
- Real-time simulation duration is dominated by topology length and chosen train speed, not by computational overhead.

These results demonstrate that the abstraction and associated validation mechanisms remain computationally tractable across infrastructure data sets of realistic size.

## 7.5 What Was Validated

The validation activities performed using the demonstrator can be summarised as follows:

- **Configuration consistency:** validation of static infrastructure data and topology representations, including paths, areas, and normalised extents.
- **Ontology consistency:** validation of generic ontology principles, such as object classification, nullary object semantics, and attribute category constraints.
- **Type correctness:** verification that specialised attributes respect declared types, ranges, and restrictions.
- **Scalable application of analysis:** demonstration that dynamic, train-centric scenario execution and formal verification can be applied consistently across realistic infrastructure data sets.

Together, these activities provide evidence that the modelling framework and object-model methodology are internally sound and suitable for early-phase requirements validation.

These activities were applied uniformly across all imported infrastructure data sets (see Table 2), including the largest available examples.

## 7.6 What Was Not Validated and Why

Several aspects were not validated in the demonstrator, by deliberate scoping:

- **Complete behavioural correctness:** detailed behavioural models for all subsystems were not defined, and comprehensive behavioural verification was therefore out of scope.
- **Full interaction semantics:** the message-based interaction model is specified conceptually and only partially realised; full verification of interaction protocols was not attempted.
- **Operational performance:** no claims are made regarding operational performance, throughput, or real-time behaviour.
- **End-to-end system validation:** the demonstrator does not represent a complete ETCS trackside system and was not intended for operational or certification-level validation.

These omissions reflect **conscious scoping decisions**, aligned with the demonstrator's purpose and TRL, rather than limitations of the modelling approach itself.

## 7.7 Interpretation of Validation Results

The validation results should be interpreted in light of the demonstrator's abstraction level and objectives. The activities performed show that:

- the ontology-based object-model methodology can be applied at realistic infrastructure scale,
- automated validation mechanisms can be embedded early in the modelling process, and
- consistency, typing, and lifecycle properties can be analysed prior to detailed architectural or implementation design.

Executable and formal validation led directly to revisions of modelling assumptions, encoding choices, and data-handling strategies, illustrating the practical value of exercising the abstraction under realistic scenarios. For example, alternative path-extent encodings were abandoned after reachability analysis revealed unacceptable state-space growth in the formal model, and early executable validation exposed inconsistencies and quality limitations in certain imported RDF concepts, which were either refined in the ontology or reported back to the data provider. The modelling framework was therefore not only evaluated, but iteratively refined through validation.

At the same time, these results do not constitute validation of a complete signalling system or endorsement of specific signalling principles. Rather, they demonstrate that the proposed abstraction and modelling framework provide a viable and extensible foundation for structured early-phase reasoning. In this sense, the work aligns with the System Inception use case: supporting iterative clarification and strengthening of requirements before architectural and design commitments are fixed.

## 7.8 Relation to Subsequent Discussion

The validation activities described in this chapter inform the discussion of design choices, findings, and implications presented in Chapter 8. They highlight:

- the strengths of the modelling framework for early-phase validation,
- the importance of abstraction-level choices for tractability and analysability, and
- the areas where additional work is required to extend validation coverage.

# 8 DESIGN CHOICES, FINDINGS, AND IMPLICATIONS

This chapter consolidates the methodological position of the demonstrator by distinguishing between deliberate design choices, findings that emerged through practical use, and their broader implications. Section 8.1 documents intentional design principles and abstraction decisions made to support early requirements validation and verifiability. Section 8.2 presents findings that arose when these principles were exercised on realistic infrastructure data and formal tool chains. Section 8.3 synthesises these insights and discusses their implications for modelling and verification practice. Sections 8.4–8.6 then position the work with respect to the digital twin paradigm, summarise scope limitations, and outline reuse potential and future directions.

Taken together, the material in this chapter provides a grounded interpretation of the demonstrator's contribution, clarifying both what was intentionally designed and what was learned through implementation and validation.

## 8.1 DESIGN PRINCIPLES AND METHODOLOGICAL CHOICES

(Methodological contributions)

This section documents the **deliberate design principles and methodological choices** that guided the development of the demonstrator. These principles were chosen to balance modelling expressivity, verifiability, and long-term sustainability, and to support early validation of requirements in evolving railway signalling systems-of-systems. While refined during implementation, the principles described here reflect intentional decisions rather than post-hoc adjustments.

### 8.1.1 Choice of Abstraction Level: Ontology as the Conceptual Foundation

A foundational design choice was to identify the **ontology** as the primary carrier of the modelling abstraction. The ontology determines which concepts are represented, how they are related, and which distinctions are considered relevant for analysis. In doing so, it fixes the modelling scope and expressivity and makes abstraction choices explicit.

Object model generation resolves inheritance, mixins, and attribute specialisations defined in the ontology and materialises them into concrete modelling interfaces, allowing the resulting object model to be used directly without requiring detailed knowledge of the underlying ontology structure.

By placing these decisions in the ontology, the modelling approach avoids implicit assumptions being scattered across behavioural models, test cases, or verification artefacts. Changes to abstraction scope or modelling intent are made explicitly at the ontology level and propagate systematically to executable and formal artefacts through object model generation.

In this sense, the ontology defines the **conceptual level of abstraction**, while the object model provides its operational realisation.

### 8.1.2 Modelling Required Behaviour Rather Than Reference Design

The demonstrator deliberately focuses on modelling and validating **required behaviour**, rather than on defining a complete reference design or implementation model. This reflects the observation that, in many system development and procurement contexts, the primary challenge is not to design a solution, but to define clearly what the system shall do and under which assumptions.

Once required behaviour is expressed precisely and validated early, creating a design or implementation model becomes a more tractable task and can be addressed in later phases.

Conversely, developing detailed designs without first achieving clarity and consistency in requirements often fails to address the root causes of late-stage issues.

Accordingly, behavioural modelling in the demonstrator, as presented in D30.3, is used to explore feasibility, consistency, and adequacy of required behaviour, rather than to prescribe specific architectural or algorithmic solutions.

### 8.1.3 Separation of Extent Representation from Paths and Areas

Another deliberate design principle is the separation of **topological extent** from the objects that use it. Paths and areas do not directly encode their extent; instead, each path or area object refers to a separate **extent definition object**.

This separation of concerns enables uniform handling of different path and area types, reuse of generic operations such as intersection and overlap checking, and consistent modelling of both static and dynamic extents. By abstracting extent into first-class objects, the modelling framework avoids proliferation of specialised representations and supports generic validation and test-case generation mechanisms.

### 8.1.4 Path Extent Representation Using Directed Segments

A central design choice in the demonstrator concerns how the **extent of paths and areas** is represented. The chosen representation models path extent using **directed segments** and associated offsets, rather than list-based or sequence-based encodings.

In this approach, the extent of a path is defined by the following attributes (names prefixed with x_ to denote extent):

- x_first: the first directed segment of the path,
- x_last: the last directed segment of the path,
- x_inner: a set of directed segments representing the interior of the path,
- x_first_o: an integer offset within the first directed segment,
- x_last_o: an integer offset within the last directed segment.

This representation provides a **general and minimal encoding** of arbitrary path extents. It supports both traditional fixed-block paths and dynamic, moving-block-style paths within a single modelling scheme, and it enables uniform handling of extent-related operations across different path types.

The primary motivation for this design was **simplicity and tractability**. The representation is “as simple as possible” while still being expressive enough to capture the required semantics. In particular:

- it avoids deep structural recursion,
- it separates topological structure (directed segments) from extent semantics,
- and it provides a uniform basis for generic operations such as intersection, overlap checking, border extraction, and path length computation.

Several alternative representations were explored during development, including list-based encodings of segments or nodes. These alternatives proved problematic in the formal model, particularly when using HLL. Due to eager partial evaluation performed by the model checker, list-

based encodings led to expansion over all possible segment values present in the static configuration data, resulting in excessive state exploration and poor performance.

A potential mitigation—encoding lists using integer indices rather than sort instances—was considered but not pursued, as it would have required additional encoding layers and obscured the underlying domain concepts. The directed-segment-based representation was therefore chosen as a deliberate trade-off favouring **clarity, generality, and verifiability** over maximal expressivity.

Areas are handled consistently with this approach. An area's normalised extent is defined as a set of paths, each of which uses the same directed-segment-based extent representation. This allows area-related operations to reuse the same generic mechanisms as path operations, further reinforcing separation of concerns and reuse.

This experience illustrates a broader design principle applied throughout the demonstrator: modelling choices were evaluated not only for expressivity, but also for their impact on **formal analysability** and tool behaviour. The chosen path extent representation exemplifies how careful abstraction design can significantly influence the feasibility of automated verification and scenario-based validation.

### 8.1.5 Static Area Extent as a Design-for-Verifiability Trade-off

In the upper ontology, areas are defined to have **static extent only**. This is a deliberate restriction motivated by design-for-verifiability considerations. Allowing areas to have dynamic extent would introduce additional complexity in both executable and formal analysis without providing corresponding benefit for the intended modelling and validation use cases.

At the same time, the modelling framework allows more complex or dynamic spatial constraints to be represented indirectly. For example, sets of paths—such as protection paths associated with train movements—can be used to model dynamic regions of interest without requiring areas themselves to have dynamic extent.

This choice illustrates a recurring design pattern in the demonstrator: complexity is not eliminated but **relocated into modelling constructs that are easier to analyse and reason about**.

### 8.1.6 Lifecycle Discipline for Dynamic Objects

Dynamic objects in the modelling framework are managed using an explicit lifecycle pattern based on factory-managed object pools and staged activation. Objects are first created in an inactive state and configured before being made active. Activation and deactivation are controlled operations subject to generic consistency checks.

The lifecycle is enforced operationally through factory-managed pools (illustrated in Figure 11). Dynamic objects move through configuration and activation stages before becoming active, and through corresponding staged deactivation before returning to the inactive state. Static objects reside in a separate pool and are not subject to lifecycle transitions.

A key aspect of this design is that behavioural models cannot bypass lifecycle rules. If a behavioural model attempts to activate an object whose configuration is incomplete or inconsistent, the generic activation step rejects the transition. In executable runs, this results in an immediate error; in formal analysis, it manifests as a violated proof obligation or counterexample. In this way, invalid state transitions are not silently accepted but are made explicitly visible during validation.

Reference counting and staged activation together ensure that only consistency-preserving transitions to and from the active state are accepted. Objects that are still referenced by active

objects cannot be deactivated, and activation is only permitted once all declared constraints are satisfied. This guarantees that declared invariants hold whenever objects are in the active state.

The lifecycle mechanism was introduced deliberately as a reusable design pattern to support invariant-based reasoning in dynamic system-of-systems models. It separates temporary configuration states from semantically valid system states, enabling both executable validation and formal reasoning to operate over well-defined object existence semantics. The lifecycle discipline is therefore not an implementation detail, but a structural element of the modelling approach.

Because these pools are themselves ordinary, typed set-valued attributes of the factory, lifecycle state is fully represented within the object model and is therefore visible to both executable validation and formal analysis. This makes lifecycle semantics part of the modelling abstraction rather than an external implementation detail.

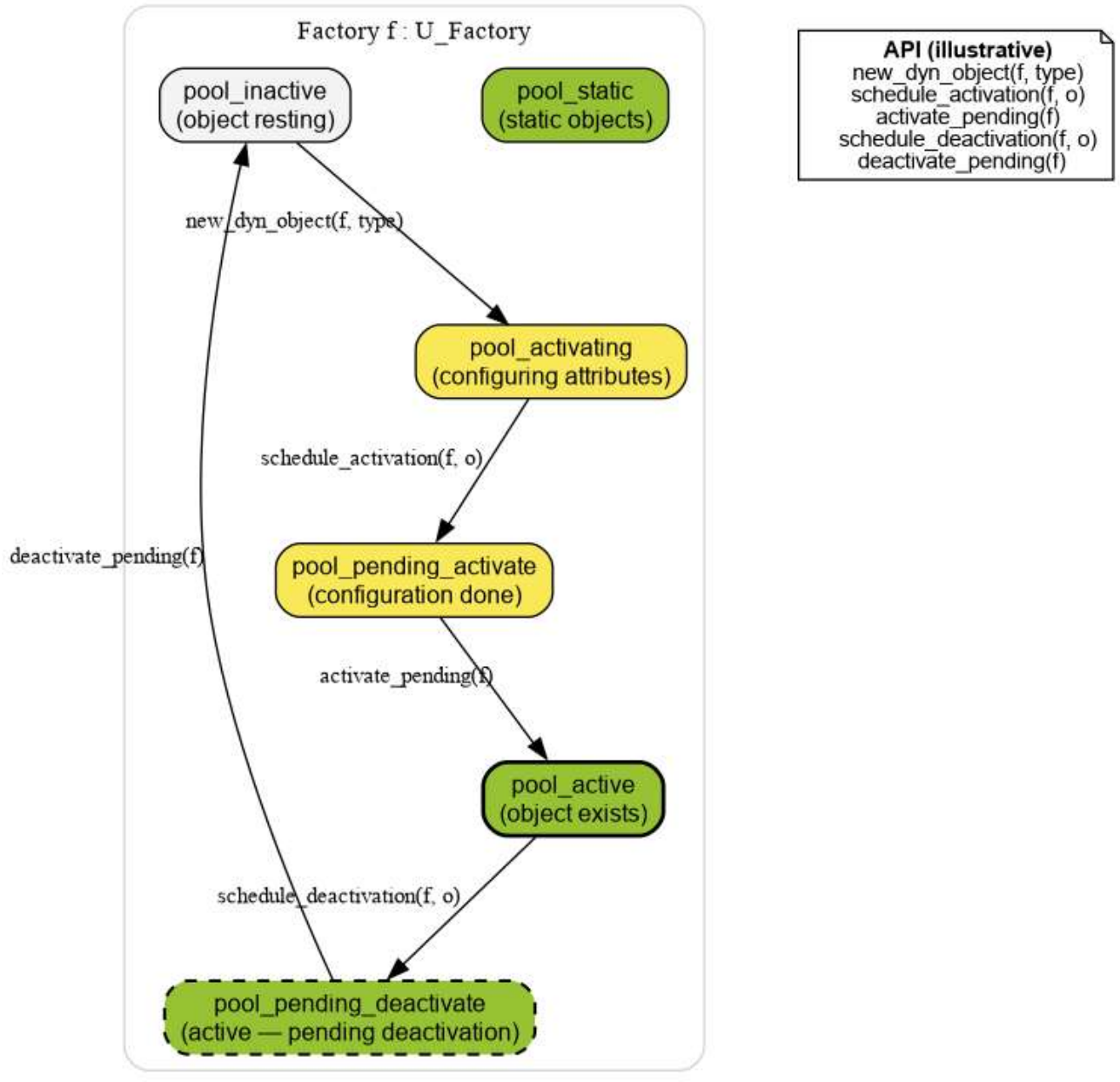



**Figure 11: Factory-managed object pools and lifecycle transitions**

### 8.1.7 Uniform Object Semantics and Nullary Design for Modular and Verifiable Modelling

A central design choice in the modelling framework is the adoption of total object semantics, in which all attributes and relations (i.e. attribute-based references in the object model) are defined for all objects. Instead of using NULL values or union types (e.g. option types, where a value may either contain a valid object or represent absence), each leaf type defines a dedicated nullary object, identified by a static attribute (nullary = True), which serves as a type-safe placeholder for absence of a value of the corresponding type. In this way, relations that would otherwise be undefined instead evaluate to the nullary object of the target type.

This choice was motivated by the goal of supporting modular and reusable modelling patterns, as well as by verification considerations. By modelling relations as total and uniformly defined, the modelling framework enables generic constructs and operations to be applied consistently across the model without special-case handling. For example, generic operations that compose or traverse relations can be defined once and applied uniformly, without requiring explicit checks for undefined or absent values, as all relations return well-defined objects. In formal models, including HLL-based models, the formalisation must capture all relevant aspects of the system semantics, including partial functions, union types, and NULL-like values. While implementation-level models may require such constructs to faithfully capture implementation semantics, they introduce additional modelling complexity and variability.

By modelling relations as total and representing absence through nullary objects, such modelling complexity is avoided, resulting in a more uniform and, in practice, scalable verification model.

The lifecycle discipline (see Section 8.1.6) complements this approach. Dynamic objects are created in an inactive state and may contain nullary values during configuration. Activation (as defined by the lifecycle discipline) is only permitted once all attributes are assigned non-nullary values and all consistency constraints are satisfied. Consequently, all active objects represent semantically valid system entities and are guaranteed to be free of nullary references. This ensures that incomplete or inconsistent configurations are not silently propagated into the active system state, but are detected explicitly at the point of activation.

These relations are total and may be evaluated for all objects, including inactive and nullary ones. For inactive or nullary objects, relations evaluate to nullary values (or corresponding canonical values, such as empty sets), ensuring closure under evaluation. Domain constraints and properties typically apply to active objects, which represent the current system state.

Nullary objects are therefore not used to represent optionality in the domain. Instead, they serve as placeholders in incomplete or inactive states, while optional relationships are modelled explicitly (e.g. via sets). This separation allows the framework to maintain totality and uniform typing while ensuring that all active system states satisfy declared invariants. Nullary objects thus remain confined to inactive or intermediate states and do not form part of semantically valid system configurations.

Finally, the design reflects the intended stage of use of the modelling framework. At earlier phases of the system lifecycle, experience indicates that introducing constructs such as partial functions or union types can lead to unnecessary verification overhead without corresponding benefit. The adopted approach therefore shifts consistency concerns into the modelling semantics and lifecycle rules, allowing verification effort to focus on higher-level system properties and requirements.

### 8.1.8 Separation of Data Instantiation and Architecture Instantiation

In the demonstrator, instantiation is uniformly performed by creating objects in the object model. What differs is the category of objects involved—Global Objects, Design Concepts, or Architecture Concepts—and the stage at which they are created. These object categories are instantiated in a deliberate and structured processing pipeline sequence, reflecting different sources of input data and modelling intent.

Static topology and trackside objects are instantiated from RDF-based infrastructure data, while architectural components are instantiated separately using parameterised configuration and generic Python code that applies a standard architecture to the imported data.

This staging supports clarity and reuse by making explicit which objects originate from infrastructure data, which represent domain concepts, and which reflect architectural modelling choices. It also avoids hard-coding architectural assumptions into data formats and simplifies validation activities by separating concerns between data-driven and architecture-driven aspects of the model.

**Taken together, the design principles described above (Section 8.1) establish the intended modelling structure and abstraction choices of the demonstrator. Applying these principles to realistic infrastructure data, non-trivial modelling scenarios, and formal verification workflows, however, revealed additional insights and constraints that could not be anticipated fully in advance. These findings, which emerged through implementation, validation, and toolchain interaction, are discussed in the following section (Section 8.2).**

## 8.2 Findings from Modelling and Toolchain Experiments

(Lessons learned from real use)

Applying the design principles from Section 8.1 in practice – using realistic infrastructure data, scenario execution, and formal toolchain workflows – revealed several findings that could not be fully anticipated in advance. These findings emerged through implementation, scenario-based execution, and interaction with the formal verification tool chain, and they reflect practical constraints, trade-offs, and requirements imposed by the chosen abstraction level.

The subsections that follow document selected findings that are considered particularly relevant for understanding the strengths and limitations of the modelling approach. They include insights related to typing discipline, formal model structure, toolchain behaviour, and assumptions in underlying data representations. Together, these findings complement the design intent described earlier and provide empirical grounding for the implications discussed in Section 8.3.

### 8.2.1 Flexible Typing in Attribute Signatures and the Need for Proof Obligations

The ontology framework used in the demonstrator allows a high degree of flexibility in how types are used in attribute signatures. In particular, the signature of an attribute may refer not only to concrete leaf classes, but also to **class types, mixin types, or specialisation types** defined in the ontology. This capability proved valuable for expressing generic relationships and reusable modelling patterns without prematurely committing to specific concrete types.

This form of flexible typing is non-standard, even for ontology-based modelling approaches, and its implications only became fully apparent through implementation and formalisation. While such flexibility increases expressivity, it also introduces additional requirements on **type checking and**

**verification**, since the validity of certain type constraints cannot be determined locally at the point where an attribute is defined.

In the demonstrator, some type constraints depend on the **global structure of the ontology's sort hierarchy**, including inheritance relations, mixins, and specialisations. As a result, it is not always sufficient to perform type checking during ontology definition or object model generation alone. Instead, certain constraints must be verified against the formal model, where the complete sort hierarchy is available.

To support this, the demonstrator generates, from the ontology's directed acyclic graph (DAG) of sorts, a **mirrored sort hierarchy** with fresh type names used explicitly in the formal HLL model. This mirrored hierarchy enables formulation of **proof obligations** that assert required type relationships, such as subtype constraints induced by attribute specialisation or restrictions. These proof obligations are then discharged using formal verification techniques.

This approach illustrates a key lesson from the work: increasing modelling expressivity—here, through flexible use of types in attribute signatures—inevitably shifts some validation effort from local checks to **global verification mechanisms**. The experiments confirmed that this trade-off is manageable in practice, but only when formal verification infrastructure is treated as an integral part of the modelling framework rather than as an optional add-on.

The experience also reinforces a broader design principle underlying the demonstrator: modelling flexibility should be introduced deliberately and accompanied by explicit verification support. When this discipline is followed, flexible typing can be used to improve generality and reuse without compromising the soundness or analysability of the resulting models.

### 8.2.2 Mirrored Sort Hierarchies in Formal Models

The flexible typing approach described in Section 8.2.1 has direct consequences for how the ontology and object model can be represented in a formal language. In particular, the formal model must be able to reason about the full structure of the ontology's type system, including inheritance relations, mixins, and attribute specialisations.

In the demonstrator, the ontology defines a directed acyclic graph (DAG) of types, where nodes correspond to class types, mixin types, and specialisation types. These types are meaningful at the modelling level and are used explicitly in attribute signatures and specialisation rules. However, this structure cannot be mapped directly into the formal model using only the leaf object types, since doing so would lose information required to verify type-related constraints.

To address this, the demonstrator generates a **mirrored sort hierarchy** in the formal HLL model. For each type in the ontology's DAG, a corresponding formal sort is introduced, using a fresh type name. The relationships between these sorts mirror the structure of the ontology: subtype relations in the ontology are reflected as sort inclusions in the formal model, and equivalence classes of types are represented explicitly.

This mirrored hierarchy serves several purposes. First, it allows the formal model to express proof obligations that refer to non-leaf types, such as mixins or specialisation sorts, even though concrete objects are always instances of leaf types. Second, it enables verification of constraints that depend on type relationships rather than on individual objects, such as ensuring that attribute restrictions are consistent with declared supertypes.

The need for a mirrored sort hierarchy only became apparent through experimentation with formal model generation and verification. Early attempts that relied solely on leaf-level types proved

insufficient to express and verify certain classes of type-related constraints. Introducing the mirrored hierarchy made it possible to formulate these constraints precisely and to discharge them using the model checker.

This experience illustrates a broader lesson about ontology-based formal modelling: When the ontology's type system is used as a first-class modelling artefact, the formal model must reflect that structure explicitly. Attempting to collapse or simplify the type hierarchy too early can undermine verifiability and obscure modelling assumptions. The mirrored sort hierarchy therefore represents a deliberate design choice that preserves semantic information from the ontology and supports sound formal reasoning.

### 8.2.3 Enumerated Types as Sorts in HLL

In the formal HLL model generated by the demonstrator, **enumerated types are represented as sorts**, rather than being encoded as integer ranges or uninterpreted constants. While this modelling choice may be unfamiliar to some readers, it proved beneficial in practice and contributed to improved clarity and uniformity of the formal model.

By representing enumerations as sorts, enumerated values become first-class typed entities in the formal language. This allows constraints involving enumerated attributes to be expressed using the same operators and typing mechanisms as constraints over object types. As a result, properties that combine object relationships and enumerated state—such as mode flags, authority types, or directional attributes—can be formulated in a more direct and readable way.

This approach also simplifies reasoning about attribute domains. Instead of relying on numeric encodings or implicit assumptions about value ranges, the formal model explicitly constrains enumerated attributes to the intended set of values through the type system itself. This reduces the risk of accidental misuse and makes modelling assumptions more transparent to both tool support and human readers.

The decision to encode enumerations as sorts aligns naturally with the ontology-based modelling approach adopted in the demonstrator. Since enumerations are already defined explicitly in the ontology, mapping them to formal sorts preserves semantic structure across the transition from ontology to formal model.

This encoding supports a clear conceptual correspondence between the executable Python model and the formal HLL model by keeping enumerated domains explicit and well typed in both representations, even though their concrete realisations differ.

The experience suggests that, in ontology-based formal modelling, treating enumerations as first-class sorts is a pragmatic choice that improves model comprehensibility and supports systematic verification, particularly in models that already make extensive use of typed object relations.

### 8.2.4 Hard-Coded Assumptions in LCF Revealed by Real Data

Applying the modelling framework to real infrastructure data provided by Trafikverket revealed several **hard-coded assumptions** associated with the practical use of the Layout Configuration Format (LCF)[3] that are sufficient for its original modelling scope but become limiting when modelling more general or dynamic scenarios.

[3] More precisely, these hard-coded assumptions arise from conventions adopted by tools supporting LCF, rather than from the LCF specification itself.

One such assumption is that a node contains **at most one point object**. In the RDF-based topology data from Trafikverket, examples exist where two point objects are separated by an edge of zero length and therefore effectively reside within the same node. While this situation can be represented indirectly in LCF, it exposes a structural limitation when attempting to model topology more faithfully or when generalising modelling assumptions.

A second assumption concerns how **object direction** is represented. In LCF, directionality is specified using a node index. However, real infrastructure includes signals and other trackside objects that may face in multiple directions, or that are only relevant for a subset of the possible passage relations through a node. Encoding directionality solely via a single node index is therefore insufficient to capture such cases precisely.

The ontology-based modelling approach used in the demonstrator does not rely on these assumptions. In the upper ontology, a node may contain multiple objects of the same type if needed, and object directionality is represented explicitly using a *facing* attribute defined as a set of node-connector pairs. This representation allows directionality to be expressed relative to specific passage relations, rather than being tied to a single connector index.

These findings illustrate a broader lesson: assumptions that were reasonable and efficient in an original configuration context may need to be revisited when modelling more general system behaviours or when integrating data from heterogeneous sources. The ontology-based approach made such assumptions explicit and provided a structured way to generalise them, enabling a more faithful representation of real infrastructure data without compromising modelling consistency.

### 8.2.5 Toolchain Findings: Model Checker Behaviour and sHLL Semantic Gap

The depth and complexity of the formal models generated in the demonstrator led to several noteworthy findings regarding the supporting formal verification toolchain.

When applying the model checker to formal HLL models generated from the combined upper and lower ontology, a number of **runtime errors** were uncovered. These errors were not due to violations of the HLL language specification but rather emerged from the interaction between complex sort hierarchies—including equivalence classes and mirrored sort structures—and the implementation of the model checker itself. The issues were subsequently analysed and corrected, resulting in improvements to the model checker.

This experience demonstrates that ontology-based modelling at scale can exercise formal tools in ways that simpler or more conventional models do not. In this case, the modelling approach served not only as a consumer of formal verification technology, but also as a **stress test** that revealed previously unknown limitations in the toolchain.

In parallel, experiments were conducted using **sHLL**, an imperative extension of HLL, with the intention of representing behavioural models—particularly behaviour trees—in a more direct way. These experiments were ultimately abandoned after identifying a **semantic gap** between the execution model assumed by sHLL and the object model–based interface used in the demonstrator.

**Specifically, HLL functions invoked from (imperative) sHLL code can only refer to values in the discrete synchronous states of the underlying HLL execution model.** While sHLL syntactically supports calling HLL functions, the semantics of such calls are tied to HLL state transitions rather than to the evolving state of the imperative sHLL code. As a result, using HLL functions as an interface to an object model from sHLL is not semantically meaningful in this context.

This finding is an important negative result. It highlights that extending a formal language with imperative constructs does not automatically make it suitable for object-oriented or object model–based abstractions. More generally, it reinforces the importance of aligning formal language semantics with the modelling abstractions they are intended to support.

### 8.2.6 Role of Graphical Visualisation in Scenario-Based Validation

Scenario-based validation using generated test cases highlighted the importance of **graphical visualisation as a natural and efficient way to validate modelling abstractions**. Visualisation enables modelling issues to be detected by inspecting execution traces in a topology viewer, in a form that directly reflects the level of abstraction at which requirements and system behaviour are being explored.

A typical usage of visualisation in the demonstrator is to *play* execution traces generated from test cases, with respect to a chosen test objective, and to validate visually that expected events and state changes occur during realistic movement scenarios over instantiated infrastructure data. In this way, visualisation supports direct assessment of whether modelling assumptions and abstractions behave as intended.

Visualisation proved particularly effective for assessing **path-based abstractions and movement-related behaviour**. Issues such as incorrect path extents, unintended overlaps, or unexpected interactions between train movements and protection paths were often immediately apparent when viewed graphically in the context of the infrastructure topology. These observations could then be traced back to modelling assumptions, configuration data, or test case definitions, supporting iterative refinement of both the models and the underlying abstractions.

This experience is consistent with earlier findings reported in Shift2Rail TD2.7, for example in the Alex case study, where visual inspection of execution traces played a key role in identifying specification and modelling issues. The demonstrator reinforces this lesson by showing that visualisation is not merely a presentation aid, but an effective validation mechanism when working with executable models and abstraction-level representations.

A related practical finding concerned the **availability of suitable visualisation support**. Prior to the demonstrator work, tooling existed to visualise LCF-based track topology and fixed-block paths and areas. However, this support was inherently limited to fixed-block representations and could not accommodate paths whose extent may begin and end at arbitrary positions, as required for moving block–style abstractions and train-centric modelling.

During the demonstrator development, this limitation was recognised as a significant obstacle to effective validation of dynamic, path-based modelling concepts. As a result, a more general visualisation interface was specified, building on the same LCF railyard representation but extended to support directed segments with known length and arbitrary path extents. Once this interface became available and was implemented, progress in modelling, scenario execution, and validation accelerated markedly.

Taken together, these observations show that, for spatial and path-centric system models, adequate visualisation is a **key enabler of validation**. Being able to inspect dynamic path evolution in context proved essential for identifying modelling issues and iterating efficiently on abstractions. Without such visual support, both executable and formal validation become harder to apply effectively in practice.

### 8.2.7 Dormant Attributes as a Design-for-Verifiability Mechanism

During validation activities, it became apparent that a **natural and principled modelling of lifecycle relationships between train-centric dynamic object types**—such as CUT (Central Unit Train), TLP (Train Location Path), Authority Base (AB), Authority Path (AP), and SWEPT paths—exposes the inherent complexity of these relationships in the formal model. These object types are subject to strict creation and deletion order constraints (e.g., a CUT must exist before a TLP can be created, and an AB must exist before an AP within it), and representing these constraints uniformly using incarnation attributes is conceptually well-founded.

The resulting formal model complexity is therefore not artificial but reflects the underlying semantics. In practice, however, this complexity interacts unfavourably with current formal verification engines, in particular with the eager partial evaluation strategy used by the HLL model checker, leading to scalability bottlenecks during validation.

To address this, the demonstrator introduced the notion of **dormant attributes**. Dormant attributes are identified by a naming convention (prefix z_) and are treated as statically assigned, reflecting associations that are effectively invariant across the lifetime of a dynamic object. For example, the association between a CUT and its corresponding TLP can be represented using a dormant attribute, since the lifecycle rules already ensure that this relationship remains stable whenever the objects exist.

Using dormant attributes reduces formal model complexity by allowing certain relationships to be treated as static, even though the objects themselves are dynamic. This enables more efficient validation in the formal HLL model while preserving the intended semantics of the train-centric abstractions.

This experience illustrates a broader design-for-verifiability lesson: **not all relationships between dynamic objects need to be modelled dynamically**. When strong lifecycle constraints already limit how objects can relate to each other, it can be beneficial to encode implied invariants explicitly, trading a small amount of modelling generality for substantial gains in analysability and scalability.

### 8.2.8 Cost and Practical Use of Derived Relations

Practical use of the demonstrator highlighted that not all derived relations have the same computational characteristics or should be treated equally in validation workflows. Some relations are both fundamental and inexpensive to compute, while others are useful but computationally costly, and therefore best treated as optional or on-demand analyses.

An example of a derived relation that is inexpensive and central is the set of instances of the Global Object concept **directed segment**. Directed segments form a basic abstraction used pervasively in path and area-related reasoning, and their computation scales well even for large infrastructure data sets. Making such relations readily available is therefore appropriate and beneficial.

By contrast, computing the **transitive closure** of the next relation over directed segments proved to be significantly more expensive. While the next relation itself is inexpensive and readily available as input, deriving its transitive closure involves graph-wide computation whose complexity grows rapidly with the size of the infrastructure model, making it unsuitable for routine use on large data sets. For the largest data set used in the demonstrator, computing this transitive closure required on the order of one hour on a standard laptop, making it unsuitable as a routinely available relation.

This experience underlines a practical modelling lesson: derived relations should be classified according to both their semantic importance and their computational cost. Relations that are

fundamental and inexpensive can be made readily available, whereas more expensive derived relations should be computed selectively, based on the needs of specific validation tasks. Treating such relations as optional avoids unnecessary overhead and supports scalable application of the modelling framework.

### 8.2.9 Message Interfaces Benefit from Compact Path-Extent Encodings

The choice to represent path extent internally using a set-based encoding (via x_first, x_last, and x_inner) is well suited for validation and efficient membership-based reasoning. This representation supports generic operations such as overlap checking, consistency verification, and test case generation, and integrates naturally with the path- and area-based abstractions used throughout the demonstrator.

At the same time, this internal representation highlights a practical interface consideration. External systems or environment models that command the creation or modification of a path typically exchange information through message payloads. Transmitting **sets of object references** in such payloads is often undesirable. Object references typically have local meaning within a given model instance and may depend on lifecycle state or allocation context. Set-valued payloads also scale poorly with path length, leading to large or variable message sizes, and they expose internal representation choices at system interfaces rather than expressing intent. From a validation perspective, reasoning about set-valued payloads often requires more global analysis, which can complicate scalability.

A compatible engineering approach is therefore to **decouple external message encodings from the internal representation** used by the object model. Instead of carrying an explicit set of inner segments in a message payload, a command to create or modify a path can be expressed compactly using:

- the first directed segment (x_first),
- the last directed segment (x_last), and
- a bounded procedural encoding of intermediate routing choices, for example a fixed-width bit sequence (such as a 32-bit integer).

In such an encoding, each bit represents a local routing decision at the next branching point along the path (e.g. 0 = left, 1 = right). Given the current segment and the topology relation for directed segments, the bit sequence deterministically selects the next segment at each step until the last segment is reached. Bits beyond what is required for a particular path instance are “don’t-care”. The internal set-based representation can then be reconstructed by decoding this compact payload and materialising the corresponding x_inner set.

This finding illustrates a general modelling lesson: a representation that is well chosen for analysis and validation is not necessarily the best format for interchange. Where appropriate, lightweight encoding and decoding layers can translate between compact, intent-oriented message formats and analysis-oriented internal models, preserving modelling semantics while supporting efficient communication and scalable validation.

## 8.3 Implications for Modelling and Verification Practice

(Interpretation & positioning)

This section explains **what the design choices and findings imply more generally**.

The design choices and experimental findings described in Sections 8.1 and 8.2 have several broader implications for modelling and verification practice in the context of evolving railway signalling systems-of-systems. While these implications are drawn from a specific demonstrator, they are not tied to a particular signalling architecture or technology and are therefore relevant beyond the immediate scope of D30.3/D30.4.

### 8.3.1 Abstraction Level Selection is Central to Verifiability

A central implication of this work is that verifiability is determined primarily by abstraction choices, rather than by the choice of verification techniques alone. Selecting an abstraction that is too detailed can lead to models that are expressive but intractable, while overly coarse abstractions risk omitting critical distinctions required for meaningful validation.

The demonstrator shows that identifying the ontology as the primary carrier of abstraction allows modelling scope and expressivity to be controlled explicitly. When the abstraction is chosen deliberately at the ontology level, executable and formal models can be generated mechanically and analysed consistently. This suggests that early investment in abstraction design can significantly reduce downstream verification effort.

### 8.3.2 Expressivity Must Be Matched by Verification Infrastructure

Several of the experimental findings highlight a recurring pattern: increasing modelling expressivity introduces corresponding requirements on verification infrastructure. Flexible typing in attribute signatures, use of mixins and specialisations, and rich path abstractions all improve generality and reuse, but they also require explicit support for global type reasoning and proof obligations.

The work demonstrates that such expressivity can be supported in practice, but only when verification mechanisms are treated as first-class modelling concerns. Attempting to introduce expressive abstractions without adequate formal infrastructure risks producing models that are difficult or impossible to analyse systematically.

### 8.3.3 Separation of Concerns Enables Scalable Validation

The modelling framework consistently applies separation of concerns at multiple levels: abstraction versus realisation (ontology versus object model), structure versus behaviour, topology versus extent, and data instantiation versus architecture instantiation.

This separation enables scalable validation by isolating complexity and allowing generic mechanisms—such as consistency checking, test case generation, and scenario execution—to be reused across different contexts. The experience suggests that such separation is not merely a software engineering convenience, but a prerequisite for sustainable validation in system-of-systems settings.

The same principle applies to derived relations: while some relations are fundamental and inexpensive to compute, others—such as transitive closures over large topological graphs—are inherently costly and should therefore be treated as optional, on-demand analyses rather than as universally available modelling primitives.

A related separation-of-concerns applies to interaction modelling. The demonstrator distinguishes between the *logical intent* of message exchange (e.g. request, response, notification) and the *concrete representation* of messages in the object model. Message patterns are treated as a conceptual view rather than hard-coded ontology types, while concrete messages are characterised

by their payload structure. This avoids premature protocol commitments and supports reuse and validation across different system contexts.

### 8.3.4 Reuse of Formal Model Generation Infrastructure

A further implication of the approach concerns the efficient use of specialised expertise in formal methods. Formal model generation in the demonstrator is driven primarily by the upper ontology, which defines the generic concepts, typing discipline, and structural constraints required for formal analysis. As a result, the formal model generation infrastructure is largely independent of the domain-specific concepts introduced in lower ontologies.

This separation allows formal modelling expertise to be concentrated at the level of the upper ontology and its associated generation mechanisms, while domain experts can extend the modelling framework through lower ontologies without re-implementing or re-validating the formal encoding. In practice, this supports reuse of formal modelling infrastructure across domains and contributes to more scalable and sustainable application of formal analysis in early-phase modelling activities.

### 8.3.5 Compact Encodings for Path Extent Updates at Message Interfaces

A broader implication is that internal representations chosen for analysability should not automatically be exposed at message interfaces. As illustrated by the path-extent update case in Section 8.2.9, compact intent-oriented encodings can provide predictable payload sizes and cleaner interface semantics, while still allowing decoding into analysis-friendly internal representations.

### 8.3.6 Scenario-Based Validation Complements Formal Verification

The scenario-based executable validation activities complement formal verification by exercising the modelling framework under realistic dynamic evolution. Execution traces generated from generic test cases make it possible to assess plausibility, consistency, and internal coherence of modelling abstractions even in the absence of complete behavioural models.

This combination of executable scenarios and formal verification supports a broader notion of validation than either approach alone. It allows issues to be detected early, supports iterative refinement, and provides tangible feedback to stakeholders who may not be familiar with formal methods.

### 8.3.7 Toolchains are Part of the Modelling System

A further implication is that formal verification tools and execution environments should be viewed as integral parts of the modelling system. The experiments revealed that non-trivial modelling abstractions can expose limitations and assumptions in tools, and that tool behaviour may influence modelling choices.

Treating tools as fixed black boxes can therefore constrain modelling practice unnecessarily. Conversely, close interaction between modelling and toolchain development can lead to improvements in both, as demonstrated by the identification and correction of issues in the formal model checker during this work.

### 8.3.8 Early Validation Supports Convergence and Reuse

Finally, the work reinforces the value of early validation for achieving convergence on requirements and modelling assumptions. By enabling validation activities at the stage where abstraction choices

and requirements are still fluid, the modelling approach supports informed decision-making and reduces the risk of late-stage rework.

This has implications for reuse as well. A modelling framework that promotes early convergence and disciplined abstraction is more likely to be reusable across projects and domains, as it captures stable conceptual structure rather than project-specific design decisions.

**Overall, the implications discussed in this section reinforce the central role of abstraction design in achieving effective and sustainable verification. The demonstrator shows that when abstraction choices are made explicitly—at the ontology level—and are supported by disciplined object model realization and verification infrastructure, it becomes possible to combine expressivity with analysability in a controlled manner. These implications do not depend on a specific signalling architecture or technology. Rather, they highlight general considerations for early, model-based analysis and validation in system development, particularly in system-of-systems contexts where requirements, assumptions, and interfaces are still evolving. In this sense, the work supports a shift from late implementation-centric validation toward earlier, model-based reasoning grounded in well-chosen abstractions.**

## 8.4 Relation to the Digital Twin Paradigm

The validation approach supported by the demonstrator can be viewed as **aligned with aspects of the digital twin paradigm**, in the sense that an executable model of the ETCS trackside system is instantiated using real infrastructure data and exercised using realistic operational inputs. This allows requirements and behaviour to be validated against concrete scenarios derived from real-world configurations, while remaining focused on early-phase analysis rather than operational deployment.

In particular, the demonstrator **supports validation scenarios in which** an instantiated trackside model **can be driven** by ETCS radio-based messages exchanged between RBC and onboard systems, either synthetically generated or, in principle, derived from recorded operational data. **While such validation has not been performed within the scope of the demonstrator to date**, the modelling approach and execution infrastructure are designed to enable this form of validation when suitable data access is available.

When used in this way, the instantiated model functions as a **validation-oriented, digital-twin–like representation** of the trackside system, suitable for analysing requirements, safety constraints, and behaviour under realistic conditions.

The demonstrator does not aim to constitute a full digital twin of the railway system, nor to support continuous synchronisation with live operational systems. Instead, it provides a **scoped, laboratory-based model** intended to support requirements validation, design exploration, and scenario-based analysis, in line with the System Inception use case and early development phases.

## 8.5 Limitations and Scope Boundaries

The demonstrator framework was developed to explore a combination of modelling techniques and systematic validation mechanisms intended to support **early requirements validation**. The results presented in this report should therefore be understood primarily as a **proposal of a modelling abstraction and methodology**, rather than as a complete system model or implementation.

Within the chosen abstraction level, the demonstrator has the following scope limitations:

- **Behavioural completeness:** The demonstrator does not provide detailed behavioural models for all subsystems. Behavioural modelling is used selectively to exercise and validate modelling abstractions and consistency constraints, not to represent full subsystem logic.
- **System interaction realisation:** The conceptual model for system-of-systems interaction is only partially realised in the demonstrator. While message-based interaction abstractions are defined and exercised conceptually, not all interaction patterns are implemented end-to-end.
- **Operational claims:** The demonstrator does not make claims regarding operational performance, production readiness, or suitability for safety case approval. It is not intended to replace design, implementation, or certification activities carried out in later development phases.

These limitations reflect **deliberate scoping decisions** to keep the demonstrator focused on abstraction design and early-phase validation (the primary objectives of the work), rather than on implementation or certification concerns.

## 8.6 Reuse Potential

Despite its prototype nature, the modelling approach explored in the demonstrator exhibits **substantial reuse potential**. This potential lies primarily in the **abstraction principles, modelling discipline, and validation mechanisms** introduced.

At the core of this reuse potential is the separation between a **domain-independent upper ontology**, a **domain-specific lower ontology**, and a mechanically generated **object model interface**. This separation allows the same modelling and validation approach to be applied across different signalling domains by adapting or replacing the lower ontology, while preserving the upper ontology and object model generation infrastructure. As a result, the approach is not inherently tied to a specific ETCS configuration (including moving-block–oriented variants), but is applicable across ETCS architectures and other railway signalling contexts, such as CBTC.

Reuse is particularly strong at the level of:

- abstraction design and ontology structuring principles;
- the object model methodology as a common semantic interface for requirements, behaviour, and validation;
- instantiation pipelines that separate infrastructure data from architectural modelling choices; and
- validation mechanisms combining executable scenarios with selected formal analysis.

A key enabler of reuse in the proposed methodology is the separation between **conceptual richness and operational simplicity**. While the ontology may require structural complexity to support modularity, reuse, and systematic abstraction, this complexity is internalised through object model generation. As a result, users of the modelling framework interact with a uniform and simple object model interface, independent of how concepts are organised or refined in the ontology.

This separation of concerns is particularly important when applying the methodology at scale or across organisational boundaries, where different stakeholders may contribute to ontology definition while others focus on modelling, validation, or analysis activities. By isolating conceptual complexity at the ontology level and exposing a stable operational interface, the approach supports sustained reuse without requiring all users to engage with ontology engineering details.

The reuse potential outlined above provides the foundation for a range of **future exploration directions**, which are described in more concrete terms in the following subsection.

### 8.6.1 Additional Directions for Future Exploration

Building on the reuse potential discussed above, several further directions for future exploration were identified during the demonstrator work. These topics have not been implemented or evaluated in practice, but emerged naturally from the modelling approach and from considerations of how the demonstrator could be applied to more advanced system-of-systems scenarios. They are outlined here to illustrate the breadth of validation questions that the framework could support in future work.

One area of interest concerns **dynamic evolution of configuration and architecture at run time**. This includes extending the demonstrator to support dynamic modification of global objects, allowing aspects of static configuration data to be adapted during execution. Closely related is the dynamic creation and deletion of architectural components. For example, one or more interlocking (IXL) components could be shut down temporarily while the system-of-systems continues to operate in a degraded mode, and later replaced by newly instantiated components that assume responsibility for the affected areas. Exploring such scenarios would enable identification and validation of requirements for dynamic reconfiguration and resilience in server-like or cloud-based architectures for ETCS trackside systems, whether realised as traditional IXL and object-controller components or as applications deployed on shared computing platforms.

Another set of directions relates to **reuse and integration of existing formalised component models**. The demonstrator could be extended to integrate models developed in earlier work, such as the formal methods–based model of Trafikverket's Alex level crossing developed in Shift2Rail TD2.7. Integrating such components would enable execution-based system-of-systems testing that combines multiple formally specified subsystems within a shared modelling and validation framework. Related to this is the potential use of digital-twin representations for selected components, enabling hybrid validation scenarios that combine abstract models with higher-fidelity component behaviour.

A further direction for future exploration concerns **alignment with ontology and information-modelling initiatives promoted by the European Union Agency for Railways (ERA)**. The ERA ontology [17] and related information models focus primarily on standardised representation of infrastructure, assets, and interoperability-related information. Investigating systematic mappings between such ERA-defined concepts and the **upper ontology** used in the demonstrator could enable reuse of agreed terminology and data structures for instantiation of static configuration elements, while retaining the demonstrator's ontology and object model as the basis for behavioural modelling and requirements validation. Such alignment would support interoperability between data-centric standards and validation-oriented modelling without requiring behavioural semantics to be imposed on existing ERA ontology.

Further opportunities arise in **modelling realistic environment and human decision-making behaviour**. One example is the development of a behaviour tree–based executable model that emulates the behaviour of a dispatcher in traffic management. Such a model could interact with ETCS trackside via the same interface used by a Traffic Management System (TMS), providing realistic and regulation-constrained inputs to the trackside system. This would enable validation of ETCS trackside behaviour under operationally meaningful conditions, while keeping the focus on requirements and interaction semantics rather than on implementation details.

Future work may refine **how logical message intent maps to concrete encodings**, building on the conceptual separation between message patterns and payload representation identified in this work.

The demonstrator could also be extended to support **state persistence and reinitialisation**. Loading saved state from disk to reinitialise a subsystem would enable execution-based workflows such as checkpointing, shutdown and restart of processes, or migration of execution between nodes. This capability would support validation of requirements related to fault handling, recovery, and continuity of operation, and is conceptually aligned with scenarios described in the Moving Block specification [2].

Finally, the modelling approach opens the possibility of **AI-assisted behavioural model authoring**. Instead of relying solely on manually developed reference designs or implementation models, the demonstrator could support the generation of behavioural models using prompt-based AI agents or “vibe coding” approaches. Python was chosen as the executable modelling language partly for this reason, as modern AI-based development tools are particularly effective at generating Python code. Constraining AI-generated models to operate strictly through the object model interface would allow such models to be subjected to the same executable and formal validation mechanisms, enabling exploration of how AI-assisted modelling can reduce effort while preserving verifiability of critical properties.

# 9 CONCLUSIONS AND NEXT STEPS

This chapter concludes the report by summarising the contributions of the work, clarifying their scope and interpretation, and outlining relevant next steps. It consolidates the methodological position established in earlier chapters and places the demonstrator's results in the context of early, model-based reasoning for evolving railway signalling systems-of-systems.

## 9.1 SUMMARY OF CONTRIBUTIONS

This report has presented the modelling approach, abstraction choices, and validation activities developed within the Europe's Rail Joint Undertaking project R2DATO and demonstrated through deliverable D30.3. Together, the demonstrator and this report address the challenge of **early, systematic validation of requirements** for evolving railway signalling systems-of-systems.

The work makes the following key contributions:

- A **clearly defined modelling abstraction**, expressed explicitly through an ontology that captures the conceptual system domain and determines modelling scope and expressivity.
- An **object model generation approach** that operationalises the ontology into executable and formal modelling interfaces, enabling consistent instantiation, analysis, and validation.
- A disciplined treatment of **dynamic objects, lifecycle semantics, and invariant enforcement**, supporting meaningful reasoning in the presence of dynamic system state.
- A set of **train-centric and path-based modelling abstractions**, suitable for representing movement permissions, constraints, and non-nominal situations.
- Demonstration of **instantiation from real infrastructure data** and application of executable and formal validation mechanisms, including scenario-based validation using generated test cases.
- Empirical findings that reveal **practical requirements on modelling infrastructure and toolchains**, including implications for type systems, formal representations, and verification support.

Taken together, these contributions establish a coherent foundation for disciplined, early, model-based reasoning about complex signalling systems-of-systems.

## 9.2 POSITIONING WITHIN EUROPE'S RAIL OBJECTIVES

The work described in this report aligns with Europe's Rail objectives by strengthening early-phase validation capabilities for interoperable, cost-effective, and future-ready signalling systems. In particular, it contributes to:

- strengthening **early-phase validation** of requirements and architectural assumptions,
- enabling **objective, model-based analysis** before construction or procurement,
- supporting **harmonisation** through shared conceptual models and explicit abstraction,
- and reducing reliance on late-stage, implementation-centric validation.

By focusing on modelling discipline and abstraction rather than on specific solutions, the approach is applicable across multiple signalling variants and organisational contexts.

## 9.3 Interpretation of Results and Scope

Demonstrator D30.3 and this report should be understood primarily as a proposal of a modelling methodology and abstraction framework for early, repeatable validation of requirements in evolving railway signalling systems-of-systems. The implemented demonstrator serves as a proof-of-concept realisation of this approach, exercising its core mechanisms—ontology definition, object model generation, lifecycle semantics, and combined executable and formal validation—on realistic infrastructure data. The principal contribution therefore lies not in a finished signalling system or toolchain, but in the articulation and empirical evaluation of a structured abstraction that balances expressivity with analysability.

The prototype demonstrates feasibility, tractability, and scalability at realistic infrastructure scale, while remaining intentionally scoped and partial with respect to complete behavioural modelling, full interaction realisation, and operational deployment.

The abstraction realised in the demonstrator reflects a deliberately general train-centric model in which movement authorities (AB/AP) are defined independently of particular architectural constraints such as fixed routes or dedicated interlocking subsystems. This generality makes it possible to derive more constrained architectural variants as refinements within the same modelling framework.

For example, a typical ETCS Level 2 architecture using route-based interlocking could be represented by introducing explicit constraints that restrict Authority Base paths to routes determined by a separate interlocking subsystem and by enforcing single-train occupancy rules at the route level. Conversely, hybrid train detection architectures could be represented by relaxing such constraints, allowing multiple trains within defined extents under controlled conditions. In these cases, the underlying ontology structure, object model generation mechanisms, lifecycle semantics, and validation infrastructure remain unchanged. Architectural variation is realised through additional constraints and behavioural definitions in the lower ontology, rather than through structural modification of the modelling framework itself.

The most general abstraction is defined once; architectural variants are obtained by constraining it, not by redefining it. Although these refinements have not been fully exercised in the present demonstrator, the modelling framework is explicitly structured to support such controlled variation.

A further implication concerns the separation between the domain-independent upper ontology and domain-specific lower ontologies. This separation establishes a stable abstraction boundary between the generic modelling framework and specific signalling architectures. Distinct configurations—such as ETCS Level 2 variants with or without lineside signals, ETCS with ATO, or CBTC architectures with differing interlocking responsibilities—can in principle be captured as separate lower ontologies while reusing the same upper ontology, object model generation mechanisms, lifecycle semantics, and validation infrastructure. Under such a scaling scenario, architectural variation primarily affects the lower ontology and behavioural models, while the validation framework itself remains structurally stable.

Rather than treating modelling, lifecycle semantics, and validation as separate engineering activities, the approach integrates abstraction selection, operational realisation, and verification infrastructure into a single coherent methodological framework. The demonstrator therefore illustrates not only how signalling concepts can be represented, but how abstraction can be structured so that validation and reasoning scale with architectural evolution rather than fragmenting across system generations.

A useful way to interpret the modelling approach is to view it as introducing a disciplined modelling layer in which all relations are well-defined and no undefined values occur. Instead of allowing partial or optional values, absence is represented explicitly through nullary objects, and incomplete configurations are separated from semantically valid system states by the lifecycle mechanism. This establishes a clear distinction between intermediate and valid states: inactive objects may be incomplete, while active objects are guaranteed to satisfy all declared constraints. As a result, models become easier to reason about, as there are no undefined expressions or implicit assumptions, and the uniform treatment of relations supports modular and reusable modelling patterns that can be applied consistently across the model. This interpretation also clarifies why the approach is particularly suited to early phases of system development, where clarity, consistency, and validation of required behaviour are prioritised over detailed representation of implementation-specific constructs.

The approach can also be understood as an instance of design-for-verifiability, in which modelling constructs are deliberately structured so that consistency and analysability are ensured by construction, rather than through additional verification effort.

## 9.4 Next Steps

The work presented in this report establishes a methodological foundation rather than a finished system. Several natural next steps follow from this foundation.

First, behavioural coverage can be extended by developing more complete behavioural models for selected subsystems and interaction patterns, building on the established object model interface and lifecycle semantics.

Second, the message-based interaction abstraction can be further realised and exercised, enabling more complete validation of system-of-systems scenarios, including richer coordination between trackside components and external systems.

Third, the approach can be applied to additional infrastructure data sets and architectural variants, including configurations that combine ETCS with interlocking subsystems, hybrid train detection, or alternative traffic management structures. Such applications would allow the methodology to be exercised and evaluated further across a broader range of signalling contexts.

Fourth, validation could be extended using recorded ETCS radio-based message data, subject to availability, to assess how well the train-centric abstractions align with operational communication patterns.

Finally, the modelling framework provides a structured basis for tool-assisted behavioural modelling, including exploration of AI-supported authoring of behavioural models constrained by the object model interface. This would allow further evaluation of how modelling effort can be reduced while preserving analysability and explicit abstraction.

Taken together, these directions indicate that the demonstrator provides a scalable and adaptable foundation for continued development of early, model-based requirements validation in evolving railway signalling systems-of-systems.

## 9.5 Concluding Remarks

This work demonstrates that explicit abstraction design, combined with disciplined operational realisation and integrated validation mechanisms, provides a viable basis for early, model-based reasoning about evolving railway signalling systems-of-systems.

Rather than advocating a particular implementation or signalling architecture, the demonstrator illustrates how careful modelling choices can enable clarity, consistency, and analysability at a stage where changes are still feasible and valuable. In this sense, the contribution supports a shift from late validation of detailed designs toward earlier, abstraction-driven reasoning grounded in shared conceptual models.

The contribution of D30.3/D30.4 is the articulation and proof-of-concept validation of a disciplined modelling methodology for early requirements validation in evolving signalling systems-of-systems. It demonstrates that a carefully chosen ontology-driven abstraction can remain both expressive and analysable at realistic scale, thereby enabling earlier and more systematic reasoning before detailed design commitments are made.

## REFERENCES

[1] Europe's Rail. Grant Agreement Project 101102001 – FP2 – R2DATO.

[2] Shift2Rail X2Rail-5. Deliverable D4.1: Moving Block Specification.

[3] Shift2Rail X2Rail-5. Deliverable D10.1: Requirement analysis.

[4] Shift2Rail X2Rail-5. Deliverable D10.3: Configuration Data.

[5] Shift2Rail X2Rail-5. Deliverable D10.4: Verification Report.

[6] Shift2Rail X2Rail-5. Deliverable D10.6: Formal Methods Application.

[7] Shift2Rail X2Rail-5. Deliverable D10.9: Formal Methods Guidebook.

[8] Shift2Rail X2Rail-5. Deliverable D14.2: Proposed Methodologies based on FMs Guidebook.

[9] Shift2Rail X2Rail-2 Deliverable D5.2, Formal Methods Application, Revision 1.6, Nov 3, 2020.

[10] Borälv, A. et al. Holistic Study of Formal Methods and Standardization in Specification, Development, Verification and Validation of Railway Signalling System Software. World Congress on Railway Research (WCRR), 2022.

[11] Borälv, A. (2018). Interlocking Design Automation Using Prover Trident. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds) Formal Methods. FM 2018. LNCS, vol 10951.

[12] Borälv, A. AC/DC and GO: An Ontology-based Approach to Requirements Validation. Tutorial, RSSRail, Nov 2025 (6th Int'l Conf. on Reliability, Safety, and Security of Railway Systems).

[13] HLL: Language Definition. Technical Report, Prover Technology. 2021.⟨hal-03294999⟩.

[14] LCF 2.0: Language Definition. Per Larsson, Olav Bandmann. ⟨hal-03221150⟩.

[15] Behavior Trees in Robotics and AI: An Introduction. Michele Colledanchise and Petter Ögren. ISBN 9781138593732.

[16] Requirements for functions and interfaces in the Alex product, ALEX 17-001. Trafikverket, 2017.

[17] ERA Ontology. Version 3.2.1. European Union Agency for Railways. 2026. Available at: https://data-interop.era.europa.eu/era-vocabulary.

[18] Duggan, P. and Borälv, A. Mathematical proof in an automated environment for railway interlockings. Technical Paper, IRSE News Issue 217, Dec 2015.

# 10 A: ONTOLOGY FRAMEWORK

This appendix complements Chapter 3 by illustrating *how* the ontology is defined and structured in the prototype framework, using selected implementation-level examples.

## 10.1 ONTOLOGY DEFINITION LANGUAGE

The ontology-definition language is realised using generic Python functionality and lightweight conventions, allowing ontology definitions (classes, mixins, and specialisations) to be expressed directly as executable Python code. This approach supports experimentation:

- no separate domain-specific language is required (beyond python itself); and
- integration with tooling is immediate.

Classes, mixins, and specialisations are defined using generic functions that return Python dataclasses, assigned to global variables with the same name as the entities defined. The code snippet below illustrates such definitions in the upper ontology, showing how the core object semantics (nullary, static, dynamic, active) are introduced compositionally using mixins and specialisations (rather than hard-coded inheritance):

- ONTO_SUPER is the top-most class in the upper ontology.
- A class specifies its attributes, mixins, and specialisations using lists.
- Attribute category is an enumeration in Python (e.g., Cat.xxx).
- Satt() is a construct dedicated to attribute specialisation. Multiple specialisations can be expressed in one Satt() instance, and each specialisation is typed (e.g., Spec.xxx).

```
CC, CM, CS  = x.create_type, x.create_mixin, x.create_specialization
DECL_DYN = CM('DECL_DYN',
    doc="Mixin for attributes 'dynamic' and 'active'.", atts=[
        Attr("dynamic", Cat.STAT, AttBool()),
        Attr("active", Cat.DYN, AttBool())])
HAS_NULLARY = CM('HAS_NULLARY',
    doc="Mixin for static attribute 'nullary'.",
    atts=[
        Attr("nullary", Cat.STAT, AttBool())])
ONTO_OBJ = CC('ONTO_OBJ',
    base=ONTO_SUPER,
    mixin=[DECL_DYN, HAS_NULLARY])
DYNAMIC = CS('DYNAMIC',
    doc="Specialization for dynamic object.",
    satts=[
        Satt("dynamic", [(Spec.DEF_CONST, True)])])
STATIC = CS('STATIC',
    doc="Specialization for static object.",
    satts=[
        Satt("dynamic", [(Spec.DEF_CONST, False)]),
        Satt("active", [(Spec.MAKE_STATIC,),
                        (Spec.DEF_FUN, "helper_upper.not_nullary")])])
```

## 10.2 MODULAR COMPONENTS

The prototype framework is used to define classes, mixins, and specialisations, stored on a conceptual creation stack, enabling modular, component-based construction of the ontology (upper and lower).

One usage of these components is to generate graphical depictions of type dependencies.

Figure 32 illustrates one such component, showing the inheritance relation for lower-ontology classes inheriting the node (U_Node) and edge (U_Edge) types from the upper ontology. Class names of the form N4_* denote track-crossing nodes with different passage relations.

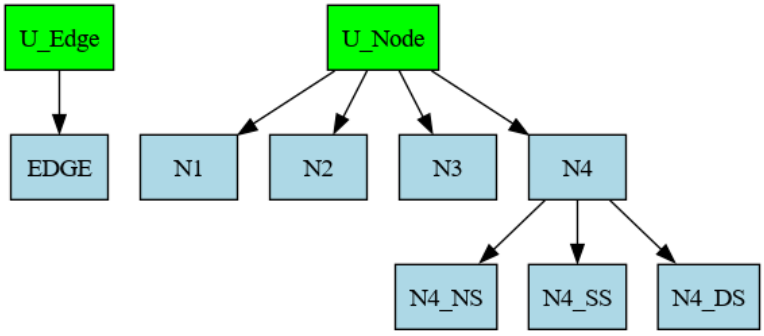



**Figure 12: Lower ontology classes that inherit U_Edge and U_Node**

Figure 13 illustrates another such component, showing types in the lower ontology that inherit area types in the upper ontology (U_S_Area and U_D_Area, for static and dynamic areas, respectively). The lower ontology concepts include Area of Control (for an RBC), radio hole, tunnel, and locking area. The dynamic area is a Trafikverket-specific concept used in the demonstrator.

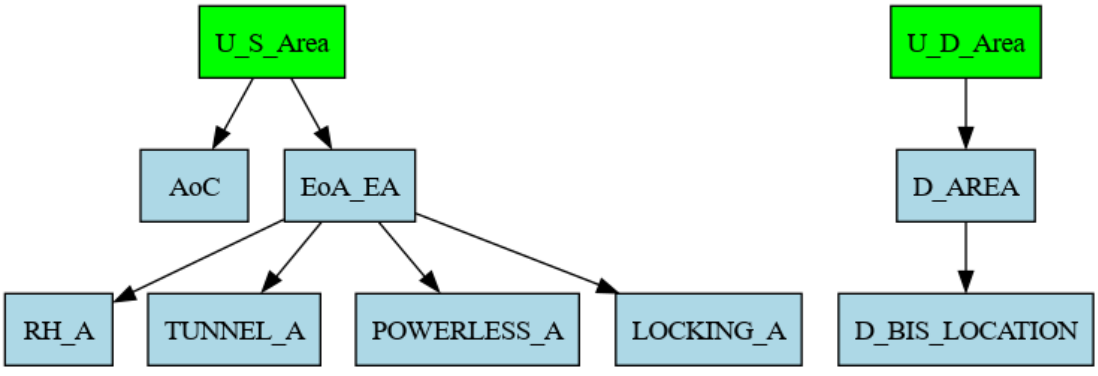



**Figure 13: Lower ontology classes inheriting area types in upper ontology**

In addition to supporting modular documentation and visualisation, the component-based organisation of ontology definitions is also used to support refinement and extension steps. In particular, certain ontology components are generated or refined based on configuration information, for example to introduce message types and interface-related concepts that depend on the chosen system architecture or object-controller interfaces.

In such cases, the ontology is first constructed from its core components and type-checked, after which additional ontology fragments are generated and composed to reflect interface refinements. The combined ontology is then used as the basis for generating an updated object model interface. This incremental construction and regeneration of the object model allows architecture- and interface-dependent concepts to be introduced in a controlled and modular way, without hard-coding such refinements into the core ontology.

## 10.3 Documentation and Other Visualization

The ontology framework generates rudimentary text-based documentation of the object model. In addition, it supports configurable generation of diagrams showing inheritance, mixin and specialisation relationships.

Figure 14 illustrates the classes, mixins, and specialisations on which the directed segment class (DIR_SEGM) depends, illustrating how multiple modelling mechanisms combine to define a single concept (mixins are colored orange, and specializations are colored pink).

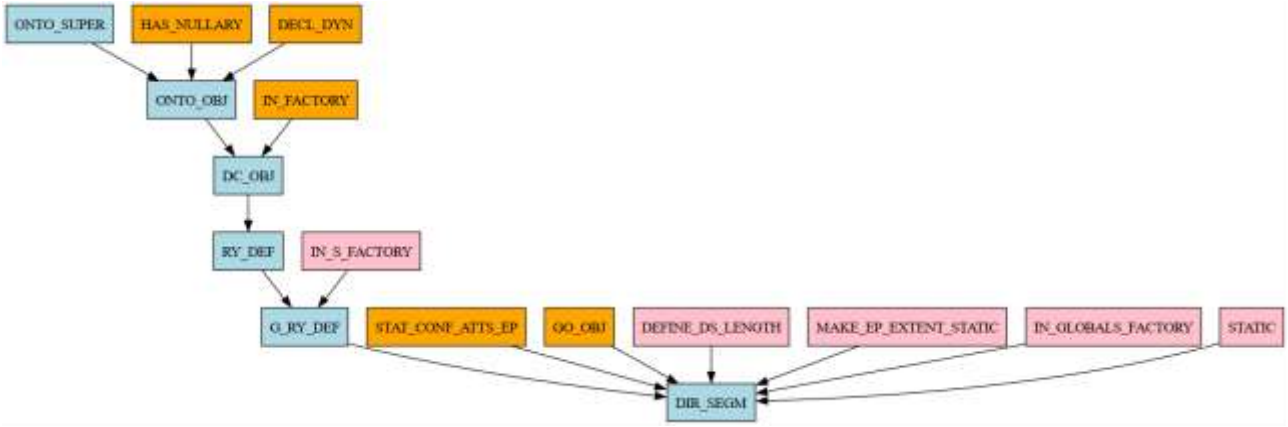



**Figure 14: Classes, mixins, and specialisations on which DIR_SEGM depends**

The following exemplifies a simple text-based documentation of a path extent class, illustrating the types it depends on and its attributes (with type indication), separating defined attributes, configuration-time attributes and run-time attributes.

```
Class: EXT_D_OP_DX
  Doc: Railyard Extent: Dynamic Offset Path with Dynamic Extent.
  Class sorts: ['EXT_D_OP_DX', 'EXT_OP', 'EXT_DEF', 'DC_OBJ',
                'ONTO_OBJ', 'ONTO_SUPER']
  Mixin sorts: ['HAS_X_PARENT', 'HAS_NORMALIZED_PATH_EXTENT',
                'HAS_OP_OFFSETS', 'IN_FACTORY', 'DECL_DYN',
                'HAS_NULLARY']
  Satts sorts: ['DYNAMIC', 'ISA_NON_EMPTY_PATH',
                'X_PARENT_DX_OP', 'HAS_DEFINED_X_LENGTH']
  Defined attributes:
    static  bool 'dynamic' := True
    dynamic  int 'x_length' with range 1..100000000 := helper_upper.x_length
  Configuration-time attributes:
    static  bool 'nullary'
    static  FACTORY 'in_factory'
    static  U_D_OP_DX 'x_parent'
  Run-time attributes:
```

```
dynamic  bool 'active'
dynamic  DIR_SEGM 'x_first'
dynamic  DIR_SEGM 'x_last'
dynamic  set 'x_inner' with signature (DIR_SEGM) with cardinality 0..100
dynamic  int 'x_first_o' with range 0..10000000
dynamic  int 'x_last_o' with range 1..10000000
```

## 10.4 Architecture Concepts in the Upper Ontology

This section illustrates the inheritance structure of Architecture Concepts (AC) defined in the upper ontology. The diagram shows how abstract architectural roles are decomposed into static and dynamic variants, and how scoping concepts (U_Globals, U_Part, U_Container) relate to factories and communication links.

The diagram is intended as a structural reference. It complements the textual description in Chapters 3 and 6 by making explicit the relationships between upper-ontology architectural types.

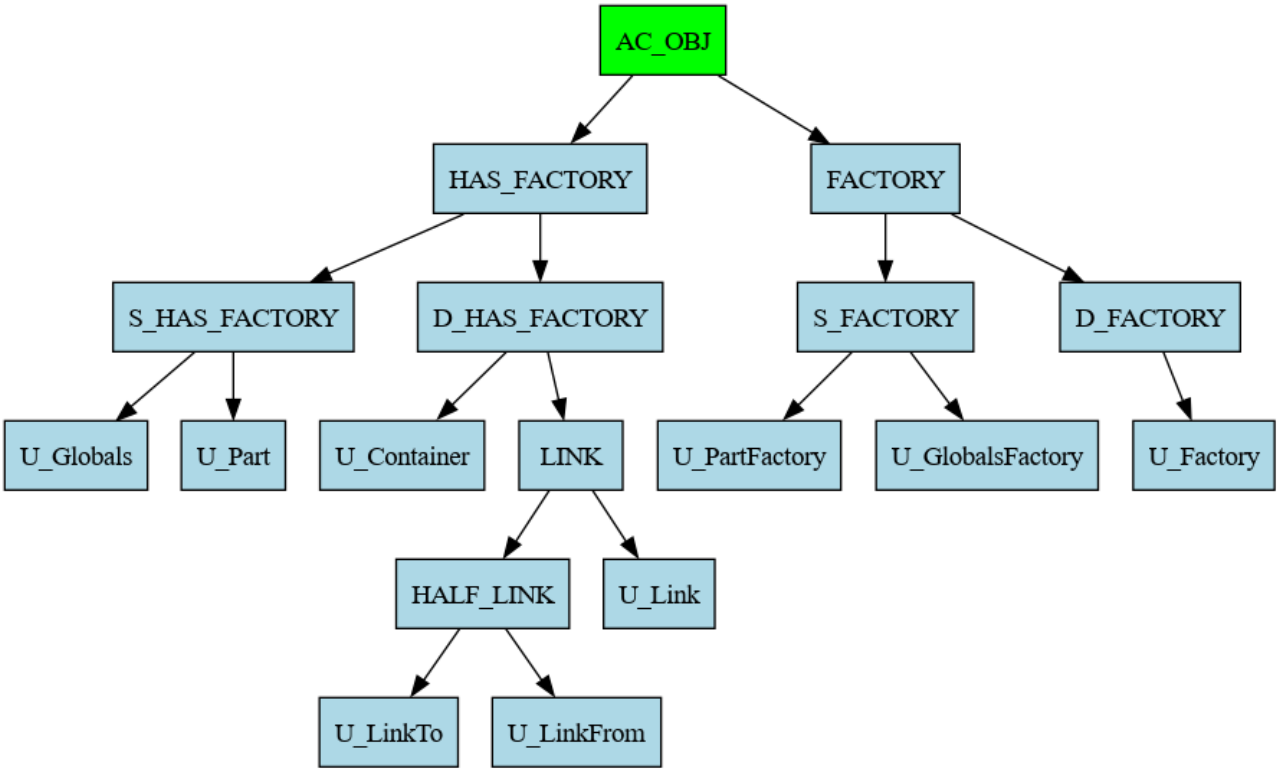



**Figure 15: Inheritance diagram (AC) for upper ontology**

- AC_OBJ is the root for all architectural concepts.
- U_Globals and U_Part represent static architectural scopes.
- Factories exist in both global and part-local variants.
- Containers are dynamic architectural entities allocated to parts.
- Communication is modelled using regular links (U_Link) and half links (U_LinkTo, U_LinkFrom).

# 11 B: OBJECT MODEL INTERFACES

This appendix provides **example-based illustrations of the generated object model interfaces**, focusing on how ontology-defined concepts are realised as concrete modelling interfaces in both **Python** (for executable validation) and **HLL** (for formal verification).

The purpose of this appendix is **not** to document the complete generated interfaces, but to illustrate:

- how object types and attributes appear in each interface,
- how static and dynamic attributes are separated,
- and how the same modelling semantics are preserved across executable and formal representations.

A convention used is that a global function exists to read each attribute. In HLL, these global functions constitute the primary interface for formulating properties (in terms of the object model). In Python, the global functions are optional, since object slots provide a more direct means to read (and write) attributes (see Section 11.1.3).

The framework ensures generated identifiers are valid in both Python and HLL, applying systematic renaming where necessary (for example, normalising characters originating from RDF data).

All code excerpts shown in this appendix are **exactly as generated** by the demonstrator, without manual modification.

## 11.1 OBJECT MODEL INTERFACE (PYTHON)

This section illustrates how object types and attributes defined in the ontology are realised in the executable object model interface generated in Python.

### 11.1.1 Example: Object Type AB (Authority Base)

The following Python class represents the lower-ontology object type **AB (Authority Base)**. An instance of this class corresponds to a concrete Authority Base object in an instantiated model.

If the parameter nullary_obj is set to True, the instance represents the **nullary object** for this type.

```
class ObjAb(Obj):
    onto_name = 'AB'
    __slots__ = (
        "ab_orientation", "ab_subtype", "active",
        "dynamic", "in_factory", "my_tlp",
        "path_length", "ry_extent", "train", "z_ap"
    )
    def __init__(self, name, nullary_obj=False):
        super().__init__(name, static_obj=False, nullary_obj=nullary_obj)
        self.nullary = x_att(self, Cat.STAT, Typ.BOOL, Def.CONST,
                             value=nullary_obj)
        self.dynamic = x_att(self, Cat.STAT, Typ.BOOL, Def.CONST,
                             value=True)
        self.active = x_att(self, Cat.DYN, Typ.BOOL, Def.NA)
        self.ab_orientation = x_att(self, Cat.STAT_INC, Typ.ENUM, Def.NA,
                                    ranges=['AB_FWD', 'AB_BWD'])
        self.ab_subtype = x_att(self, Cat.DYN, Typ.ENUM, Def.NA,
                                ranges=['AB_RP', 'AB_SR', 'AB_RV'])
```

```
        self.path_length = x_att(self, Cat.DYN, Typ.INT, Def.FUN,
                                 value=helper_upper.path_length)
        self.ry_extent = x_att(self, Cat.STAT_INC, Typ.OBJ, Def.NA,
                               sign=[y_sort('EXT_D_OP_DX')])
        self.train = x_att(self, Cat.STAT_INC, Typ.OBJ, Def.NA,
                           sign=[y_sort('CUT')])
        self.z_ap = x_att(self, Cat.STAT, Typ.OBJ, Def.NA,
                          sign=[y_sort('AP')])
```

**Key observations:**

- Each attribute is realised as a structured attribute object with explicit category, type, and definition mode.
- Static, dynamic, incarnation, and dormant attributes are treated uniformly at the interface level.
- Function-defined attributes (such as path_length) are bound to generated helper functions.
- The class uses Python's __slots__ mechanism to enforce a fixed attribute set and prevent accidental attribute creation.

This object model interface provides the primary executable API through which behavioural models and test cases interact with instantiated objects, enforcing typing, lifecycle, and attribute semantics defined by the ontology.

### 11.1.2 Example: Object Type EXT_D_OP_DX (path extent)

The following Python class represents the upper-ontology object type EXT_D_OP_DX **(path extent)**. Each AB path object has an associated instance of this class.

```
class ObjExt_D_Op_Dx(Obj):
    onto_name = 'EXT_D_OP_DX'
    __slots__ = (
        "active", "dynamic", "in_factory", "x_first", "x_first_o",
        "x_inner", "x_last", "x_last_o", "x_length", "x_parent")

    def __init__(self, name, nullary_obj=False):
        super().__init__(name, static_obj=False, nullary_obj=nullary_obj)
        self.nullary = x_att(self, Cat.STAT, Typ.BOOL, Def.CONST,
                             value=nullary_obj)
        self.dynamic = x_att(self, Cat.STAT, Typ.BOOL, Def.CONST,
            value=True)
        self.active = x_att(self, Cat.DYN, Typ.BOOL, Def.NA)
        self.in_factory = x_att(self, Cat.STAT, Typ.SORT, Def.NA,
            sort='FACTORY')
        self.x_first = x_att(self, Cat.DYN, Typ.SORT, Def.NA,
            sort='DIR_SEGM')
        self.x_first_o = x_att(self, Cat.DYN, Typ.INT, Def.NA,
            ranges=[(0, 10000000)])
        self.x_inner = x_att(self, Cat.DYN, Typ.SET, Def.NA,
            cardin=[(0, 100)],
            sign=[y_sort('DIR_SEGM')])
        self.x_last = x_att(self, Cat.DYN, Typ.SORT, Def.NA,
            sort='DIR_SEGM')
```

```
        self.x_last_o = x_att(self, Cat.DYN, Typ.INT, Def.NA,
            ranges=[(1, 10000000)])
        self.x_length = x_att(self, Cat.DYN, Typ.INT, Def.FUN,
            value=helper_upper.x_length,
            ranges=[(1, 100000000)])
        self.x_parent = x_att(self, Cat.STAT, Typ.SORT, Def.NA,
            sort='U_D_OP_DX')
```

### 11.1.3 Example: Instantiate and Configure an Object

The following code illustrates object instantiation and writing of static attributes (slots) using the Python interface (for one of the RDF data examples from Trafikverket). This example is representative; concrete identifiers and topology details vary between infrastructure data sets.

In this example, the object is a track-circuit area (lower ontology type name TTD_A, generated Python class name ObjTtd_A_Tc). A dictionary n2o mapping names to objects is used. Since configured attributes area_nodes and area_edges are static, writing them requires override=True.

```
name = '9e57dbd4-7283-4536-990a-63ac34d2f919'
tc = ObjTtd_A_Tc(name)
n2o[name] = tc
e_names = ['(12312,120):2-(12432,240):0',
           '(12312,120):1-(12492,120):1',
           '(12492,120):0-(12596,120):1',
           '(11910,120):0-(12312,120):0',
           '(12432,240):2-(12432,343):1',
           '(12596,120):0-(12672,120):0']
n_names = ['(12672,120)', '(12492,120)',
           '(12432,240)', '(12596,120)',
           '(12432,343)', '(12312,120)',
           '(11910,120)']
nodes = set([(n2o[n],) for n in n_names])
getattr(tc, 'area_nodes').write(nodes, override=True)
edges = set([(n2o[e],) for e in e_names])
getattr(tc, 'area_edges').write(edges, override=True)
```

## 11.2 Object Model Interface (HLL)

This section illustrates how object types and their attributes are realised in the HLL representation of the object model. It mirrors the structure of the representation of the object model in Python, while respecting the constraints of a many-sorted first-order logic setting.

### 11.2.1 Static and Dynamic Attribute Structures

For each leaf object type, the formal model defines:

- one structure holding **static attributes**, and
- one structure holding **dynamic attributes**.

**Example: AB (Authority Base)**

```
struct {
    in_factory : FACTORY,
    my_tlp     : TLP,
```

```
    nullary    : bool,
    ry_extent  : EXT_D_OP_DX,
    train      : CUT,
    z_ap       : AP
} stat_AB;

struct {
    ab_orientation : AB_ORIENTATION /* enum_t */,
    ab_subtype     : AB_SUBTYPE /* enum_t */,
    active         : bool
} dyn_AB;
```

**Key observations:**

- Static and dynamic attributes are represented explicitly and separately.
- Dormant attributes (e.g. z_ap) appear in the static structure, reflecting their special semantics.
- Enumerated attributes are represented using sorts (see Appendix C).

### 11.2.2 Example: Path Extent Definition (EXT_D_OP_DX)

The following example illustrates how **path extent definition objects** are represented in the formal model.

```
struct {
    in_factory : FACTORY,
    nullary    : bool,
    x_parent   : U_D_OP_DX
} stat_EXT_D_OP_DX;

struct {
    active    : bool,
    x_first   : DIR_SEGM,
    x_first_o : int [0, 10000000],
    x_inner   : cf__DIR_SEGM,
    x_last    : DIR_SEGM,
    x_last_o  : int [1, 10000000]
} dyn_EXT_D_OP_DX;
```

**Key observations:**

- The internal set-based representation of path extent (x_inner) is a **characteristic function**.
- Boundary segments and offsets are represented explicitly and remain fully typed.
- This structure supports uniform reasoning about fixed-block and moving-block paths.

### 11.2.3 Enum Type

The following illustrates encoding of an enumerated type using sorts in HLL. In this example, the enumerated type is called AP_SUBTYPE, with possible values MA, SR_DIST and RV_DIST.

```
  sort AP_SUBTYPE /* enum_t */;
  sort 'leaf_enum_sort(MA)';
  sort 'leaf_enum_sort(SR_DIST)';
```

```
  sort 'leaf_enum_sort(RV_DIST)';
  sort 'leaf_enum_sort(MA)', 'leaf_enum_sort(SR_DIST)', 'leaf_enum_sort(RV_DIST)'
< AP_SUBTYPE /* enum_t */;
  sort { MA } < 'leaf_enum_sort(MA)';
  sort { SR_DIST } < 'leaf_enum_sort(SR_DIST)';
  sort { RV_DIST } < 'leaf_enum_sort(RV_DIST)';
```

## 11.3 ATTRIBUTE ACCESS

In the generated Python interface, attributes are accessed using the slot mechanism (read/write).

The generated HLL interface includes, for each attribute, a **global access function** to read the attribute value. The definition accounts for specialisation, including constants, function-based definitions, and direct field reads from per-type attribute structures. The remainder of this section provides representative examples with different kinds of specialisations.

### 11.3.1 Attribute Specialised to Constant

```
// DECL_DYN : used by ['ONTO_OBJ']
// DECL_DYN : introduces 2 attributes
Declarations:
  bool dynamic(DECL_DYN);     //static
  bool active(DECL_DYN);      //dynamic
Definitions:
  // Specialized by sorts: ['STATIC', 'DYNAMIC']
  // Return type: static  bool 'dynamic'
  dynamic(obj) :=
  (obj
      | STATIC _ => False
      | DYNAMIC _ => True);
```

### 11.3.2 Attribute Specialised to Function

```
// HAS_PATH_LENGTH : used by ['EDGE_PATH', 'OFFSET_PATH']
// HAS_PATH_LENGTH : introduces 1 attributes
Declarations:
  int path_length(HAS_PATH_LENGTH); //dynamic
Definitions:
  // Specialized by sorts: ['PATH_LENGTH_DEF']
  // Return type: dynamic  int 'path_length' with range 0..None
  path_length(obj) :=
  (obj
      | PATH_LENGTH_DEF o => helper_upper.path_length(o));
```

### 11.3.3 Attribute of Type Object

```
// HAS_LINK_FROM : used by ['U_Link', 'U_LinkFrom']
// HAS_LINK_FROM : introduces 1 attributes
Declarations:
  U_Container link_from(HAS_LINK_FROM);   //static
Definitions:
  // Specialized by sorts: []
  // Return type: static  U_Container 'link_from'
  link_from(obj) :=
  (obj
```

```
      | RBC2RBC o1 => struct_s_RBC2RBC(o1).link_from
      | RBC2TMS o1 => struct_s_RBC2TMS(o1).link_from
      | TMS2RBC o1 => struct_s_TMS2RBC(o1).link_from
      | RBC2OC o1 => struct_s_RBC2OC(o1).link_from
      | OC2RBC o1 => struct_s_OC2RBC(o1).link_from
      | RBC2TRAIN_OUT o1 => struct_s_RBC2TRAIN_OUT(o1).link_from
      | TRAIN2RBC_OUT o1 => struct_s_TRAIN2RBC_OUT(o1).link_from);
```

### 11.3.4 Attribute of Type Set

The following illustrates the global function to read the static attribute node_passage, which is a set of node-connector index pairs (tuples of length 2).

```
// U_Node : used by ['N1', 'N2', 'N3', 'N4']
// U_Node : introduces 2 attributes
Declarations:
  cf__int_0_3_int_0_3 node_passage(U_Node);    //static
  int node_degree(U_Node); //static
Definitions:
  // Specialized by sorts: ['DEGREE_1', 'DEGREE_2', 'DEGREE_3', 'DEGREE_4']
  // Return type: static  set 'node_passage' with signature (int(0,3), int(0,3)) with
cardinality 0..None
  node_passage(obj) :=
  (obj
      | DEGREE_1 _ => lambda(int [0, 3],int [0, 3]):(i0,i1) := (i0,i1 | _,_ => False)
      | DEGREE_2 _ => lambda(int [0, 3],int [0, 3]):(i0,i1) := (i0,i1
         | 0,1 => True
         | 1,0 => True
         | _,_ => False)
      | DEGREE_3 _ => lambda(int [0, 3],int [0, 3]):(i0,i1) := (i0,i1
         | 0,1 => True
         | 0,2 => True
         | 1,0 => True
         | 2,0 => True
         | _,_ => False)
      | DEGREE_4 o1 => (o1
        | NO_SLIP _ => lambda(int [0, 3],int [0, 3]):(i0,i1) := (i0,i1
           | 0,2 => True
           | 1,3 => True
           | 2,0 => True
           | 3,1 => True
           | _,_ => False)
        | SINGLE_SLIP _ => lambda(int [0, 3],int [0, 3]):(i0,i1) := (i0,i1
           | 0,1 => True
           | 0,2 => True
           | 1,0 => True
           | 1,3 => True
           | 2,0 => True
           | 3,1 => True
           | _,_ => False)
        | DOUBLE_SLIP _ => lambda(int [0, 3],int [0, 3]):(i0,i1) := (i0,i1
           | 0,1 => True
           | 0,2 => True
           | 1,0 => True
           | 1,3 => True
           | 2,0 => True
           | 2,3 => True
           | 3,1 => True
```

```
          | 3,2 => True
          | _,_ => False)));
```

## 11.4 Attribute Specialisation by Functions

This section illustrates, through a concrete example, how function-based attribute specialisation is realised consistently in the generated interfaces in Python and HLL. Although realised in different languages, both interfaces preserve the same modelling semantics and separation of concerns.

The body of the Python and HLL functions are supplied as helper functions (they are not generated from the ontology structure).

The example specialises the attribute path_length, which computes the length of a path. Since paths refer to a separate extent definition object, the computation first retrieves the associated extent and then derives the length from its directed segments and offsets.

### 11.4.1 Attribute Specialisation in Python

In the generated Python interface, function-based attribute specialisation is realised using a regular Python function. The function takes a path object as input, retrieves its extent definition object via ry_extent, and computes the path length accordingly. The function is located in the module helper_upper.py.

```
def path_length(o1: 'Obj') -> int:
    o = o1.ry_extent.read()
    if o.x_first.read() == o.x_last.read():
        assert len(o.x_inner.read()) == 0
        return (
            o.x_last_o.read() -
            o.x_first_o.read())
    return (
        sum(ds_length(ds) for (ds,) in
            o.x_inner.read()) +
        ds_length(o.x_first.read()) -
        o.x_first_o.read() +
        o.x_last_o.read())
```

### 11.4.2 Attribute Specialisation in HLL

In the generated HLL interface, function-based attribute specialisation is realised using HLL functions. The attribute path_length is declared as a function of return type int over objects of sort HAS_PATH_LENGTH (a mixin sort introducing the attribute).

Each function definition in the module helper_upper.py has a corresponding named field in a global structure called helper_upper, and each field is defined by a HLL function.

```
Definitions:
  helper_upper := {
    ...,
    ns_defs::ns_path_length,
    ... };

Namespaces: ns_defs {
```

```
  ns_path_length(obj) :=
    ns_x_length(ry_extent(obj));

  ns_x_length(obj) :=
    if x_first(obj) = x_last(obj)
    then x_last_o(obj) - x_first_o(obj)
    else SUM ds:DIR_SEGM (
       if x_inner(obj)(ds) then
       ds_length(ds) else 0
    ) + ds_length(x_first(obj)) -
      x_first_o(obj) +
      x_last_o(obj);
}
```

The following can be noted:

- The computation is split into two functions. The first function (ns_path_length) retrieves the extent definition object and delegates the computation to a second function. This mirrors the separation between path objects and extent definition objects in the object model.
- The HLL operator SUM aggregates the lengths of directed segments belonging to the interior of the path. Membership is tested using the characteristic-function representation of the set-valued attribute x_inner, written as x_inner(obj)(ds).
- Although expressed in a declarative, synchronous formal language, the function mirrors the same conceptual computation performed in the Python version.

### 11.4.3 Discussion

This example illustrates how function-based attribute specialisation is handled consistently across the executable and formal object model interfaces. While Python and HLL impose different syntactic and semantic constraints, the generated interfaces preserve the same modelling assumptions, attribute structure, and separation of concerns.

In particular:

- attribute specialisation logic is defined at the modelling level and realised coherently in both languages;
- the separation between paths and extent definition objects is preserved; and
- analysis-friendly set encodings (characteristic functions) can be exploited naturally in the formal model without obscuring domain semantics.

This correspondence supports confidence that executable validation and formal verification are applied to the same conceptual model, even though they are realised in different technical forms.

## 11.5 SUMMARY

This appendix demonstrates that:

- the **same conceptual object model** is realised coherently in both Python and HLL,
- executable and formal interfaces expose **equivalent object types and attributes**, and
- differences between the interfaces are confined to representation details required by the respective languages.

This structural correspondence underpins confidence that executable validation and formal verification operate over the **same modelling abstraction**, despite being realised in different technical forms.

# 12 C: INSTANTIATED OBJECT DATA IN HLL

This appendix illustrates how **instantiated object data** is represented in the formal HLL model generated by the demonstrator. Whereas Appendix B focuses on the *generic object model interface* (types, attributes, and access functions), this appendix shows how **concrete object instances and their attribute values** are encoded in HLL after instantiation, forming the concrete basis for formal analysis and verification.

The examples are intentionally selective. Their purpose is to make the mapping from instantiated objects to formal representations concrete and understandable, not to document all generated data structures.

All HLL excerpts shown in this appendix are **automatically generated** from the instantiated object model and are presented without manual modification.

## 12.1 Object Instances as Sort Instances

In the formal model, each instantiated object corresponds to:

- a **sort element** (representing the object identity), and
- two associated structures holding its **static** and **dynamic** attribute values.

**Example: Instances of AB (Authority Base)**

The following excerpt illustrates how object instances of type AB are declared as elements of the sort AB.

```
Types:
  sort {
        'ObjAb::0_bwd',
        'ObjAb::0_fwd',
        'ObjAb::1_bwd',
        'ObjAb::1_fwd',
        ...
       } < AB;
  sort {'!_AB_null'} < AB;
```

Each identifier (e.g. ObjAb::0_fwd) denotes a distinct Authority Base object. In this example:

- each train is associated with two AB objects (forward and backward orientation), reflecting the train-centric modelling conventions used in the demonstrator, and
- the nullary AB object is listed separately.

## 12.2 Mapping Object Instances to Attribute Structures

For each object instance, the formal model defines:

- one structure containing **static attribute values**, and
- one structure containing **dynamic attribute values**.

Access functions map each sort element to its corresponding structures.

### 12.2.1 Static Attribute Structures

The following example shows the static attributes for one AB object.

```
// Sort AB, object ObjAb::0_fwd
'ss_ObjAb::0_fwd' := {
    AB_FWD                  /* ab_orientation */,
    'Factory(RBC::0)'       /* in_factory */,
    'ObjTlp::0'             /* my_tlp */,
    False                   /* nullary */,
    'ry_extent(ObjAb::0_fwd)' /* ry_extent */,
    'ObjCut::0'             /* train */,
    'ObjAp::0_fwd'          /* z_ap */
};
```

**Key observations:**

- Static attributes directly reflect values assigned during instantiation and remain unchanged during execution.
- References to other objects (e.g. train, ry_extent) are explicit, using object identifiers.
- Dormant attributes (such as z_ap) appear here as static associations.

### 12.2.2 Static Attributes for Path Extent Objects

Path extent definition objects (EXT_D_OP_DX) are instantiated separately and are associated to their parent path objects (x_parent). This structure links the extent object back to its owning path object and places it under factory management.

```
// Sort EXT_D_OP_DX, object ry_extent(ObjAb::0_bwd)
'ss_ry_extent(ObjAb::0_bwd)' := {
    'Factory(RBC::0)'   /* in_factory */,
    False               /* nullary */,
    'ObjAb::0_bwd'      /* x_parent */
};
```

## 12.3 NULLARY OBJECTS

For each leaf type in the ontology, the demonstrator generates a **nullary object,** to provide a type-safe representation of absence, and to avoid the need for null references or union types.

**Example: Nullary Factory Object**

The following excerpt illustrates the dynamic attributes of the nullary factory object.

The attribute sort2objs uses a generated ‘fresh’ sort (‘leaf_sort(ONTO_OBJ)’) corresponding to the top-level ontology type. This allows the factory to maintain mappings over all object types in a uniform and type-safe way.

```
// Sort U_Factory, nullary object !_U_Factory_null
'ds_!_U_Factory_null' := {
    False /* active */,
    lambda(DYNAMIC):(j0) := (j0 | _ => False) /* pool_activating */,
    lambda(DYNAMIC):(j0) := (j0 | _ => False) /* pool_active */,
    lambda(DYNAMIC):(j0) := (j0 | _ => False) /* pool_inactive */,
    lambda(DYNAMIC):(j0) := (j0 | _ => False) /* pool_pending_activate */,
```

```
    lambda(DYNAMIC):(j0) := (j0 | _ => False) /* pool_pending_deactivate */,
    lambda(STATIC):(j0) := (j0 | _ => False) /* pool_static */,
    lambda('leaf_sort(ONTO_OBJ)'):(i0) :=
        (i0 | _ => lambda(ONTO_OBJ):(j0) := (j0 | _ => False))
        /* sort2objs */
};
```

**Key observations:**

- All attributes of nullary objects are assigned well-defined “empty” values.
- Set-valued attributes are represented as characteristic functions returning False.
- The nullary object is never active and does not participate in system behaviour.

## 12.4 SUMMARY

This appendix illustrates how:

- instantiated objects are represented as sort elements in the formal model,
- attribute values are captured explicitly in static and dynamic structures, and
- nullary objects provide a uniform, type-safe representation of absence.

These instantiated representations are the concrete data over which properties are expressed in HLL and execution traces are analysed, providing a common reference point between executable validation and formal analysis.

Together with Appendix B, these examples demonstrate how the ontology-driven object model supports a **clean and traceable mapping** from conceptual modelling abstractions to executable and formal representations, enabling systematic validation and verification.

# 13 D: CONSISTENCY CHECKING AND PROOF OBLIGATIONS

This appendix provides illustrative examples of **generic consistency requirements** and **proof obligations** generated by the demonstrator and expressed in terms of the object model. These examples complement the discussion in Chapter 4 and Chapter 8 by showing how modelling principles embedded in the ontology and object model are enforced through executable and formal validation.

The intent of this appendix is **not** to provide a complete catalogue of all generated properties, but to illustrate:

- the kinds of consistency principles that are enforced,
- how these principles are expressed in terms of the object model interface, and
- how they support disciplined modelling and early validation.

## 13.1 Scope of Consistency Requirements

The consistency requirements illustrated here fall into three broad categories:

1. **Ontology-level principles**, which apply universally to all models built using the framework.
2. **Object lifecycle and existence properties**, ensuring coherent treatment of static, dynamic, and nullary objects.
3. **Attribute consistency properties**, capturing typing and lifecycle-related constraints.

These requirements are independent of any specific behavioural model. They apply equally to executable instantiations and to formal verification.

## 13.2 Ontology-Level Consistency Principles

The ontology embeds a small set of foundational principles that apply to all object types and instances. These principles ensure that object existence, nullary semantics, and classification into static and dynamic objects are handled consistently.

The following examples illustrate how such principles are expressed in HLL.

### 13.2.1 Nullary Objects Are Never Active

This property states that nullary objects—used to represent absence—are never considered active.

```
ALL x:ONTO_OBJ (nullary(x) -> ~active(x));
```

### 13.2.2 Static Objects Are Always Active

Static objects represent configuration elements that exist permanently. This requirement expresses that any static object that is not nullary is always active.

```
ALL x:STATIC (~nullary(x) -> active(x));
```

### 13.2.3 Active Dynamic Objects Are Non-Nullary

This property expresses that any dynamic object that is active represents a real, existing entity and not a nullary placeholder.

```
ALL x:DYNAMIC (active(x) -> ~nullary(x));
```

### 13.2.4 Global Objects Are Static

Global Objects (track topology primitives) are static by definition. This requirement expresses that classification.

```
ALL x:GO_OBJ ((x:STATIC));
```

## 13.3 Attribute Consistency and Lifecycle Constraints

In addition to ontology-level principles, the framework enforces consistency constraints related to attributes and object lifecycle.

Some constraints are:

- **checked dynamically** during executable validation (e.g. Python runtime checks), and
- **assumed or verified formally** in the HLL model, depending on expressivity and feasibility.

Examples include:

- Attribute values must conform to their declared type and range.
- Static attributes must not change during execution.
- Active objects must not reference inactive objects.
- Incarnation attributes must remain constant while the parent object is active.

Where such constraints cannot be enforced purely through local checks, the demonstrator generates **proof obligations** that are discharged using formal verification.

## 13.4 Enforcement Strategy: Python vs HLL

The generic ontology-level requirements identified above are handled in the demonstrator by a combination of:

- **run-time checks** in the executable object model interface in Python; and
- **statically verified proof obligations (POs)** in the formal HLL model.

This section summarises how each requirement is treated, and to what extent it is enforced or assumed in the demonstrator.

### 13.4.1 Attribute type and range conformance

*Requirement:* The value of an attribute conforms to its specified type and range.

**Python (run-time):**

This requirement is enforced generically in the Python object model API:

- Attribute structures carry their declared category, type, and range (for example, integer intervals or enumeration domains).
- Whenever a value is written or updated, the generic write operation checks that:
    - the value has the correct Python representation (e.g. object, integer, Boolean, set), and

- integer and enumeration values lie within the declared ranges.

Violations raise an exception at the point of the write operation. These checks apply both when dynamic objects are created/deleted and when attribute values are updated during execution.

**HLL (static model):**

The corresponding well-typedness is **not re-expressed** as explicit proof obligations in HLL. Instead, type and range conformance is treated as a modelling assumption in the formal model:

- the HLL model is generated only from an ontology and object model that have passed the Python-level type checks; and
- attribute domains in HLL (sorts and ranges) reflect those declarations.

In other words, Python enforces type/range conformance dynamically; HLL relies on this enforcement and does not re-check it symbolically.

### 13.4.2 Nullary objects and static vs dynamic objects

*Requirements:*

1. A nullary object is not active.
2. Each object is either static or dynamic.

These requirements are naturally expressed and verified at the HLL level as generic properties over sorts such as ONTO_OBJ, STATIC, and DYNAMIC. For example:

```
// A nullary object is not active.
ALL x:ONTO_OBJ (nullary(x) -> ~active(x));

// A static object that is not a nullary object is active.
ALL x:STATIC (~nullary(x) -> active(x));

// A dynamic object that is active is not a nullary object.
ALL x:DYNAMIC (active(x) -> ~nullary(x));
```

Successful verification of such POs in HLL provides evidence that the ontology-level constraints on nullary, static, and dynamic objects hold for all instances in the formal model.

At the Python level, the corresponding discipline is enforced by construction (through the use of the nullary_obj flag and lifecycle handling of static and dynamic objects), but the most precise formulation of these requirements lives in the HLL model as global properties.

### 13.4.3 No references from active objects to inactive objects

**Requirement:** An active object does not reference an inactive object.

**Python (run-time):**

In the executable object model, activation is performed in batches via factories. Dynamic objects that are candidates for activation are first placed in a dedicated pool of objects pending activation. A single activation step then moves this entire batch from “pending” to “active”.

Generic checks applied during activation and attribute updates enforce the following discipline:

- when a batch of pending objects is activated, any attribute that is required to reference an existing object may point only to:
    - objects that are already active, or
    - other objects in the same pending-activation batch (which will become active in the same step);
- attempts to establish references from active objects to objects that are neither active nor pending activation are rejected (exception is raised).

Similarly, when attributes of already active objects are updated, the generic write operation prevents writing references to objects that are inactive and not scheduled for activation. Violations raise exceptions, preventing inconsistent configurations from being used in execution.

This batch-based discipline allows mutually dependent objects to be co-activated while still ruling out references to objects that are not part of the current configuration state.

**HLL (static model):**

In the HLL model, this requirement corresponds to a state-level property: in **any reachable state** of the formal model, if an object is active, then all attributes that are required to reference existing objects must reference objects that are also active in that state.

Since HLL cannot quantify over attribute symbols, this requirement is realised as a family of attribute-specific proof obligations. The ontology framework provides optional generic functionality that:

1. analyses the ontology and object model to identify attributes whose values may reference dynamic objects, and
2. generates a dedicated HLL requirement for each such attribute.

The following are example properties, expressing – for a specific attribute – that in any state where the parent object is active, the referenced object is also active.

```
// If a Train Location Path (TLP) is active,
// its associated train is active.
ALL x:TLP (active(x) -> active(train(x)));

// If an Authority Base (AB) path is active, its extent
// definition object (attribute ry_extent) is active.
ALL x:AB (active(x) -> active(ry_extent(x)));
```

### 13.4.4 Static attributes remain unchanged over time

*Requirement:* A static attribute remains unchanged over time.

**Python (run-time):**

The generic attribute write operation distinguishes between static and dynamic attributes:

- For **static** attributes, attempting to write a new value without using override=True raises an exception.
- The override mechanism is used deliberately during configuration and instantiation (before execution) to assign initial values to static attributes.

Once execution starts, static attributes are effectively read-only at the Python level, and accidental modification is prevented by the generic API.

**HLL (static model):**

The same requirement can be expressed as a temporal property over the current state and the next state. The ontology framework can:

1. identify static attributes for which this requirement is relevant; and
2. generate a proof obligation for each such case.

For example, for the static attribute node_degree on nodes of sort N1:

```
ALL x:N1 (node_degree(x) = X(node_degree(x)));
```

*Interpretation:* For nodes of sort N1, the current and next values of the attribute node_degree are equal.

This property also applies to nullary objects; in the generated formal model, their static attributes are initialised to nullary values and these shall not change, so the requirement holds uniformly.

Successful verification of these POs in HLL establishes that no behavioural model modifies the selected static attributes, consistent with their intended semantics.

### 13.4.5 Incarnation attributes remain constant while active

*Requirement:* An incarnation attribute remains constant while its parent object is active.

Incarnation attributes are used to distinguish different "incarnations" of a dynamic object (for example, successive uses of the same object slot for different trains). Intuitively, such an attribute may change *between* incarnations, but should remain stable within a single active period.

The ontology framework identifies incarnation attributes and, for each such attribute, generates a proof obligation that captures this constraint. A typical pattern is:

```
ALL x:CU_TRAIN (
  active(x) & X(active(x)) ->
    cut_train_id(x) = X(cut_train_id(x)));
```

*Interpretation:* For a CU_TRAIN object (a dynamic object representing a train managed by an RBC) that is active in the current state and remains active in the next state, the train identifier cut_train_id is unchanged.

In other words, changes to the incarnation attribute are only allowed across transitions where the object is deactivated or re-activated, not while it continuously remains active.

### 13.4.6 Summary

In summary, the demonstrator realises ontology-level requirements through a deliberate division of responsibility:

- **Python** enforces local, operational constraints at runtime, including type and range conformance, prevention of invalid references, and protection of static attributes.
- **HLL** expresses and verifies global properties that cannot be enforced locally, including object existence semantics, lifecycle invariants, and attribute stability conditions.

The ontology framework bridges these two levels by analysing the ontology and object model structure and generating the required runtime checks and formal proof obligations. This hybrid strategy allows expressive modelling while maintaining disciplined and scalable validation.

## 13.5 Proof Obligations Generated from Attribute Specialisation

Some consistency constraints arise from **attribute specialisation**, particularly when attributes are:

- restricted to subtypes,
- specialised from dynamic to static or incarnation categories, or
- specialised to constant or function-based definitions.

In these cases, the demonstrator generates proof obligations that assert the correctness of the specialisation relative to the ontology's type hierarchy.

**Example: Subtype Consistency**

The following proof obligation is an generated example that captures the requirement that the specialised extent-definition type EXT_D_OP_DX is a subtype of the generic extent-definition type EXT_DEF declared in the upper ontology. Successful verification demonstrates that the specialisation is type-consistent.

```
/*
Class: HAS_PATH_EXTENT_D_OP_DX
Specialization sort (used by class): U_D_OP_DX
Attribute (that is specialized): ry_extent
Specialization type: RESTR_OBJ_TYPE
PO: EXT_D_OP_DX shall be a subtype of EXT_DEF.
*/
  ALL a:EXT_D_OP_DX (a:EXT_DEF);
```

## 13.6 Example Properties for Static Data

The following exemplify basic consistency and correctness properties for directed segments.

```
// All N2 nodes in a DIR_SEGM have offset value > 0
ALL ds:DIR_SEGM, n2:N2 (
  N2_nodes(ds)(n2) ->
    N2_offset(ds)(n2) > 0);

// If a DIR_SEGM has exactly one edge, then it contains no N2 node
ALL ds:DIR_SEGM (
  if (SUM e:U_Edge (if edges_in_path(ds)(e) then 1 else 0) == 1)
  then ~SOME n2:N2 (N2_nodes(ds)(n2))
  else TRUE);

// The number of edges in a DIR_SEGM is precisely one more than the
// number of nodes of sort N2 in the DIR_SEGM
ALL ds:DIR_SEGM (
  SUM e:U_Edge (if edges_in_path(ds)(e) then 1 else 0) ==
  SUM n2:N2 (if N2_nodes(ds)(n2) then 1 else 0) + 1);
```

## 13.7 SUMMARY

This appendix has illustrated how generic modelling principles embedded in the ontology are realised through a combination of runtime enforcement and formal verification. By separating local operational checks from global proof obligations, the demonstrator achieves disciplined validation without over-constraining modelling flexibility. The approach ensures that executable and formal analyses operate over models that respect the intended semantics of the ontology and object model.