DITTO Project Deliverable 1.1

OnTrack Documentation

Version 2: June 2015

(Version 1 circulated February 2015)

Phillip James, Markus Roggenbach and Faron Moller.
Swansea University, UK.

Abstract

OnTrack is an open toolset for railway verification developed within the SafeCap project between Swansea University and Surrey University. Within the DITTO project, OnTrack has been developed further for railway optimization and will serve as a common platform for tool integration between the DITTO partners Southampton, Leeds, and Swansea. Surrey will also continue to collaborate as a DITTO sub-site of Swansea.

In this document, we describe the OnTrack tool for generating formal models from graphical scheme plans for railway signalling systems. Here, we motivate and present the development of the tool and discuss the main architecture of the tool. Along with this, we present exemplar model transformations required to generate formal models. This discussion serves as an illustration on how the tool can be extended for other formalisms.

DITTO

www.dittorailway.uk
1 Supported Processes from Industry

Interlocking applications are developed according to processes prescribed by Railway Authorities, such as Network Rail’s Governance for Railway Investment Projects (GRIP) process. The first four GRIP phases (Output definition, Feasibility, Option selection, Single option development) define the track plan and routes of the railway to be constructed. Phase five – the Detailed Design – is contracted to a signalling company such as Siemens which chooses appropriate track equipment, adds control tables to the track plan, checks that this design is correct, and implements the solid state interlocking. Based on such a Detailed Design, the GRIP process concludes with Construction test and commission, Scheme hand back and Project close out.

Figure 1 provides some detail of how the Detailed Design is realised in a company such as Siemens. The client provides a CAD plan of the track plan and routes and the regulator provides a set of design rules. Based on these, the routes are signalled, i.e., control and release tables are developed. This scheme plan (= track plan plus tables) undergoes a thorough check before the tables are implemented in an interlocking.

Figure 2 shows how one can support such a workflow by including an automated check of the scheme plan for safety conditions. It is still an open research question as how to perform such safety checks. The challenge is how to cope with the complexity of the problem: the state space that one has to explore for safety grows exponentially in the size of the scheme plan to be verified. Here, the OnTrack tool provides several solutions, see Figure 10, which have been demonstrated to scale for small to medium size scheme plans.

Figure 3 further extends the picture. Here, the client additionally requests a certain capacity for the railway to be designed. Similarly to the situation of safety, where we added an automated model checking procedure, we would like to add a further automated model checking procedure for computing the theoretical capacity of a scheme plan.

This new procedure can be based on our models for railway safety. To check for safety, these models have to represent the individual interactions between trains and a signalling system. The objective of WA1 of the DITTO proposal is to incorporate Safety and Theoretical Capacity into System Optimisation. Here, we want to capture the micro-level dynamics and relate these to macro-level behaviour, i.e., the individual interactions between trains and a signalling system shall be connected to macroscopic traffic flow metrics, such as throughput, time of travel and train speed.

We would like to illustrate how such a capacity result might look like. Take for example the single junction shown in Figure 4. This single junction has a main line (from AG to BM) and a side line (from AG to AM). Now the question arises, how many trains one can schedule on this junction within a certain time window.
Figure 1: Design process of scheme plans

Figure 2: Design process extended with formal safety verification.
There is a mutual dependency between the number of trains that one can schedule on the main line and the number of trains that one can schedule on the side line. With simulations on a rather crude model with unrealistic assumptions (such as trains accelerate and brake immediately), we obtained the result shown in Figure 5. Within a time window of 300 seconds, there exist schedules up to the shown step-function: it is possible, to schedule up to 4 trains on the side line; if one schedules 4 trains on the side line, it is not possible to schedule a train on the main line; if one
schedules 3 trains on the side line, one can schedule up to two trains on the main line, etc.

![Capacity diagram](image)

**Figure 5: Single Junction Capacity.**

Such theoretical network capacity describes the freedom one has for optimising schedules. It is one of the objectives of WA1 to develop modelling and tool support for automatically analysing scheme plans for this theoretical network capacity.

## 2 Technologies used within OnTrack

Within the railway industry, defining graphical descriptions is the de facto method of designing railway networks. These graphical descriptions enable an engineer to visually represent the tracks and signals etc., within a railway network. In this document, we outline the OnTrack toolset that achieves the goal of encapsulating formal methods for the railway domain. Overall, the OnTrack toolset provides a modelling and verification environment that allows graphical scheme plan descriptions to be captured and supported by formal verification. Thus, it provides a bridge between railway domain notations and formal specification. This in turn makes formal methods accessible to domain engineers.

In OnTrack, we emphasise the use of a Domain Specific Language (DSL) and the decoupling of this DSL from the verification method. One of the novelties of this is that we can define abstractions on the DSL in order to yield an optimised description prior to formal analysis. Importantly, these abstractions allow benefits for verification in different formal languages. Also, due to the way OnTrack has been designed, it is easily extendable to allow the generation of formal models in any given modelling language. This means that the graphical editor of OnTrack can be used as a basis for generating different formal specifications in different languages. Finally, OnTrack is designed for the railway domain, but the clear separation of an editor with support for abstractions from the chosen formal language is a principle that is more widely applicable.
The above features of OnTrack rely on a number of technologies and techniques that we now give an overview of. We begin by considering the use of Unified Modelling Language (UML) class diagrams as meta-models for DSLs and move through the various Eclipse plugins that support the development of graphical editors and text generation based on such meta-models.

2.1 UML Class Diagrams for DSL Descriptions

UML Class Diagrams [8] are industrially accepted for modelling a variety of systems across numerous domains. Often they are used to describe all elements and relationships occurring within a domain. As such, a UML Class Diagram can be thought of as describing a domain specific language (DSL), and many tools and frameworks use UML class diagrams as a starting point for the description of a domain specific language. A typical example of such an endeavour is given by the Data Model [9] of our research partner Siemens Rail Automation, previously Invensys Rail. It aims to describe all elements within the railway domain. In this section, we briefly discuss the components that form UML class diagrams. A snippet that is similar to the Invensys Rail Data Model for the Railway domain is illustrated using a UML class diagram in Figure 6.

2.1.1 Elements of Class Diagrams

The UML diagram in Figure 6 illustrates many of the main features of class diagrams. Here, we give a very brief introduction to these concepts, further details can be found in [8]. Overall Figure 6 contains:

- Classes, represented by a box, e.g. Node, Point, Platform etc. These represent concepts in the railway domain.
- Properties, listed inside a class, e.g. PointType : Int in the class Point expresses that all Points have an integer to identify the type of Point.
- Generalisations, represented by an unfilled arrow head, e.g. Point is a generalisation of JunctionNode.
- Associations, represented by a line/arrow between two classes, e.g. the Platforms link between Station and Platform. Associations can have direction, and also multiplicities associated with them. The multiplicities on the Platforms association between Station and Platform can be read as: “A Station has zero or more Platforms”.
- Compositions, represented by a filled diamond, e.g. the hasLamp composition for SignalHead and SignalLamp, tell us that one class “is made up of” another class. In a similar fashion to associations, compositions can also have multiplicities.
- Operations are also represented inside a class, although there are none present in the given snippet.

Such a class diagram can be thought of as describing all the components that can be found within a domain.
Figure 6: A snippet of a UML class diagram that is similar to the Invensys Rail Data Model.
2.2 Eclipse Frameworks: EMF, GMF and Epsilon

In this section, we discuss the main Eclipse Integrated Development Environment (IDE) components and plugins that we use for creating domain specific languages and the associated tool support. To this end, we discuss the Eclipse Modelling Framework (EMF) [10], the Graphical Modelling Framework (GMF) [3] and Epsilon [7]. Each of these plugins are developed for Model Driven Engineering and Development [6, 1] of domain specific languages.

2.2.1 The Eclipse Modelling Framework

Many people consider the core of a language to be its abstract syntax. From an abstract syntax, one can develop artefacts such as a concrete syntax or model transformations to another abstract syntax. The Eclipse Modelling Framework [10] is a modelling framework and code generation facility for building tools and other applications based on a structured data model. Part of this framework includes Ecore [10] which is a UML class diagram like language for describing meta-models for DSLs. This model is stored using the XML Metadata Interchange (XMI) file format and can be edited using a number of varying viewpoints. From such a XMI model specification, EMF provides tools and runtime support for producing various Java classes for the model, along with a set of adaptor classes that enable viewing and editing of the model. Finally, such a model serves as the basis for creating a graphical syntax for a DSL using the graphical modelling framework.

![Ecore Meta-model](image)

**Figure 7:** Ecore Meta-model for a simplified version of Ecore.

In Figure 7 we give a UML class diagram view of the Ecore model for a simplified version of Ecore itself. That is, as Ecore is a language to describe abstract syntax, Figure 7 gives the meta-model for this language. This shows that the Ecore language contains the concepts EClass, EAttribute, EReference, and EDataType. All elements have a name property except EDataTypes. The intuitive names correspond to what each element represents in the Ecore language. That is, we can define EClasses which can have EAttributes of a certain type given by EDatatype. Finally, there can be EReferences between EClasses representing an association between the EClasses. These concepts fit
well with some of the UML class diagram concepts defined in Section 2.1, showing the close relationship between Ecore and UML class diagrams.

2.2.2 The Graphical Modelling Framework

The Graphical Modelling Framework project [3] provides the features allowing one to develop, from an Ecore meta-model, a graphical concrete syntax for a domain specific language. The result of applying the GMF process is a graphical editor encapsulating this graphical concrete syntax. Such an editor is shown in Figure 8. This editor consists of a drawing canvas (in the centre) and a palette (right hand side). Graphical elements from the palette can be dragged and positioned onto the drawing canvas. Overall, the editor can be used to produce model instances of the domain specific language described by the underlying Ecore meta-model.

Figure 8: OnTrack GMF Editor for railway track plans.

GMF uses the Graphical Editing Framework GEF for many of its features, but provides a useful development framework on top of GEF. The main features of GMF can be split into two components: a tooling framework for developing graphical editors and a runtime framework for running such editors. Here, we discuss the tooling component.

2.2.3 GMF Tooling

The tooling component of GMF provides easy access and model driven editing to several models that are required to create a GMF editor plugin. Figure 9 shows the development process for a typical GMF editor.
To begin, a new project is created. As part of creating a new GMF project, the user is required to bind the project to an underlying Ecore meta-model. Next, the user fills in details for the following models:

**Graphical Definition Model:** The graphical definition model is where the user can define the various figures to be used for the concrete syntax. Figures are designed for any classes and relationships that need to be present in the editor from the underlying Ecore model. These figures are then collected into a Figure Gallery.

**Tooling Definition Model:** As illustrated in Figure 8 (right hand side), most editors created using GMF include a palette allowing users to create and work with constructs from the concrete syntax of the DSL. The tooling definition model is where users can define and design the elements to be included and displayed in the palette.

**Mapping Model:** The mapping model is one of the most important models used when generating a GMF editor. It is where one can define how elements from the graphical definition model and tooling definition model are linked to elements from the underlying Ecore domain meta-model.

![Diagram](image)

**Figure 9:** The steps involved in developing a GMF Editor.

**Generator Model:** Finally, the generator model combines the information of the previous models with details that are needed to generate code for the editor. The generation of this model from the mapping model is often an automatic step, however it is possible to customise this model to include advanced features such as extension points [3] if required. Finally, this model can be used to generate a domain specific language editor similar to the one shown in Figure 8.

Further details and examples of developing GMF editors can be found in [10, 3].
2.3 Model Transformations and Text Generation

Often, the development of a GMF editor is motivated by the possibility of producing, from an instance model created by the editor, some sort of output usually in the form of text or program code. Similarly, many people wish to transform the model into a slightly different model, or compare it to another model that may be an instance of a different meta-model. To help with these tasks, users can make use of what are known as model transformations. Although there are several possible frameworks for defining model transformations based on EMF meta-model, e.g. QVT [10], JET [10] and Xpand [2], we will concentrate our review on Epsilon [7].

2.3.1 Epsilon

Epsilon, the extensible platform of integrated languages for model management [7], provides a family of languages and features for defining and applying model transformations, comparisons, validation and code generation. In the case of Ecore meta-models, the main types of model transformation which are of interest to us are:

1) Model to model transformation (M2M): Model to model transformations define how a model instance of one Ecore meta-model can be transformed into a model instance of (optionally) another Ecore meta-model. For example, a simple model transformation of a model instance of the meta-model in Figure 7 might be to add to the instance a new “EClass” called Class from which all other classes in the model instance are related via an “isA” association.

2) Model to text transformation (M2T): Model to text transformations can be viewed as model to model transformations, where the output model is simply an instance of the (very general) meta-model defining sequences of characters. Such transformations are often used for code generation from a given model to a programming language. Later in Sections 5, we will use this type of transformation to generate formal specifications from graphical models. Interestingly when generating text, one can opt to use a meta-model for the output text or to skip the meta-model and simply directly output text.

To support the above model transformations Epsilon provides several languages [7] of which we consider and use:

EOL: The Epsilon object language that provides a common set of model management constructs. EOL forms the base language of which the other Epsilon languages are constructed.

ETL: The Epsilon transformation language for specifying model to model transformations.
EGL: The Epsilon generation language for model to text transformations. EGL provides a templating feature for code generation without requiring a meta-model for the output model.

EWL: Finally, the Epsilon wizard language for defining and executing transformation workflows, including activating transformations from a GMF editor.

More details on these languages and their formal definitions can be found in the Epsilon book by Kolovos et al. [7].

3 The OnTrack Toolset (Version 1) Architecture

In this section we highlight the main architecture of OnTrack with respect to the model transformations that are implemented within the toolset. OnTrack has been created using the GMF framework [3] and multiple associated Epsilon [7] model transformations.

Figure 10 shows the workflow that we employ in OnTrack. Dashed boxed indicate features under development.

Firstly, based on a provided CAD track plan and the associated control tables, a user draws a Scheme Plan (SP) using the OnTrack graphical front end (right path). These scheme plans are models formulated relative to OnTrack’s DSL meta-model, see Figure 11. It is work in progress, to read a RailML representation and to transform it into a model relative to OnTrack’s DSL meta-model (left path, with dashed boxes).

A scheme plan is then the basis for subsequent workflows that support its verification, simulation, analysis etc. Scheme plans can then be translated to formal specifications in various (specification) formalisms. This can be achieved in two possible ways:

1) Using a meta-model for the formal specification language.

The first option is to have a meta-model describing the formal specification language. A “Represent” transformation then translates a Scheme Plan into an equivalent Formal Scheme Plan over the meta-model of the formal specification language. Then various model to text transformations turn a Formal Scheme Plan into a Formal Specification Text ready for verification using external tools. These “Generate for Verification” transformations can also enrich the models appropriately for verification. Here, the advantage of this approach is that the “Generate for Verification” transformations can be defined generally for the formal specification language meta-model.
Figure 10: The OnTrack workflow.
2) Direct generation of a formal specification.

The second approach is to directly generate a formal specification. Once again, these transformations turn a Scheme Plan into a Formal Specification Text ready for verification. Again, these transformations can still enrich the models appropriately for verification.

Once a formal model has been generated, it can then be simulated or verified using the tools associated with the formal specification language that has been used for generation. For example, ProB can be used for animating and verifying CSP | | B, SPASS can be used for verifying Casl, TimedCSP Simulator can be used for simulating and visualising train runs.

It is at this point, where OnTrack is extendable, i.e., other models from the DITTO context can be generated. These models then can be simulated and analysed with the respective tools. In principle, this workflow can be fully automatic. In a prototyping phase, one would support it only partially, i.e., OnTrack produces a file that then needs to be loaded into another tool, rather than OnTrack opening this other tool directly.

The remainder of this document is dedicated to providing technical details of the above workflow implementation in OnTrack.
4 OnTrack’s DSL (Version 1)

OnTrack implements the workflow from Section 3 in a typical EMF, GMF and Epsilon architecture [3, 10, 7]. That is, as discussed in Section 2.2.2 a graphical editor realised in GMF is the front end for the user. As a basis for our tool, we have taken a modified version of the DSL developed by Bjørner. The concepts outlined in Figure 11 of Bjørner’s DSL can easily be captured within an ECORE meta-model that underlies our toolset. Practically, this ECORE meta-model is a UML class diagram represented using XML.

Implementing a GMF front-end for this meta-model involves selecting the concepts of the meta-model that should become graphical constructs within the editor and assigning graphical images to them. Figure 8 shows the OnTrack editor that consists of a drawing canvas and a palette. Graphical elements from the palette can be positioned onto the drawing canvas. For example, the linear unit element from OnTrack is now a drawable element. Within the editor, the Epsilon Wizard Language (EWL) for model transformations has been used to implement calls to the various scripts realising different transformations. Below, we give some details on the main models involved in the design of OnTrack, before discussing details of the model transformations for generating formal specifications.

4.1 OnTrack Graphical Model

The graphical model defines the graphical elements to be used within the GMF editor. For OnTrack, we have opted to use a series of SVG (Scalable Vector Graphic) figures for classes within OnTrack’s DSL. Figure 12 shows several of the figures we have used for given classes.

![Figure 12: The OnTrack SVG Figures for signals, linear units and points.](image)

We have defined relationships between the classes as connections in the form of lines between SVG Figures. The full graphical model storing the information on these various figures is given in Figure 13.

From the graphical model definition in Figure 13, we can see that concepts such as Point and Linear are represented as drawable nodes, with an associated SVG filename, within the GMF editor. Also notice that even though unit is an explicit concept in OnTrack’s DSL, it has no direct representation as a drawable node within the graphical model. This is because we do not want users to be able to draw units, but only in their concrete forms of point and linear. We also observe that the element signal does not explicitly occur within OnTrack’s DSL, however from a graphical perspective when designing
scheme plans, railway engineers use signals to distinguish route boundaries. The second aspect we see from the graphical definition is that relationships such as UnitHasC1 (unit has connector from OnTrack’s DSL) are represented graphically as connections between nodes and thus do not have an associated SVG file. Finally, we see a list of Diagram Labels that are used for displaying elements such as a name of a signal on the diagram.

Figure 13: The graphical model definition for OnTrack.

Figure 14: The tooling model definition for OnTrack.
4.2 OnTrack Tooling Model

The tooling model defines the elements that can be selected from the palette of the GMF editor. The tooling model for OnTrack is shown in Figure 14. The elements appearing in this tooling model can be seen in the palette on the right hand side of the editor in Figure 8.

![Tooling Model Diagram]

The tooling model simply contains a list of creation tools. We have defined a creation tool for each element appearing in the OnTrack graphical model. For example, there are creation tools for each of the nodes Signal, Linear etc., and also for each of the connections unitHasC1, unitHasC2 etc. The properties box (at the bottom of Figure 14) shows the details that a user is presented when creating an element on the canvas. For example, a user will be shown the text “Create a new signal” when selecting a signal element from the palette.

4.3 OnTrack Mapping Model

Finally, the mapping model for OnTrack, given in Figure 15, links the graphical model elements to their creation tools defined by the tooling model.

In particular, Figure 15 shows the specific mapping details for the Point element of OnTrack’s DSL. It shows that the underlying model Element for a Point (which is a sub class of Unit) is created by the Creation Tool Point from the tooling model and is represented by the NodePoint figure from the graphical model. This means that in the resulting editor, whenever the user selects the Point element from the palette, the corresponding NodePoint figure will be drawn to the canvas, and the underlying Point (and hence Unit) element created in the model instance.

Finally, using the three presented models, a generation model for the OnTrack editor was automatically created, and the code for the editor in Figure 8 was generated from
this model. The generated OnTrack editor has been extended with a series of model transformations defined using the Epsilon framework [7]. These model transformations implement the transformations that are shown in the OnTrack workflow in Figure 10.

![Image: The mapping model definition for OnTrack.](image)

**Figure 15:** The mapping model definition for OnTrack.

5 Model Transformation for Generating Specifications

Here we describe the second approach of generating formal specification, namely the direct implementation of the generate specification for verification transformations. We illustrate this approach for the concrete case of Casl. For a discussion of the meta-model based transformations, we refer the reader to our work on OnTrack’s transformations allowing output of CSP | | B specifications [5].

The generate text for verification transformation translates meta-model instances of OnTrack’s DSL into formal specification text. This transformation is implemented using the Epsilon Generation Language (EGL) [7] for generating text. We have designed the generation such that it mirrors the specification structure and modelling approach given in [4]. EGL allows template files to be written describing the text to be generated. These templates provide two main features for outputting text, namely the ability to output static text and to output dynamic text. Static text is considered text that is always generated independent of the model that the text is being generated for. By contrast, dynamic text is text that is text dependent on the given model. By default, any text written in an EGL template is considered to be static text. For example we know that the specification of datatypes, OnTrack’s DSL specification and similarly our extension of this with a generic specification of dynamical aspects is the same for all models. Hence this is rather straightforwardly encoded as static text to be always output, for example see Figure 16.
Later in the same EGL template, we can then specify the output of the concrete track plan created using OnTracks graphical editor. Obviously, the details of the text to be generated for the track plan depends on the concrete model under consideration. For example, consider the free type of Units. Such a free type is built from the concrete elements of linear units and switch points contained within the graphical model. Hence we can specify the template in Figure 17 for the dynamic generation of the free type Unit. The result of applying this EGL fragment, for example, to the concrete track plan in with track units la1, la2, ..., lb13 is the following Casl fragment:

free type Unit ::= la1 — la2 — la3 — la4 — ... — lb12 — lb13.

Considering Figure 17, the first element of EGL that we notice is the use of [% and %]. Any text specified between such a set of brackets is interpreted as code. For example, the line var rail : RailDiagram := RailDiagram.allInstances().at(0); is a line of EGL code for declaring the variable rail and assigning to it the current track plan instance within the graphical editor. This variable, can then be used throughout the EGL template to refer to the current model instance.

Next, we see an EGL if statement (line 5). This statement checks the number of elements in the hasUnits relation of the current rail diagram. If there are linear units or points that have been drawn in the diagram, the code inside the if statement is executed. The first line within the if statement is a piece of static text to be generated. That is, as long as the if statement is entered, the text free type Unit ::= will always be output by the EGL template. Lines 9 through to 15 perform a loop through the units of the concrete track plan instance. For each unit up until the last but one in the collection the dynamic text generation [%=unit.id%] is executed (line 14). Here, the dynamic text generation also
contains the = symbol. This indicates that the text following is a piece of code that returns a value. For example, unit.id is a field containing the name that has been given to the current unit element. This name will then be output by the generation process. The dynamic text generation block on line 14 is immediately followed by the static text generation “—”. This produces the “—” symbol between elements of the free type. Finally, after the while statement, there is another block of code (lines 17 to 20) that outputs the last unit identifier in the collection. For this unit, there is no static generation of the “—” symbol which matches the expected output for the definition of a Casl type.

```
1. [%
2. var rail : RailDiagram :=
    RailDiagram.allInstances().at(0);
3. %]
4. ...
5. [%if(rail.hasUnits.size > 0){%]
6.   free type Unit :=
7.   [%
8.   var i := 0;
9.   while (i < rail.hasUnits.size()-1){ %]
10.      [%
11.         var unit : Unit := rail.hasUnits.at(i);
12.         i := i+1;
13.      %]
14.     [%=unit.id%] !
15.   [%}
16. %]
17. [%
18.   var unit : Unit := rail.hasUnits.at(i);
19. %]
20.   [%=unit.id%]
21. [%]%
```

**Figure 17:** Dynamic text generation for the concrete elements of the free type Unit.

In a similar manner to the presented free type generation, it is possible to continue to explore the elements of the diagram generating the concrete track plan specification in Casl. For example the various predicates and operations that encode the topology of the track plan are all represented in the diagram through associations similar to the “hasUnit” from Figure 17. Finally, after generation of the track plan, the safety property to be proven over the track plan can also be generated through a combination of static and dynamic text generation, see [4] for further details. The result is a full Casl ready for verification of the current editor model instance.
6 Summary

The OnTrack tool automatizes workflows for rail simulation, rail verification, rail analysis. It currently supports entering a scheme plan via a graphical editor and its transformation into a number of specification formalisms. The resulting models can then be used as input to various tools associated with these formalism for various analysis purposes. This process is open to the integration of further formalism and tools. Furthermore, the workflow can be automatized to a higher degree.

Overall, OnTrack provides a flexible and extendable toolset that is usable by engineers from the railway domain to produce formal specifications ready for verification, simulation and analysis. Hence, OnTrack will provide the common open tooling environment for the DITTO project (see also Appendix A). The next stage of our work will involve the development of a basic prototype for interfacing between existing tools and initial safety analysis based on movement authorities.
7 References


Appendix A: Ditto Overview

OnTrack shall serve as a common platform for tool integration between the DITTO partners Southampton, Leeds, and Swansea. Surrey will also continue to collaborate as a DITTO sub-site of Swansea. Below we give an outline of the main aims and objectives of the Ditto project.

This project will combine and build upon the work undertaken by three of the project teams in the RSSB/EPSRC Capacity at Nodes programme (Challenging Established Rules for Train Control, OCCASION and SafeCap) with a view to making a significant contribution to meeting the requirements of the Future Traffic Regulation Optimisation (FuTRO) programme and to UK rail related research more generally. It will contribute to FuTRO by establishing basic principles and proofs of concept and by developing optimisation formulations, algorithms and processes that will deliver a step change in rail system performance and help to meet future customer needs. This will be done by taking into account developments in human and automatic control on trains and in control centres (particularly related to ERTMS) and by making better use of data, particularly with respect to time and position of trains. The proposal contains four inter-related and complementary technical strands, with specific aims as follows:

1) Safety – although FuTRO currently resides in the management layer of railway operations, safety is a fundamental and overriding consideration in operations management and control. The safety strand of the proposal underpins the traffic management strands, allowing optimisation activities to proceed in the knowledge that safe operating conditions are being maintained and that theoretical capacity limits are not being exceeded. The tools developed will also have generic applications to traffic regulation.

2) Reliability – the trade-offs between the provision of additional train services, and the resultant increases in capacity utilisation, and the maintenance of service quality are an area of particular interest within the industry, and this strand of the proposal aims to quantify these trade-offs so as to develop timetables that optimise capacity utilisation without compromising service reliability.

3) Dynamic simulation – micro-level data on the status of individual trains will be combined to produce an optimal, macro-level outcome, transmitting the system-wide needs back to the microlevel, so that individual train movements can be optimised within overall system requirements.

4) Network integration – we will produce optimised timetables that can be adjusted in real time through dynamic simulation. We will examine the scope for artificial intelligence to combine our optimisation and simulation tools to produce tractable solutions to real-time traffic control.