Designing domain specific lanaguages for verification and applications to the railway domain.

James, Phillip

Designing Domain Specific Languages for Verification and Applications to the Railway Domain

Phillip James

Submitted to Swansea University in fulfilment
of the requirements for the Degree of Doctor of Philosophy

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

January 2014

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed . ... (candidate)

Date 31/1/14

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed .... (candidate)

Date 31/1/14

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ... . (candidate)

Date 31/1/14

Abstract

The development and application of formal methods is a long standing challenge within the field of Computer Science. One particular challenge that remains is their uptake into industrial practices. Even though many successful research projects have illustrated the applicability of formal methods to industrial practice, their use within industry is often still limited. This thesis introduces a methodology for developing domain specific languages for modelling and verification that aims to aid in the uptake of formal methods within industry. It also concretely illustrates the success of this methodology for the railway domain.

The presented methodology addresses some of the issues limiting the use of formal methods within industrial practices. Concretely it addresses issues surrounding faithful modelling, scalability of verification, and accessibility to modelling and verification processes for practitioners within the domain. To achieve a faithful, scalable and accessible procedures, we present a design methodology for creating domain specific languages based around the CASL specification language. At the core of this methodology is a domain specific language that abstracts away technical details of Computer Science from the end user. The starting point for the suggested methodology is an informal domain description in the form of a UML class diagram. The result is a graphical tooling environment encapsulating formal modelling and verification for systems from the given domain.

As well as presenting the various steps involved in our methodology, the thesis provides the technical constructions required for the methodology. Namely, various institutions supporting the capture of UML class diagrams, and an institution comorphism from these into MODALCASL. We also propose the use of domain specific lemmas for supporting verification within a domain. Finally, to illustrate that our methodology meets our aims of being faithful, scalable and accessible, we successfully apply it to two example problems from the railway domain. One of which is an industrial example supplied by Invensys Rail. The application to these problems results in the OnTrack toolset that encapsulates, into an accessible graphical editor, various approaches for verification within the railway domain.

Acknowledgements

I would like to show my appreciation and gratitude to Dr. Markus Roggenbach for his continued support, guidance and patience throughout the time it has taken to complete this work. He has successfully guided me through the rocky roads of a PhD and I have learnt so much from the times we have spent together.

My thanks is extended to my examiners Prof. Magne Haveraaen and Prof. John Tucker. Their valuable time and comments have helped improve this thesis. I would also like to thank all of the Department of Computer Science for making my time in Swansea so memorable. I especially thank Dr. Anton Setzer for being my second supervisor, and Prof. Faron Moller and Dr. Arnold Beckmann for their insightful comments towards this work. A special thank you also goes to Erwin R. Catesbeiana (Jr) for signalling me towards the end of the tunnel. I am also grateful for the continued interest and time of Simon Chadwick, Dominic Taylor and David Johnson from Invensys Rail.

During my time as a PhD student I have had the privilege of meeting and working with many great friends. I thank both Fredrik Nordvall Forsberg and Matthew Gwynne for (literally) being at my side every day. Without Fredrik there would be many hills that would have remained unconquered, whilst Matt often provided a needed source of distraction. I also thank Liam O'Reilly for providing a hotline for support on both a professional and personal level. I would like to thank the following close friends who have all contributed in so many different ways to this thesis: Stephen Richards, Temeshgen Kahsai, Hoang Nga Nguyen, Karim Kanso, Andrew Lawrence, Casper Bach Poulsen, Margit Hanssen, Matt Edmunds, Ian Doidge, Liam Betsworth, Jennifer Pearson, Tom Owen, Simon Robinson and Patrick Oladimeji.

A big thank you goes to Edward Thom for his thorough proof reading. I also thank Edward, his wife Alison and daughter Rachel for the many "mini breaks" that often provided a much needed rest.

A special thank you is reserved for my family. Gregory my Dad, Corinne my Mum, Michael my brother, Kimberley my sister and Dilwyn, Beryl and Sylvia my Grandparents. Your continued love, support and encouragement made all this possible, thank you.

Finally, I am eternally grateful to my wife Emma for sharing this often tricky journey with me. Together, with your love, any mountain feels conquerable. I truly thank you.

Contents

Chapter 1

Introduction

Contents

Formal verification of railway control software has been identified as one of the Grand Challenges of Computer Science [Jac04]. As is typical with formal methods, this challenge comes in three parts: the first addresses the question of whether the proposed mathematical models faithfully represent the physical systems of concern; the second is a question of whether the proposed technologies scale up to systems of an industrial size and complexity; the third is the question of how to employ and utilise available technologies in a manner that is accessible to practitioners in the domain of interest and not just to the developers of the approach. This chapter introduces the aims and contributions of this thesis towards overcoming these issues.

1.1 Domain Specific Languages for Verification

For many years, the application of verification processes such as model checking and interactive theorem proving to varying industrial case studies has been successfully illustrated, e.g. see [Sim94, BG00, Win02, WR03, PGHD04, HKRS09, Jam10]. Even though these approaches have been successful from a Computer Science perspective, the adoption of formal methods within industry is still limited [BH06] due to questions around faithful modelling, scalability and accessibility. Without experts in the field of formal verification, the modelling approaches presented are often in a form that is acceptable to computer scientists, but not to the engineer working within the domain.

These presentations thus lead to doubts in the approach by the engineers and a low level of confidence towards the capabilities of the approach to correctly capture the systems being modelled. At the same time, many methods of verification are prone to suffering from a scalability problem that makes their application to large industrial problems lengthy and often unfeasible. Finally, tool support for verification procedures is often aimed towards a Computer Science audience interested in verification and hence is not easily accessible to engineers outside the field of formal methods. This thesis presents a new approach to solve these issues for a given application domain. The approach combines a broad range of results to bridge the gap between theoretical verification approaches and practical application. The underlying theme to the approach is that

> "Domain Specific Languages (DSLs) can aid with modelling, verification and encapsulation of formal methods tools within a given domain".

In this thesis, we show this hypothesis to be true within the railway domain. To this end, we provide a general methodology for the integration of the formal methods within industrial design processes. We base our work on the CASL algebraic specification language [Mos04b, BM04] and automated theorem proving. We apply this methodology to two comprehensive case studies from the railway domain, demonstrating the chosen methodology achieves our three design aims.

1.2 A Design Methodology

To achieve a faithful, scalable and accessible modelling and verification procedure, we present a design methodology for creating domain specific languages [MHS05, FP10] based around the CASL specification language. Domain specific languages (DSLs) aim to abstract away technical details of Computer Science from the domain engineer, allowing them to create programs or specifications without having to be an expert programmer or specifier. Considering Figure 1.1, the starting point for the suggested methodology is an informal domain description, as is often found within industry in the form of natural language and accompanying UML (Unified Modelling Language) [Obj11]. It is commonly the case that graphical notations are used to depict the elements found within these domain descriptions. The result of following the presented methodology is the creation of a graphical domain specific language that not only makes the task of automatic verification possible, but also scalable. Such verification gains are achievable via careful design of the domain specific language to ensure it captures and exploits domain knowledge relevant to the class of properties which one would like to verify. The overall result from our methodology is a graphical tooling environment which incorporates a "push button" verification process for critical systems within the given domain. The methodology is not only theoretically sound but also ensures faithful modelling and verification techniques that scale and are supported by tools that are usable by the every day domain engineer.

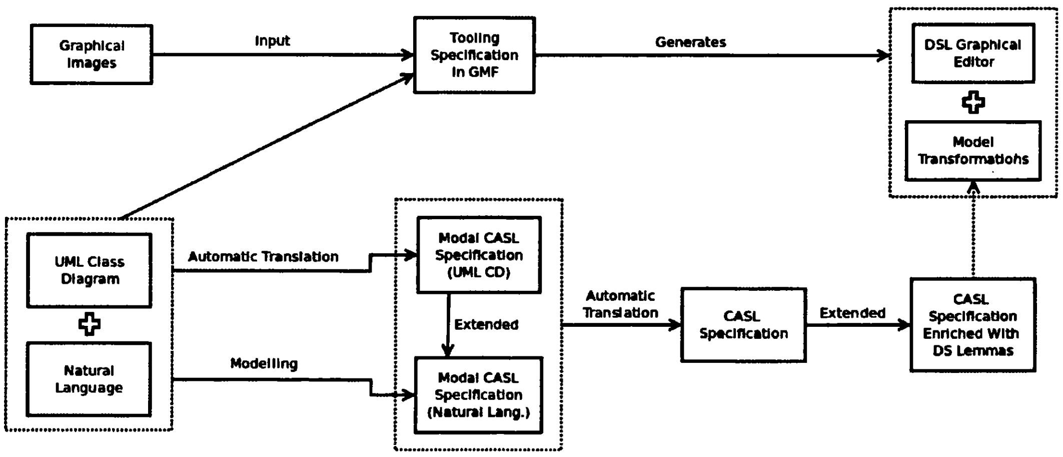


Figure 1.1: A methodology for designing domain specific languages aimed at verification.

Concretely, the methodology results in two processes. The first is designing the domain specific language and associated tool support. The second a verification process used when applying the tools output from the design phase.

1.2.1 Designing the DSL and Tool Support:

The design process is undertaken by a team comprising of a computer scientist working in close collaboration with a domain engineer. A close working relationship ensures the resulting domain modelling is faithful. Considering Figure 1.1, the following steps are involved in the design process:

1. *Informal DSL Design and Automatic Translation.* The first step in the methodology involves the domain engineer classifying all the concepts, attributes, relations etc. within the domain into an informal DSL using UML class diagrams [Obj11] and accompanying natural language descriptions (or, in the language of Bjørner, a narrative [Bjø09]). This step is often already undertaken within industry when describing a domain. It is also standard practice to use UML class diagrams for this task, see for example [Rai10, AG07]. From such UML class diagrams, names and relations can be automatically extracted and translated into a formal specification in ModalCasl. We suggest and support a particular automatic translation from UML class diagrams to ModalCasl that we define in Chapter 5. We also apply this translation to our two case studies within the railway domain.

2. *DSL Analysis.* Next, the domain engineer and computer scientist consider the resulting formal ModalCasl specification. At this stage, the ModalCasl specification may be extended to incorporate the natural language descriptions from the informal domain description. They may also be extended with proof goals encoding any properties to be proven in the verification step. At this point, for the sake of better proof support, we suggest the application of an existing

automatic translation from ModalCasl to Casl. Once both are happy all elements of the domain are modelled faithfully, the computer scientist can begin the task of supporting the DSL with domain specific lemmas. Naturally, there cannot be a universal solution to finding such domain specific lemmas. However, in our experience, for all the DSLs we have considered, such lemmas have existed and followed from knowledge of the domain engineer. We discuss such lemmas in Chapter 6 and show that these lemmas allow for scalable verification based on ideas that are often inherent within the domain. Overall, this step ensures faithful modelling of the domain, giving the domain engineer a higher level of confidence in the approach. It also enables scalability of the approach through the application of domain specific lemmas.

3. *Graphical Editor Creation.* The creation of a domain specific language is often aided by the use of a development framework. There are several examples of such tools including ASF+SDF [vdBvDH$^+$01] a meta-environment based on a combination of the algebraic specification formalism ASF and the syntax defining language SDF. ASF+SDF allows creation of domain specific languages and tools such as parsers, compilers and static analysers for the created language. There is also MetaEdit+ [KLR96], which is an industrial tool allowing the creation of visual domain specific languages. Interestingly, MetaEdit+ has been used to create a domain specific modelling for railway layouts, see [KLR96]. With respect to our methodology, we make use of the *Graphical Modelling Framework*, GMF [Gro09]. GMF is an Eclipse plugin that provides the infrastructure to create, from a UML like model, a Java based graphical editor. Based on the UML class diagrams captured in Step 1, the domain engineer and computer scientist use GMF to create a tooling environment for the DSL. This allows for elements to be presented in the tool in a way that is accessible to the domain engineer through use of graphical representations that are native to the domain. Such an approach is illustrated in Chapter 7 where we give details of the OnTrack Toolset for modelling and verification within the railway domain. The result of this step is a tool that allows graphical description of systems formulated over the informal DSL, that is, the UML class diagram and accompanying narrative.

4. *Model Transformation Development.* The final step in the methodology is down to the computer scientist. The editor designed in Step 3 is open (via the Epsilon framework [KRPP13]) to extension with model transformations [SK03, Kus06, KRPP13]. Such transformations allow for the graphical models produced by the editor to be translated to text. Thus, the graphical editor can be extended with model transformations to allow output of Casl specifications formulated using the specifications from Step 2. The development of these model transformations is illustrated in Chapter 7. The result of this process is a tool for generation of formal models that is readily usable by engineers from the domain under consideration.

Overall, following this design process ensures we meet our three aims: faithful modelling is achieved by starting with an informal industrial description and forming a formal specification through a close working relationship between the domain engineer and computer scientist; scalability of the verification procedure is achieved through the domain knowledge captured in property supporting domain specific lemmas; finally, accessibility to the specification and verification of systems is achieved through graphical tooling incorporating domain specific concepts and constructs.

1.2.2 The Verification Process:

The result of applying our methodology is a framework to accommodate the verification process illustrated in Figure 1.2. This process is undertaken purely by the domain engineer using the toolset gained from the design process and follows three main steps:

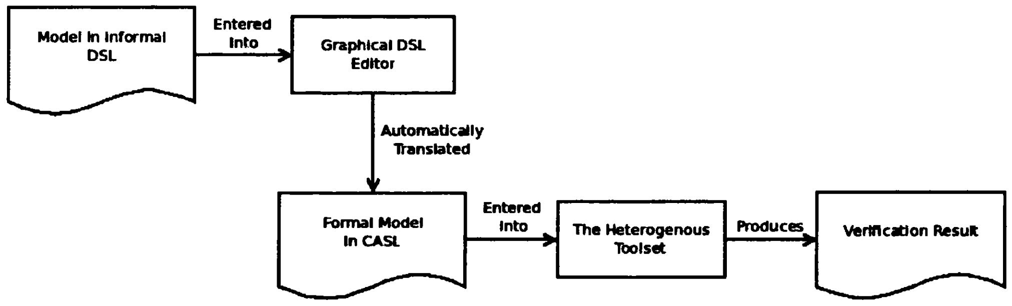


Figure 1.2: A verification process based on the designed tools.

1. *Model Development Based on the Informal DSL.* The first (optional) step within industry is to outline or specify a design informally. This step should be undertaken, and often is, using the vocabulary outlined within the informal DSL captured by Step 1 of the design process.

2. *Graphical Modelling.* Next, the domain engineer can encode the system design using the graphical editor. Once encoded, the engineer can then automatically produce formal specifications ready for verification. As the graphical editor contains constructs that are specific to the domain, the learning cost for a domain engineer is minimal due to familiarity with the editor constructs.

3. *Verification.* Finally, the formal specifications produced in Step 2 can be automatically verified using (in our case for CASL specification) the Heterogeneous Toolset HETS [MML07]. Due to the domain specific lemmas that were developed during the design process, such verification is "Push Button".

Each step in this verification process meets the aim of being accessible thanks to the tools constructed as part of the design methodology.

1.2.3 Application within the Railway Domain

The railway domain is a prominent example of where formal methods have successfully been applied, but uptake of such methods within industry is limited. Approaches that have been taken include algebraic specification, e.g. [Bjø09], process algebraic modelling and verification, e.g. [Win02, PGHD04], and model oriented specification, where, for example the B method has been used in order to verify part of the Paris Metro railway [BG00] in terms of both safety and liveness properties. The above approaches have illustrated the successful application of formal methods to the railway domain, but all fail to comment on the faithfulness and applicability of such processes by domain engineers. Many of the approaches also note problems with scalability or are illustrated on small industrial examples where the question of scalability is unclear.

Along with presenting the technical details enabling the methodology in Figure 1.1, this thesis, in co-operation with Invensys Rail, instantiates the methodology with both academic and industrial examples for verification within the railway domain. Invensys Rail [Inv13] are a world leading supplier of railway solutions, varying from control software to trackside equipment. Our association with Invensys Rail has led to faithful railway models that are formal and analysable by current verification technologies. Also, we do not want to hide the engineering knowledge held by our industrial partners, and have exploited this knowledge in several places to gain clever domain specific abstractions that aid with verification, thus presenting a "proof of concept" that our approach works in practice.

The first presented application to the railway domain was inspired by Bjørner [Bjø09], whose natural language specification of the railway domain we follow for our academic example. Bjørner has given a formal version of this natural language specification using the Raise Specification Language (RSL) [RAI93]. In contrast, we focus on using CASL, the Common Algebraic Specification Language [Mos04b] for our methodology, as it provides us with more features than RSL, including, importantly, established tool support in the form of the Heterogeneous Toolset (HETS) [MML07]. For our industrial example, we use as a starting point the Invensys Rail Data Model [Rai10] which has been developed by our industrial partner and aims to describe all elements in the railway domain. For both Bjørner's domain description and the Invensys Rail Data Model, we successfully provide a set of domain based lemmas that aid with verification. This demonstrates our overall aim that domain specific languages can be designed to support automatic verification. We then present the OnTrack tool which is a graphical railway layout specification tool that we have designed in co-operation with Invensys Rail, thus ensuring accessibility to the described verification process by railway engineers. Finally, to illustrate that our approach does in fact reach the aim of scalability and allow for the verification process described in Section 1.2.2, we model and automatically verify several real world railway designs. This verification ensures that train collisions do not occur within those models.

1.3 Modelling and Verification in the Railway Domain

In recent years there has been a large amount of interest [FH98, Bjø00, BG00, Win02, WR03, BCJ$^+$04, PGHD04, KMS08, Bjø09, Hax12, Win12] in the application of formal methods, including formal verification of systems within the railway domain. Here we concentrate particularly on the formal verification of railway computer systems with regards to safety properties.

Various forms of formal methods have been applied for safety verification within railways. These include approaches using process algebraic modelling and verification in CSP (Communicating Sequential Processes) by Winter [Win02] and classical contributions by Simpson [Sim94] and Morley [Mor93]. Approaches using SAT-based model checking [KMS08, Jam10, JR10] and also model-oriented specification using the B method [BG00] have also been considered. With respect to specification languages, Haxthausen has used techniques from Algebraic Specification [HP00], whilst the algebraic specification language ASF-SDF [Ber89] has been used for language prototyping of DSLs in areas as diverse as railway interlockings [GvVK95] and financial products [AvDR95]. With respect to modelling, Dines Bjørner has notably developed a DSL for the railway domain [Bjø00, BGP99, Bjø03, Bjø09, BCS99]. Bjørner's DSL – the DSL on which we build later – has been applied in the PRaCoSy (People's Republic of China Railway Computing System) project to model a 600km line between Zhengzhou and Wuhan [BCS99].

There have been a number of approaches that apply model checking to the verification of concrete control programs from the railway domain. For example a comparison of the use of different model checkers in the analysis of control tables has been conducted by Ferrari et al. [FMGF11]. They state that model checking large interlocking systems is unfeasible with current state-of-the-art model checkers, in particular SPIN and NuSMV. However, James et al. [Jam10] demonstrate better results and the feasibility of a lower level approach involving program slicing. Also, Winter in a recent paper [Win12] has considered different optimising strategies for model checking using NuSMV and demonstrates the efficiency of their approach on very large models. Others have also applied theorem proving to the verification of railway interlocking systems, for example, the Advance FP7 project [adv] is aimed at developing Event-B models of such systems and verifying comparable safety properties. A number of prominent studies from the B community towards large scale verification include [LFFP11, SBRG12, Ant11]. A detailed comparison with these approaches is not appropriate for this thesis as our approach is at a higher level of abstraction. The justification for this higher level of abstraction is that our industrial partners wish to have feedback during the design process of interlocking systems, before the costly development of the concrete interlocking system begins.

Finally, there are several projects with a close relevance to this work. The first is the development environment for verification of railway control systems created by Haxthausen and Peleska [HP07]. This environment includes a DSL allowing modelling of control systems, and an automatic translation from models described in this DSL to executable control programs. At each level of production, various safety checking steps are taken. The difference between our approach and this one is twofold. Firstly, we present a methodology that is independent of the domain, whereas the approach by Peleska is focused in particular on railway control systems. Secondly, their approach is based on three levels of verification ending with a concrete control program. Whereas in this thesis we concentrate on the design level, considering how to provide graphical tool support with verification for such a level. Next, is the SafeCap project [saf] which has the aim of improving railway capacity safely by integrating proof-based reasoning about time and state-based models. Part of the project aims to develop an intuitive graphical domain-specific language [IR12a] for the railway domain with a tailored toolset [IR12b, ILR13] supporting verification of railway plans. Their approach is based on Event B [Abr10] and the Rodin framework [ABH$^+$10]. The approach taken in the development of this graphical language is inspired by the methodology we present in this thesis (in fact, the SafeCap DSL and Toolset is developed in co-operation with the author of this thesis). Finally, recent work by Kanso [Kan13] presents a framework that aids in the development of verified railway interlockings. The framework is built around the Agda theorem prover and has been applied to verify two existing railway control systems. The approach by Kanso also presents novel results on integrating model checking into the interactive theorem prover Agda. Here we differ to Kanso, as we concentrate on designing verification processes with industrial applicability in mind, rather than applying new formal methods to industry in an exemplary fashion.

1.4 Aims and Contributions

As stated above, the overall aim of this thesis is to show and explore the hypothesis that domain specific languages can aid with verification. The main result of this thesis is a novel design methodology for creating domain specific modelling languages for verification. It provides the theoretical framework forming the basis of this methodology, and illustrates, concretely for the railway domain, that applying the methodology results in a practical and usable graphical specification tool with automatic verification support. The work that has been completed can be split into the following main results:

UML Class Diagrams to Modal CASL: The first result of this thesis is to provide the theoretical framework allowing one to utilise industrial DSLs, formulated as UML class diagrams, for verification. As UML class diagrams only capture the static system aspects, we make the realistic assumption that the class diagram is accompanied with some natural language specification describing the dynamic system aspects. Often, for Railways this situation is given, as we shall see later, by generally accepted standard literature, for example, the description by Kerr [KR01].

On the technical side, the construction we provide is based on institution theory: to capture realistic class diagrams, we extend the UML class diagram institution by Cengarle and Knapp [CKTW08] with numerous concepts. UML class diagrams also describe invariants that hold during all executions of a system. However, they do not offer the ability of capturing a system's dynamics. To allow for this, we employ ModalCasl [Mos04a] as a general framework for specifying Kripke-structures. We give an institution comorphism, or semantic preserving map, from UML class diagrams to ModalCasl. Part of this mapping is a general construction, namely that of a Pointed Powerset Institution, which factors out a general construction principle necessary for connecting UML class diagrams with an arbitrary institution capturing system dynamics.

In ModalCasl, we then model system dynamics according to the natural language specification describing the dynamic system aspects. For the sake of better proof support, we apply an already established institution comorphism from ModalCasl to Casl [Mos04b]. Overall, the result of this translation allows the direct importation of DSL specifications from industry into the design methodology we have proposed.

Domain Specific Lemma Design: The second part of the proposed methodology is to support the formal DSL from above with sufficient lemmas for verification. We suggest the following approach: Given a DSL for a particular class of systems and a set of properties one is interested in, the DSL can be systematically extended with domain specific lemmas to allow for automatic verification. We claim that such an approach can be applied whenever one designs or adapts a DSL for verification. The overall aim of our approach is to develop a "push button" verification process for critical systems.

Formally, we consider our DSL as a loose specification, the logical closure of which we regard as implicitly encoded "domain knowledge". The second result of this thesis shows how to systematically exploit this "domain knowledge" for automatic verification, particularly within the railway domain. Concretely, we show that for a given property, one can make certain parts of the DSL explicit in the form of lemmas proven once and for all over the DSL. Given a concrete model described using such a DSL, these lemmas can be exploited to aid verification. In general, for a DSL the verification aims are known in advance. Hence, it is possible to provide a set of supporting lemmas that make verification feasible for these known aims.

To illustrate the usefulness of such lemmas, we extend the Casl DSL gained from our translation from UML for automatic proof support. Finally, we model and automatically verify several track plans provided by our industrial partner Invensys Rail as real-world challenges. This provides concrete evidence that exploiting domain knowledge eases verification. To provide proof support, we use various automated theorem provers which are accessible via the Heterogeneous tool-set [MML07].

The OnTrack Toolset: The third result of this thesis is the OnTrack Toolset. OnTrack provides graphical tool support for specifying and verifying railway layouts. It is the resulting DSL editor gained from applying our methodology and encapsulates the verification process for the railway domain. OnTrack automates workflows for railway verification, starting with graphical scheme plans and finishing with automatically generated formal models set up for verification. OnTrack is grounded on the domain specification language by Bjørner and allows for generation of Casl models prepared for verification using the domain specific lemmas from the previous section. In addition to this, OnTrack has also been designed in a way that allows it to be generic in the formal specification language used. For example, we have also extended OnTrack to produce CSP||B models [JTT$^+$13].

To create the OnTrack Toolset, we have used the GMF framework [Gro09] that provides the means for the user to take a UML class diagram and create a graphical editor for concepts contained within this diagram. From this, we have implemented various model transformations using the Epsilon framework [KRPP13]. These model transformations allow the production of formal models. They also make the toolset extendable in the sense that production of formal models in any given formal specification language is possible through the implementation of another model transformation. Our use of a DSL as the basis for OnTrack also allows for the formulation of abstractions that work for verification in several formal specification languages, and independent of the verification applied approach. This approach illustrates that abstraction principles are often inherent to the domain under consideration and not only to the formal specification language used.

In Chapter 7, we give details of the implementation of OnTrack using GMF. We also show how the model to text transformation for generating CASL models has been implemented, illustrating how the tool can be extended to produce formal models in other formal specification languages.

Industrial Application to the Invensys Rail Data Model: Throughout the thesis, we use the established academic domain modelling of Dines Bjørner to illustrate our approach. To illustrate that our approach not only works for academic scenarios and that we have achieved our aim of having industrial strength support for verification, we also present results based on our industrial partners domain modelling. We firstly show that our UML to MODALCASL translation scales up to industrial size UML diagrams in the form of the Invensys Rail Data Model [Rai10]. Then, we show that extension of this domain specific language with our presented domain specific lemmas allows for successful verification. We again verify several real world scenarios from our industrial partner. This illustrates that the core concepts of our methodology are applicable on an industrial level. Finally, work towards an industrial strength editor is ongoing, and discussions regarding technology transfer to Invensys rail are underway.

1.5 Published Material

The work presented in this thesis draws from a series of publications at various conferences and workshops. These include:

Designing Domain Specific Languages for Verification: First Steps [JR11]
(Phillip James and Markus Roggenbach. ATE 2011)
Presents a first attempt at a methodology for designing domain specific languages for verification. The paper outlines the main steps of the methodology and gives a small example illustrating how it aids verification. The methodology presented in this thesis adapts and extends the first approach from this paper.

Designing Domain Specific Languages – A Craftsman's Approach for the Railway Domain using CASL [JKMR13]
(Phillip James, Alexander Knapp, Till Mossakowski and Markus Roggenbach. WADT 2012)

Presents the main theoretical results of the translation from UML class diagrams to ModalCasl including a new UML class diagram institution and a comorphism from this to ModalCasl. The paper also presents the application of this comorphism to Bjørner's DSL. Chapter 5 is mainly based on the results in this paper. For this paper, guidance on UML and UML institutions was provided by Alexander Knapp, and details on the ModalCasl institution were provided by Till Mossakowski.

Using Domain Specific Languages to Support Verification in the Railway Domain [JBR13]
(Phillip James, Arnold Beckmann and Markus Roggenbach. HVC 2012)
This short paper and accompanying poster presents some first results on creating domain specific lemmas for the railway domain. The work also presents a domain based abstraction technique for railways. The results given in Chapter 6 of the thesis build on this work, but in a more refined form. Arnold Beckmann provided input on techniques for measuring the theoretical improvements gained in terms of proof complexity. In this thesis, we do not consider such proof theoretic measurements.

OnTrack: An Open Tooling Environment For Railway Verification [JTT+13]
(Phillip James, Matthew Trumble, Helen Treharne, Markus Roggenbach and Steve Schneider. NFM 2013)
Presents the first version of the OnTrack tooling environment. The paper gives details about the tool architecture including a discussion of the domain based abstractions that are implemented in the tool. It also gives details of a CSP||B model transformation for generating formal models in CSP||B. Finally, it discusses how to extend the tool to produce formal models in other specification languages. The work presented in this paper constitutes part of Chapter 7. The tool was developed in co-operation with Matthew Trumble and Helen Treharne at Surrey University. The editor construction was a joint effort, the model abstractions were implemented by the author of this thesis, and Matthew Trumble and Helen Treharne provided the model transformation to CSP||B.

Verification of Scheme Plans using CSP||B [JMN+13]
(Phillip James, Faron Moller, Hoang Nga Nguyen, Markus Roggenbach, Steve Schneider, Helen Treharne, Matthew Trumble and David Williams. FM-RAIL-BOK 2013)
Presents an overview of the approaches taken in various papers [MNR+13, MNR+12a, MNR+12b, MNR+12c] towards the verification of railway scheme plans in CSP||B. The aim of the paper is to form part of a body of knowledge on railway verification. The work presented is a result of a joint effort and ongoing collaboration between members of Computer Science Departments at both Swansea and Surrey University. Our main contribution to this work was towards the presented abstractions, experiments and the tooling environment provided by OnTrack. The author was also involved in the design of the CSP||B models. Some aspects of the paper are presented in Chapter 7 of the thesis.

Verification of Solid State Interlocking Programs [JKL+13]
(Phillip James, Karim Kanso, Andy Lawrence, Faron Moller, Markus Roggenbach, Monika Seisenberger and Anton Setzer. FM-RAIL-BOK 2013)
The main results in this paper combine the work of three previous MRes projects (including that of the author of this thesis) towards the Model Checking of concrete interlocking programs. Once again, the presented work is aimed at an established body of knowledge publication. The work presented in this paper does not directly contribute to this thesis.

On Modelling and Verifying Railway Interlockings: Tracking Train Lengths [To appear in SCP]
(Phillip James, Faron Moller, Hoang Nga Nguyen, Markus Roggenbach, Steve Schneider and Helen Treharne. Accepted for publication in Science of Computer Programming)
Presents an approach towards modelling the lengths of trains, extending previous work on modelling and verification using CSP||B. The paper is based on the following technical report: On modelling and verifying railway interlockings: Tracking train lengths. Technical Report CS-13-03, University of Surrey, 2012. The main results in this paper, although related, are not presented in this thesis as we consider an approach based on the CASL specification language.

Techniques for Modelling and Verifying Large Scale Railway Interlockings [To appear in STTT]
(Phillip James, Faron Moller, Hoang Nga Nguyen, Markus Roggenbach, Steve Schneider and Helen Treharne. Accepted for publication in STTT)
Presents an approach towards modelling and verification using CSP||B. The paper presents abstractions which are included in the OnTrack tool and were developed jointly by all authors of the paper. As these abstractions have been implemented in OnTrack in a manner that is independent of the formal specification language used, they are presented briefly in Chapter 7.

1.6 Chapter Overview

The remainder of this thesis is outlined as follows:

Background Material:

Chapter 2 introduces, in some depth, the field of domain specific languages and their design. It considers the classical motivations for developing domain specific languages, along with the benefits they can bring to the more traditional application domain of programming languages. This chapter also explores various frameworks for creating domain specific languages and associated tools. It gives various industrial examples of domain specific languages and tools. Finally, the chapter concludes with a discussion of the Eclipse frameworks applied in our methodology.

Chapter 3 considers the railway domain and various domain specific languages and modelling approaches that have been designed for the railway domain. This includes a detailed look at both industrial practice within the railway domain and Bjørner's DSL which is used for illustration throughout the thesis.

Chapter 4 introduces the theoretical background on specification formalisms and institution theory. It presents the reader to both CASL and MODALCASL on the syntactical and semantic level. It also discusses the Heterogeneous Toolset (HETS) and the various theorem provers that we use for proof support throughout the thesis.

Methodology Construction:

In Chapter 5 we introduce our new institution for UML class diagrams and the comorphism we have defined into MODALCASL. We discuss the concepts of UML class diagrams that we capture, and also present the general construction of a pointed powerset institution for capturing dynamical aspects of systems. Finally, we illustrate both the institution and comorphism using Bjørner's DSL.

Continuing from our comorphism presentation, Chapter 6 shows how to extend the resulting CASL specifications to allow for verification. This chapter introduces and illustrates the use of domain specific property supporting lemmas for the railway domain. Such lemmas make verification of real world systems possible. This chapter concludes by presenting various verification results that were made possible through application of the supporting lemmas.

Chapter 7 brings together the results of the previous chapters into the OnTrack toolset. We present details on the implementation of OnTrack using the GMF development process and show how to implement model transformations for generation of formal specifications. We then discuss the ability of OnTrack to implement domain abstractions that are independent of the formal verification procedure.

In Chapter 8 we apply our methodology to the Invensys Rail Data Model. We show that our UML class diagram institution and comorphism to MODALCASL capture the required elements. We also show that the domain specific lemmas we introduced previously in Chapter 6 also apply to the Invensys Rail Data Model. This chapter ends by providing verification results for models formulated using this Data Model.

Conclusions:

Finally, in Chapter 9 we draw the thesis to a close by summarising our proposed design methodology. We give concluding remarks on applying the methodology and comment on future areas of interest towards extending the methodology.

Part I

Background Material

Chapter 2

Domain Specific Languages

Contents

Throughout all areas of Science, one can compare approaches that are general in principle or specific to the task at hand. A general principle gives a solution to several problems of a similar manner. Whereas a specific solution often solves the problem in a more comprehensive manner, but can be applied to significantly less problems. In this section, we consider these approaches in Computer Science where the differences are exemplified by general purpose languages (GPLs) and domain specific languages (DSLs) respectively. We focus on DSLs as these are of greater relevance to our work. In our discussion, we consider the usual characteristics of DSLs, how to design DSLs and comment on several particularly successful DSLs. We briefly discuss UML class diagrams which are commonly used for the definition of DSLs and finally, we consider several Eclipse [ecl] based tooling frameworks provided to support DSL design.

2.1 Introduction, History and Motivations

Domain specific languages (DSLs) [FP10, MHS05], also known as special purpose [Wex81] or little/mini languages [vDK98], are languages that have been designed and tailored for a specific application domain. Some examples of domain specific languages include Risla [AvDR95], Hancock [BFRS00] and the well known Backus Naur Form or BNF [Knu64]. Another example of such a language is the commonly used HTML [Gra95, RLHJ99], which was designed explicitly with webpage creation in mind. HTML has specific features such as *elements*, *tags* and *attributes* that allow the specification of structure within a web page. The advantages of having these domain specific features are apparent as HTML has become the de facto standard for webpage creation due to its expressiveness and ease of application. Here, when we speak about expressiveness, we refer to how easily a user of the language can construct and speak about the objects they desire. Expressiveness can often be explored by considering how intuitive various language constructs are. The disadvantage of domain specific languages are that they are often costly to design and require both domain and language design expertise.

The question of what constitutes a domain specific language is a complex one. Several attempts have been made to characterise exactly what constitutes a domain specific language [MHS05, FP10, Tah08]. Here we present an overview and combination of these attempts to classify such languages. The main features of domain specific languages are often classified as follows:

Expressiveness and Generality The main telling feature of whether a language is domain specific is if it trades expressiveness for generality. Expressiveness of a language covers aspects of how easy it is to express various, often domain specific, ideas using the constructs appearing in the language. Whereas generality tries to provide generic constructs that can be used together for many application domains. A language is often called domain specific if it provides tailored notations and constructs for a particular application domain. Such tailored notations and constructs allow programmers from the domain to easily express the things they require. For example, if we consider a language such as HTML, it can only be used to specify layout. It is not a programming language but a markup language used as input to a web renderer. Hence we could probably not write a program such as quick sort using HTML. Whereas a language such as C [KR88] is Turing complete, and could be used to create any computable function or program. HTML trades generality for expressiveness, allowing users to easily specify web page layout. Using C, a task such as text layout may be possible, but would not necessarily be straightforward given the constructs and notations provided by C.

Extension and Restriction Another telling feature of a domain specific language, is that the language is based on some existing language which has either been:

1. *extended* with extra features and constructs that are only applicable to the domain under consideration, or

2. *restricted* to a sub language allowing the re-use of tools and other features available for the language, but providing insurance that only certain 'restricted' programs may be created using the language.

A typical example of restriction of a language is Spark Ada [Bar97] which is a restricted version of Ada [Pyl85]. Spark Ada has been designed especially for the domain of critical software development and only contains those Ada constructs that are considered to be "safe". For example, there are harsh restrictions on concurrency. The benefits that come with this restriction are again visible as reasoning about the runtime behaviour of a

Spark Ada program is often easier than reasoning about an Ada program and also, any Spark Ada program can be compiled using any Ada compiler. Hence the development of new compilers for the Spark Ada language was not needed. Such re-use of tools often means that development costs for new domain specific languages can be reduced in comparison to designing a new language from scratch.

An example of extension, is the EBNF or Extended Backus Naur Form language [IEC96]. EBNF provides syntactic extensions to traditional BNF that allow the user to express repetition and optional statements more easily, for example through the use of regular expressions such as the * notation. Today, EBNF is often used over BNF.

Libraries and APIs The final and maybe not so convincing classification of a domain specific language, is a language that is provided by a programming library or API (Application Programming Interface) for an existing general purpose language. A good example of such a library, is the NAG library [HP88] which provides a collection of numerical algorithms. Such a library often defines functions that are relevant to a specific domain, and hence in turn allow domain engineers access to the features that they need without having to implement the features themselves. Taking this into account, we now consider the benefits and costs of designing a DSL completely from the drawing board. In the following section we provide arguments to clarify such a decision.

2.2 Why Develop DSLs

Given the wide range of general purpose languages available, and the often high costs involved in development, it is not always clear why a domain specific language should be developed. The following list, based on the arguments presented by Mernik et al. [MHS05], provides a compelling argument for domain specific language design.

Analysis and Verification: Developing a domain specific language allows aspects relating to verification, optimisation and existing processes to be taken into account. For example, domain specific knowledge could be incorporated into the language which in turn could help verification of programs created using the language. This is the approach we emphasise throughout this thesis.

Re-use: Domain specific languages are often developed with re-use in mind. That is, constructs are created to allow maximum re-use of ideas and concepts for the application domain. Krueger [Kru92] highlights re-usability as an advantage of domain specific languages. The inclusion of domain specific notations for re-use are often highly useful in the application of the language.

Readability: Definability of domain specific notations is sometimes out of the scope of existing languages. For example, special keywords may not be definable using an existing language. A typical example of this is that many programming languages do not allow the use of infix mathematical notation. For example, the expression

"Div(a,b)" can be defined, but the infix expression "a Div b" can often not be defined.

Productivity: Domain specific languages are also able to improve the software development cycle. For example, Hermans [HPvD09] shows that domain specific languages can ease the design and implementation phases of a project. Similarly having a domain specific language allows for easier maintenance as shown by Deursen and Klint [vDK98]. Here, due to domain specific languages, domain engineers are able to easily understand and maintain the existing programs.

Along with this list of advantages, surveys into the success of domain specific languages in industry are often conducted to show the improvement that can be obtained from developing a domain specific language. A particular example of such a survey is given by Hermans [HPvD09], where the impacts of a domain specific language called ACA.Net are studied. Here the application is a visual language used to build web services. The language was found to greatly improve both reliability of software and productivity when creating software in a multiple number of projects.

Given these advantages, it is obvious that when designed in the correct way, a domain specific language can be highly useful and often required in specific domains. For this reason, Heering and Sloane [MHS05] have tried to capture a general design process for developing such domain specific languages. Here, different 'design patterns' are given for all the stages of domain specific language design, beginning from the decision to design a new language through to the implementation of the language and corresponding tool support. To continue our discussion into domain specific languages, we consider some specific languages that have been successfully designed and applied to solve an issue within a particular domain.

2.3 Industrial Examples of Domain Specific Languages

In this section, we briefly review some existing domain specific languages, commenting on the main features of each. The purpose of this section is to give an insight into when domain specific languages can be useful, and how they provide these additional useful features. The application areas for the domain specific languages we present are wide and varied, demonstrating how vast the subject area of domain specific languages is. Some further sources for examples of domain specific languages include the surveys by Deursen and Klint [vDKV00], and Mernik et al. [MHS05].

RISLA (1995) RISLA or Rente Informatie Systeem Language [AvDR95] is a domain specific language for use within the banking domain. It is a language for designing interest rate based products, and was developed using the ASF+SDF meta environment [vdBvDH$^+$01]. A product description written in RISLA is eventually compiled into a COBOL implementation. The document discusses how RISLA captures key notations for product descriptions, and how it solves some discrepancies that were apparent before its use. Overall, the key features of RISLA include modularity and re-usability.

APOSTLE (1997) APOSTLE [Bru97] is a domain specific language created for parallel discrete event simulation. Parallel discrete event simulation, sometimes called distributed simulation, refers to the execution of a single discrete event simulation program on a parallel computer. The main motivation for creation of APOSTLE was that domain engineers, namely simulation practitioners, should not be required to have a detailed understanding of the underlying protocols involved in parallel discrete event simulation in order to apply the technology. The article outlines several rounds of implementation of the language, and spends time discussing how types in a domain specific language play an important role, especially to the semantics of the language. The results of creating the language are not only increased expressability for the domain, but also a fairly generic language for the domain.

Hancock (2000) The Hancock [BFRS00] domain specific language was designed for computing with customer data collected from telephone calls, for example call lengths or numbers dialled. The motivations for designing the language included that certain features, such as data traversal, could not be directly represented using libraries of general purpose languages. The language itself is eventual compiled down to C, but provides better data abstraction mechanisms than would be possible in C. These methods help when dealing with the vast amount of data involved in the telephone industry. The designers of the language comment that Hancock makes the programmers task much easier, and also that the language should greatly reduce errors [BFRS00].

ACA.NET (2009) ACA.NET or Avanade Connected architectures for .Net [HPvD09] is a domain specific language used to describe how to create web services using the Microsoft's .Net framework [TL03]. The language is currently in its fourth version. It aims to provide a highly reusable set of architecture components that help to accelerate the design process of applications. The ACA.NET domain specific language has been heavily analysed against success by Hermans et al. [HPvD09]. The results indicate that users of the language find it helps to improve reliability whilst also reducing development costs.

2.4 Domain Specific Language Design

In this section, we explore various platforms enabling the creation of DSLs. We concentrate our efforts on the ASF+SDF [Ber89, vdBvDH+01] language for aiding in the design of domain specific languages. Firstly we introduce ASF, then we study SDF. Next, we look at the combination of ASF+SDF and the meta environment [vdBvDH+01] that is provided for tool generation. Finally, we comment on some commercially available platforms such as MetaEdit+ [SLTM91, KLR96].

2.4.1 The Algebraic Specification Formalism

ASF or the algebraic specification formalism [Ber89], is a formalism for specifying abstract data types using initial semantics. It supports modularised specifications which can contain the following features: importing and parametrisation, hiding, overloading of functions, infix operators and positive conditional equations. A typical specification written in ASF can be viewed as a set of modules that describe algebras. Such a module includes a set of sorts and function declarations, i.e. the signature, a set of variable declarations allowing the user to define terms over the declared variables, i.e. the syntax, and finally a set of equations, i.e. the properties. In ASF, each module specifies an initial algebra. A simple ASF specification is given in Figure 2.1.

The main features of this module are as follows: The `exports` keyword represents sorts and functions that are exported by the module. These sorts and functions are then visible by any module that imports this module. Any sorts and functions declared outside this are only visible within the given module; the `sorts` keyword, declares that the following elements are sorts for use within the module; the `functions` keyword declares that the following lines are functions/operations of the module along with their profiles; the `variables` keyword declares variables to be used within the equations of the module; finally, the `equations` keyword declares equations/axioms that must hold for any algebras that satisfy the module. These equations are formed using the previously defined functions and variables.

An important feature that is highlighted by the module in Figure 2.1, is the initial algebra semantics used in ASF. That is, the module specifies the Booleans up to isomorphism. No formulae expressing the "no junk" and "no confusion" properties are needed due to the initial semantics. This is in contrast to the loose semantics employed, as we see in detail later, by the CASL specification language.

2.4.2 The Syntax Definition Formalism

SDF, or the Syntax Definition Formalism [Ber89, HHKR89], is a description language allowing users to define custom syntax. SDF is similar to the tools Lexx [LMB92] and Yacc [LMB92] which are commonly used for syntax definitions of programming languages. The advantage of SDF is that it is possible to combine SDF with many different programming or specification languages, the most popular combination is with ASF. The combination with ASF allows users to specify both the syntax and semantics for a specific language they wish to design. SDF itself, allows the definition of both lexical and context free syntax. We will briefly review SDF through the means of an example. For further reading on SDF the reader is referred to [Ber89, HHKR89]. Figure 2.2 shows a typical definition of syntax for the Booleans in SDF.
The `lexical syntax` section of the specification allows the usual white space escape characters to be used as part of the syntax for Booleans. Here, the lexical syntax specifies that spaces, carriage returns and line breaks can be used. The `context-free` section of the specification is concerned with defining the context-free grammar for the Booleans. The grammar defines terminal tokens `tt` and `ff`, and non terminal tokens `or`

```
module Bools
begin
   exports
      begin
         sorts Bool
         functions:
            tt: -> Bool
            ff: -> Bool
            and: Bool # Bool -> Bool
            or:  Bool # Bool -> Bool
      end

   functions
      not : Bool -> Bool
   variables
      a,b: -> Bool
   equations
   [1] not(tt) = ff
   [2] not(ff) = tt
   [3] or(a,b) = not(and(not(a),not(b)))
   [4] and(tt,a) = a
   [5] and(ff,a) = ff
end Bools
```

Figure 2.1: Specification of the Booleans in ASF.

and and. Further examples of SDF definitions can be found in [Ber89].

2.4.3 ASF+SDF

ASF+SDF [Ber89] is the combination of the Algebraic Specification Formalism with the Syntax Definition Formulation. It has been developed at the CWI (Centrum Wiskunde and Informatica) department in Amsterdam. The combination allows users to describe both syntactical and semantic details of the language which they are aiming to design. The aim of ASF+SDF is to aid in the creation of domain specific languages.

ASF+SDF comes with a powerful meta-environment [vdBvDH$^+$01] for prototypical tool creation. The ASF+SDF meta-environment can be described as "an interactive development environment for the automatic generation of interactive systems for constructing language definitions and generating tools for them" [vdBvDH$^+$01]. That is, the meta-environment provides the user with several features including: syntax editing tools for creating and editing ASF+SDF specifications; compilation of such ASF+SDF specifications into language tools such as language parsers, debuggers, editors and

```
module Bools
begin
    lexical syntax
      layout SPACE
      functions
        [ \t\n\r] -> SPACE
    context-free syntax
      sorts Bool
      functions
        tt -> Bool
        ff -> Bool
        Bool or Bool -> Bool
        Bool and Bool -> Bool
end Bools
```

Figure 2.2: Specification of the syntax for the Booleans in SDF.

compilers; finally, user interface extensionality via user defined features. Together, these features combine to allow users to define and generate integrated development tools for their given language definition.

An application of ASF+SDF which is of particular interest to us, is an application to Vital Processor Interlockings (VPI's) [GvVK95] as used by Dutch railways. In their work, Groote et al. analyse and verify safety properties against VPI's. The VPI's are programmed using Vital Logic Code (VLC). VLC is then modelled using a process algebraic approach. The main application of ASF+SDF was in creating several tools through prototyping. The prototypical tools created by the ASF+SDF meta-environment were translated down to more efficient tools using the C programming language. For example, a parser for translating VLC into propositional logic was developed using ASF+SDF. This then enabled the verification of the interlockings using SAT Solvers. The outcome of using ASF+SDF was mainly the reduction in time required to develop prototype tools in comparison with estimates based on not using ASF+SDF.

2.4.4 Commercial Platforms

We briefly comment on some of the commercial platforms that are available for domain specific language design. We refrain from a discussion of Eclipse based platforms at this stage, and discuss them in more detail later in Section 2.6.

DSL Tools: DSL Tools [CJKW07] is a toolkit provided by Microsoft for the design of domain specific languages within the Visual Studio development environment. It allows for the creation of graphical UML like languages which can be edited within the Visual Studio IDE. It also allows the production of various tool like code generators. For prototyping a DSL, DSL Tools provides a proprietary notation for meta modelling but lacks a mechanism for model to model transformations [SK03]. Interestingly, studies by Pelechano et al. [PAMC06] have found that developing a DSL with Eclipse based tools is easier than with DSL Tools, but the resulting development environment from creating a DSL with DSL Tools is easier for users to learn to use.

MetaEdit+: MetaEdit+ [SLTM91, KLR96] is one of the most popular commercially available DSL design tools. It allows users to put objects, with properties, onto a panel and connect objects with relationships. The tool-set includes a Diagram Editor, Object browser and the possibility for code generation. When designing a DSL within MetaEdit+ the user is not required to write any code. Instead, there are a series of editors that allow the user to create objects for the DSL and to attach graphical representations to them. Similarly, for each object the user can generate code. This code generation is completely open and the user is free to generate whatever they wish. The main element lacking within MetaEdit+ is a validation process for such produced code.

AToM^3: Finally, AToM^3 [LV02] is a tool providing features for meta modelling of domain specific languages. It allows users to create a meta model for the DSL using a built-in editor for a subset of UML class diagram features. Prolog [Wie96] constraints can be attached to elements of this diagram to constrain the features of the DSL. Finally, a visual syntax can be attached to elements of the class diagram. One of the main interesting features of ATOM^3 is that it allows users to create multiple viewpoints to one underlying domain specific language. These viewpoints are given in the form of projections of the underlying meta model of the DSL [PALG08].

2.5 UML Class Diagrams for DSL Descriptions

UML Class Diagrams [Obj11] 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 [Rail0] of our research partner 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. As a running example throughout the thesis, we use Bjørner's DSL for the railway domain [Bjø03]. Bjørner's classification for the Railway domain is illustrated using a UML class diagram in Figure 2.3. We will discuss the domain specific components of the diagram later in Chapter 3, here we discuss the UML class diagram constructs.

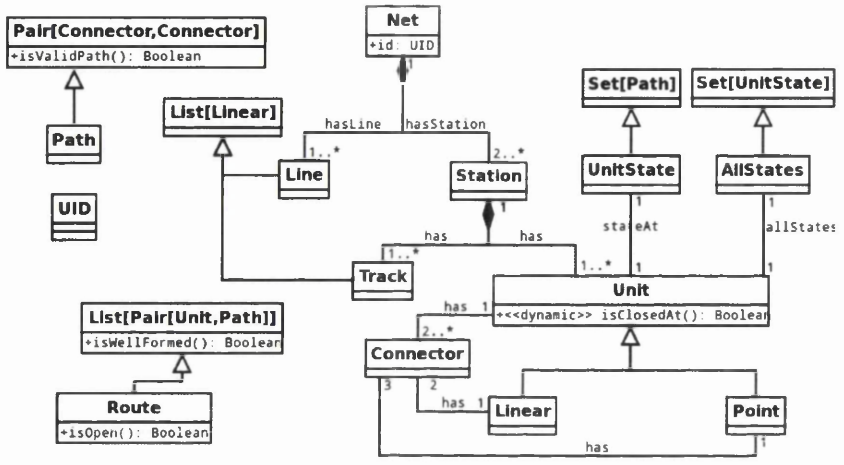


Figure 2.3: A UML class diagram for Bjørner's DSL for the railway domain.

2.5.1 Elements of Class Diagrams

The UML diagram in Figure 2.3 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 [Obj11]. Overall Figure 2.3 contains:

- Classes, represented by a box, e.g. Net, Unit, Station etc. These represent concepts in the railway domain.
- Properties, listed inside a class, e.g. id : UID in the class Net expresses that all Nets have an identifier of type UID (Unique Identifier).
- Generalisations, represented by an unfilled arrow head, e.g. Point and Linear are generalisations of Unit.
- Associations, represented by a line/arrow between two classes, e.g. the has link between Unit and Connector. Associations can have direction, and also multiplicities associated with them. The multiplicities on the has association between Unit and Connector can be read as: "One Unit has two or more connectors".
- Compositions, represented by a filled diamond, e.g. the hasLine composition for Net and Line, 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, e.g. the isOpen operation of type Boolean inside the Route class.

Such a class diagram can be thought of as describing all the components that can be found within a domain.

Here, we note that the query operations we consider throughout this work can be seen as parameterised properties. We also do not treat operations that can modify the state of an object.

2.6 Eclipse Frameworks: EMF, GMF and Epsilon

In this section, we discuss the main Eclipse 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) [SBMP08], the Graphical Modelling Framework (GMF) [Gro09] and Epsilon [KRPP13]. Each of these plugins are developed for Model Driven Engineering and Development [Ken02, BBG$^+$05] of domain specific languages.

2.6.1 The Eclipse Modelling Framework

Many people consider the core of a language to be it's 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 [SBMP08] 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 [SBMP08] which is a UML class diagram like language for describing meta models for DSLs. This model is stored using the XMI (XML based) 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.

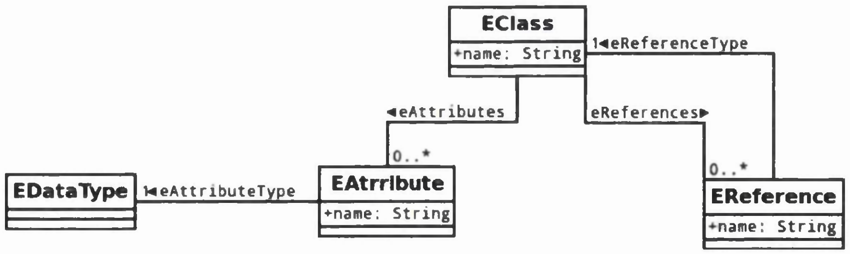
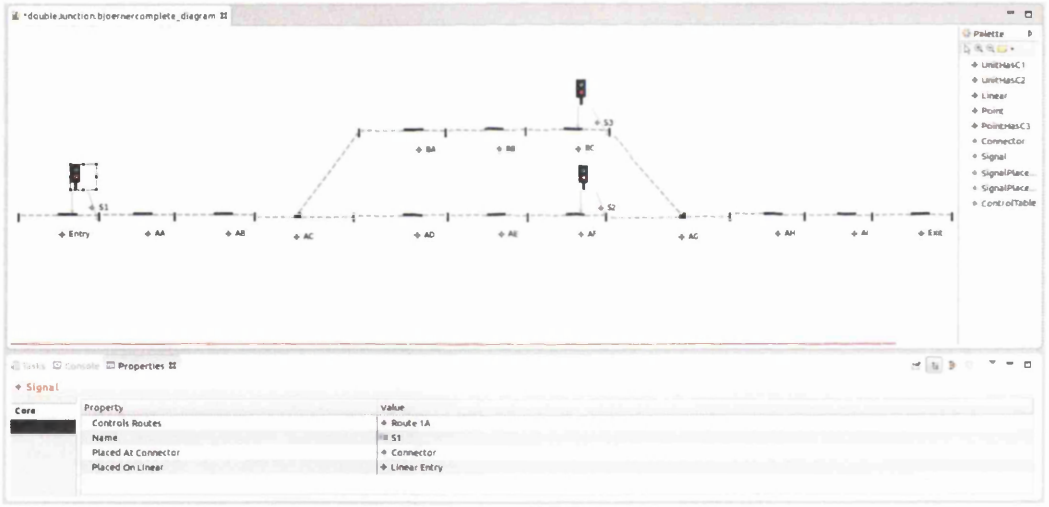


Figure 2.4: Ecore Meta Model for a simplified version of Ecore.

In Figure 2.4 we give a UML class diagram view of the Ecore model for a simplified version of Ecore itself. That is, as Ecore is nothing but a language to describe abstract syntax, Figure 2.4 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 2.5, showing the close relationship between Ecore and UML class diagrams.

2.6.2 The Graphical Modelling Framework

The GMF or Graphical Modelling Framework project [Gro09] 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 2.5. 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.



Property	Value
Controls Routes	Route 1A
Name	S1
Placed At Connector	Connector
Placed On Linear	Linear Entry

Figure 2.5: An example GMF Editor (OnTrack) 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.

2.6.2.1 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 2.6 shows the development process for a typical GMF editor.

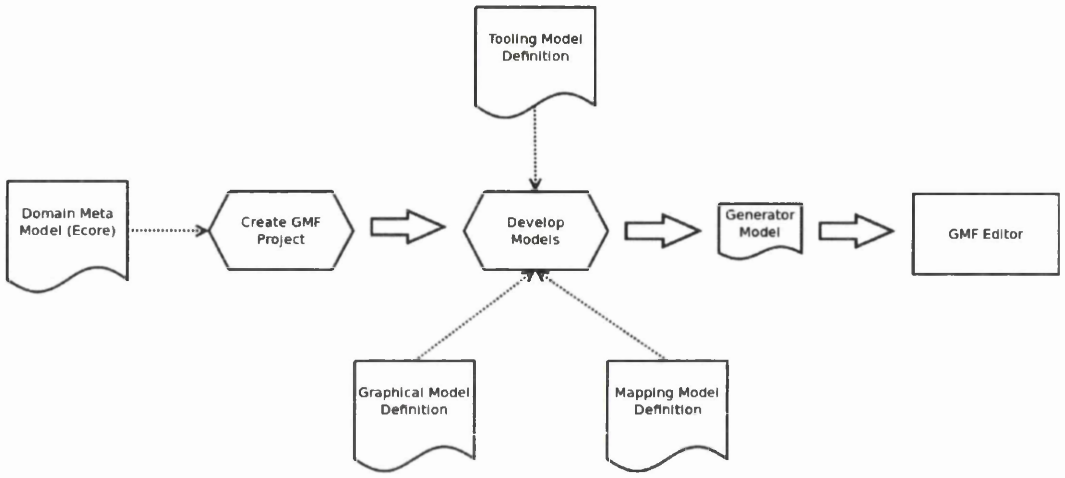


Figure 2.6: The steps involved in developing a GMF Editor.

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 2.5, 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. 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. The mapping model is used as a key element in deriving the main features of the 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 [Gro09] if required. Finally, this model can be used to generate a domain specific language editor similar to the one shown in Figure 2.5.

In Chapter 7, we return to the discussion of these various models for the design of the OnTrack toolset. Further details and examples of developing GMF editors can be found in [SBMP08, Gro09].

2.6.3 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 [SBMP08], JET [SBMP08] and Xpand [EFH$^+$04], we will concentrate our review on the technology we use in this thesis, namely Epsilon [KRPP13].

2.6.3.1 Epsilon

Epsilon, the extensible platform of integrated languages for model management [KRPP13], 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 2.4 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. Later, in Chapter 7, we will see how we use this type of transformation to implement abstractions for models.

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 Chapter 7, 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 [KRPP13] 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. [KRPP13].

Chapter 3

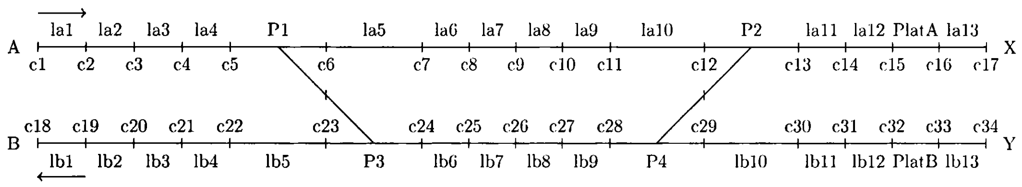
The Railway Domain and DSLs

Contents

In this chapter, we introduce the railway domain and several DSLs that have been developed for it. In particular, we discuss common terminology used by railway engineers and describe the typical design process for developing scheme plans and interlockings. We then explore some formalisations of the railway domain into domain specific languages, particularly the domain modelling undertaken by Dines Bjørner [BGP99, Bjø00, Bjø03, BCJ^{+}04, Bjø09]. Finally, we detail some ongoing and related work towards the development of DSLs for the railway domain.

3.1 Industrial Practice in the Railway Domain

In industry, companies such as our industrial partner, Invensys Rail [Inv13], undertake domain engineering with the aim to "describe all concepts, components and properties within the railway domain" [Rai10]. This modelling, for example, includes features such as rail topology (the graph underlying the railway), dimensions (e.g, where tracks are with regards to certain reference points), geography (e.g., gradient of a track), civil structures (buildings, bridges etc.), track equipment (signals, points etc.) and signalling (routes, speed restrictions etc.). Common across all such approaches for the railway domain is the informal notion of a *track plan* as a term to describe layouts of junctions and stations. Such a track plan is a combination of topological information and the conceptual abstraction of routes. Track plans are often developed as part of processes prescribed by Railway Authorities, such as Network Rail's *Governance for Railway Investment Projects* (GRIP) process. The first four phases of the GRIP process define



la1 la2 la3 la4 P1 la5 la6 la7 la8 la9 la10 P2 la11 la12 PlatA la13
A X
c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15 c16 c17
c18 c19 c20 c21 c22 c23 c24 c25 c26 c27 c28 c29 c30 c31 c32 c33 c34
B Y
lb1 lb2 lb3 lb4 lb5 P3 lb6 lb7 lb8 lb9 P4 lb10 lb11 lb12 PlatB lb13

Figure 3.1: A modified track plan from a London Underground station.

the track plan and routes of the railway to be constructed. While phase five – the detailed design – is contracted to a signalling company, such as Invensys, who choose appropriate track equipment, add control information to the track plan, and implement concrete control systems for running the railway. An example of such a track plan is shown in Figure 3.1.

The intended operation of the train station shown in Figure 3.1 is: (1) trains enter at A using track $la1$, they then proceed on the upper line to platform *PlatA* (i.e. taking route RAX); (2) alternatively, they pass over switch points $P1$ and $P3$ to the lower line and proceed to platform *PlatB* (i.e. taking route RAY); (3) trains from *PlatA* pass to the lower line using switch points $P2$ and $P4$ and leave using track $lb1$ at B (i.e. taking route RXB); (4) trains from *PlatB* stay on the lower line and leave using track $lb1$ at B (i.e. taking route RYB). These four pathways are called routes. Notice that it is a design decision which pathways form routes, as in the above track plan there is no route starting from A through points $P1, P3, P4$ and $P2$ to platform *PlatB*. A route can be assigned for use by a train (is "open"), if all tracks on the route are unoccupied and the relevant switch points are correctly set. Such a track plan is usually paired with a set of control and release tables [KR01] to form a scheme plan. This scheme plan gives details of the conditions required for availability of a route. For example, Figure 3.2 gives a typical control table for our example station. It prescribes that a given route can be assigned when all tracks in the "clear" column are not occupied by a train, and the points in the "normal" and "reverse" columns are set in those positions. The control table tells us concretely that route RAX can be assigned to a train when units la1, la2, la3, la4, P1, la5, …, la12, PlatA, la13 are unoccupied and points P1 and P2 are set to their normal positions, that is, allow trains to travel from left to right along the top line of Figure 3.1.

Route	Clear	Normal	Reverse
RAX	la1, la2, la3, la4, P1, la5, …, la12, PlatA, la13	P1, P2	
RAY	la1, la2, la3, la4, P1, P3, …, lb12, PlatB, lb13	P4	P1, P3
RXB	la12, la11, P2, P4, …, lb3, lb2, lb1	P3	P2, P4
RYB	lb12, lb11, lb10, P4, …, lb3, lb2, lb1	P3, P4	

Figure 3.2: Example control table.

The concrete operational setting and unsetting of points and routes is controlled by a so-called interlocking that is implemented based on this scheme plan. Interlockings provide a safety layer between the controller and the track, see Figure 3.3. In order to move a train, the controller issues a request to set a route. The interlocking uses rules and track information to determine whether it is safe to permit this request: if so, the interlocking will change the state of the track (move points, set signals, etc.) and inform the controller that the request was granted; otherwise the interlocking will not change the track state.

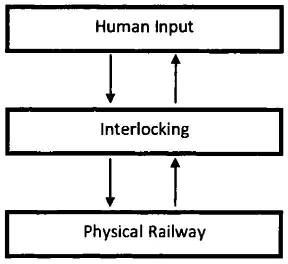


Figure 3.3: Interlocking control within the railway.

In this thesis, we do not consider the concrete implementation of such an interlocking. Verification of safety properties over concrete interlocking systems has been considered by several others [FH98, PGHD04, HP07, KMS08, Jam10]. We focus on a higher level of verification. That is, we consider verification of the design of Scheme Plans which contain the logic that is used for the implementation of interlockings.

3.2 Bjørner's DSL

In an academic setting, the process of identifying, classifying and precisely defining the elements of a domain has been coined as "Domain Engineering" by Dines Bjørner [Bjø09]. We now discuss Bjørner's classification [BGP99, Bjø00, Bjø03, BCJ$^+$04, Bjø09], i.e., DSL, for the Railway domain. Bjørner gives such a description using natural language via a narrative [Bjø03]. We explain this narrative over the following sections. The full narrative as given by Bjørner can be found in Appendix A.

Figure 3.4 shows the hierarchy of concepts for the static parts of Bjørner's DSL. A railway is a *Net*, built from *Station(s)* that are connected via *Line(s)*. A station can have a complex structure, including *Tracks*, *Switch Points* (also called points) and *Linear Units*. *Tracks* and *Lines* can only contain *Linear Units*. All *Unit(s)* are attached together via *Connector(s)*. Along with defining these concepts, Bjørner stipulates various well-formedness conditions on such a model, for example, "No two distinct lines and/or stations share units" or "Every line of a net is connected to exactly two, distinct stations" [Bjø03, Bjø00].

London's tube map, part of which is given in Figure 3.5, can easily be described with these terms: The shown part of the underground map consists of three *nets*, namely the District Line (green), the Circle Line (yellow), and the Central Line (red); these nets share several *stations* (Notting Hill Gate belongs to all three nets, whereas Queensway belongs only to the Central Line); there are *lines*, e.g. between Notting

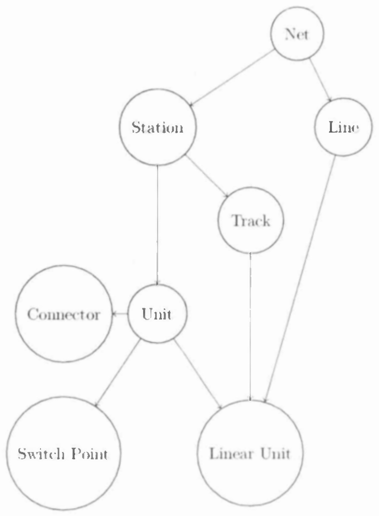


Figure 3.4: Static concepts from Bjørner's DSL.

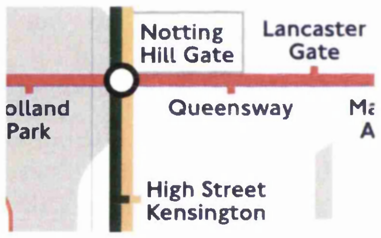


Figure 3.5: Excerpt from London's tube map.

Hill Gate and Queensway, some of which are shared between various nets. Similarly, Bjørner's approach contains the necessary terms to describe the track plan given in Figure 3.1. The whole track plan forms a *station*. This *station* contains all elements such as *switch points* $p1, p2, p3$ and $p4$ along with *linear units* $la1, la2, \ldots, lb13$ and *connectors* $c1, c2, \ldots, c34$. Platforms $PlatA$ and $PlatB$ can be considered as linear units. Finally, sequences of *linear units* within the station form *tracks*, for example, the sequence of linear units $la1, la2, la3, la4$ before point $P1$ can be seen as forming a track.

From this point on, we shall only consider the elements of Bjørner's DSL which we require to model the track plans we are interested in. Therefore, some of the conditions stipulated by Bjørner do not apply. For example, the track plan given in Figure 3.1 is open ended on the left hand side, whereas Bjørner considers only closed track plans. Hence, axioms regarding closed networks, for example the condition "all nets must contain two stations" (leaving no open lines), do not apply to our models.

3.2.1 Dynamics in Bjørner

Bjørner's DSL gains dynamics by attaching a state to each unit [Bjø03]. Each unit can be in one of several *states* at any given time. Such a state is represented using a set of paths, where a single path is given by a pair of distinct connectors (c, c'). A path expresses that a train is allowed to move along a given unit in the direction from connector c to connector c'. For example, considering Figure 3.1, the single direction of travel along the top line would mean that unit $la1$ could, at a given time, be open along the path $(c1, c2)$. As a second example, Figure 3.6 shows the states that Bjørner considers possible for a switch. To combine a unit and a path across that unit, Bjørner introduces the notion of a *unit path pair* which simply forms pairs from units and a valid path across them.

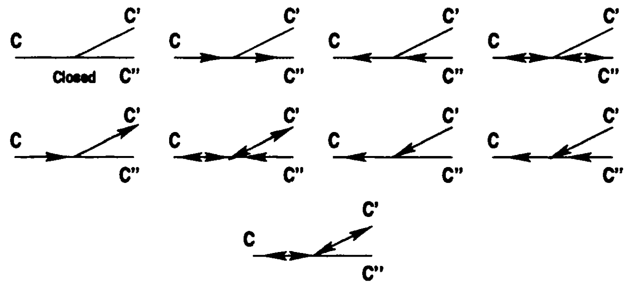


Figure 3.6: Valid states of a switch according to Bjørner's modelling [Bjø03].

Trains are not an explicit part of Bjørner's DSL. Instead, Bjørner describes the concept of a *route*, which is a dynamic "window" around a train. Concretely, routes are lists of connected units and paths along them. That is, routes are well formed lists of unit path pairs, where well formed means that adjacent units are in fact adjacent elements of the track plan, and that the direction dictated by adjacent paths in the list "match". For example, the route from A to X of Figure 3.1 would be captured as the following list:

$$[(la1, (c1, c2)), (la2, (c2, c3)), \ldots, (p1, (c5, c6)), \ldots (la13, (c16, c17))],$$

whereas:

$$[(la1, (c2, c1)), (la2, (c2, c3)), \ldots, (p1, (c5, c6)), \ldots (la13, (c16, c17))]$$

would not be a valid *route*, as the neighbouring connectors for paths across $la1$ and $la2$ do not "match". Finally, a route is considered to be *open* for use if all units within the *route* are *open* for the correct *path* across them.

Bjørner stipulates that a route can be dynamically changed over time using a movement function that, for a given time, gives the set of assigned routes. This movement function can extend or shrink a route by adding or removing a unit at one or both of its ends. Train movements can then be modelled using this function. For example, Figure 3.7 illustrates how the movement of a train through (an initial section of) the railway given in Figure 3.1 could be modelled using a movement function that is dependent on time.

Time	Routes Assigned	New Operation Type
0	[]	extension
1	[(la1,(c1,c2))]	extension
2	[(la1,(c1,c2)), (la2,(c2,c3))]	extension and reduction
3	[(la2,(c2,c3)), (la3,(c3,c4))]	extension and reduction
4	[(la3,(c3,c4)), (la4,(c4,c5))]	extension and reduction
5	[(la4,(c4,c5)), (P1,(c5,c6))]	extension
6	[(la4,(c4,c5)), (P1,(c5,c6)), (la5,(c6,c7))]	reduction
7	[(P1,(c5,c6)), (la5,(c6,c7))]	extension and reduction
8	[(la5,(c6,c7)), (la6,(c7,c8))]	...
...	...	...

Figure 3.7: Example of modelling train movements using Bjørner's DSL.

Later in Chapter 6 we will comment on some alterations to the modelling of routes by Bjørner to fit with common industrial practice within the UK.

3.3 Sample Formalisation in RSL

Bjørner has formalised his narrative in (the algebraic part of) RSL [RAI93], for example, see [Bjø00, Bjø03]. Here we show the basic elements of this encoding.

Bjørner declares a sort symbol for each of the static concepts: *Net*, *Line*, *Station*, *Track*, *Unit* and *Connector*:

```
type Net, Linear, Switch, Track, Unit, Connector
```

Then units are classified into "switch points" and "linear units" via test functions:

```
is_Linear: Unit -> Bool
is_Switch: Unit -> Bool
```

Here, Bool is the built-in datatype of booleans in RSL. Observer functions are then declared which give, e.g., for a specific net, the set of all stations belonging to it:

```
obs_Stations: Net -> Station-set
```

Here, -set is the built-in constructor of sets in RSL. This signature allows Bjørner to express the static well-formedness conditions we discussed earlier. For example the condition that "no two distinct stations share units", is formulated in RSL as:

$$\forall n : Net; s, s' : Station \bullet$$
$$s, s' \in obs\_Stations(n) \ \wedge \ s \neq s' \Rightarrow obs\_Units(s) \ \cap \ obs\_Units('s) = \{\}.$$

In a similar manner, Bjørner encodes the dynamics of railways, for example, paths are specified in RSL as:

```
type Pair' = C x C
     Path = {|(c,c'):Pair . not(c = c')|}
```

Here, we can see that the condition stating that a path is formed over two distinct connectors is captured during the construction of the type. Using such paths, Bjørner encodes routes as sequences of pairs of units and such paths:

```
type Route' = (Unit x Path)*
     Route = {|r:Route' . wellformed(r)|}
```

Again, the well-formedness check over routes is encoded into the type construction using the auxiliary predicate $wellformed : Route'$. This predicate is then encoded as:

$$\begin{aligned}
&wellformed(r) = len(r) > 0 \ \wedge \ \forall i : Nat \ \bullet \ i \in inds(r)\\
&\qquad\qquad \text{let}(u,(c,c')) = r(i) \text{ in } (c,c') \ \in obsStates(u) \ \wedge \ i+1 \in inds \ r\\
&\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow \text{let}(\_,(c'',\_)) = r(i+1) \text{ in } c' = c''
\end{aligned}$$

where $inds(r)$ simply gives the indexes of elements in the list r, $obsStates(u)$ gives the observable states of a unit u and "_" represents an unnamed variable. This predicate basically states that a route is well formed, if for all unit path pairs in the route, the path is a possible path of some state of the unit, and all "neighbouring" connectors within the paths of the route are identical. The full presentation of Bjørner's DSL encoded using RSL can be found in [Bjø03]. From this point onward, we shall use CASL rather than RSL as our chosen specification language. This is mainly due to the greater level of proof support that is available for CASL in the form of the HETS environment [MML07].

3.4 Related Work on DSLs and the Railway Domain

At this point, we reflect on some further DSL based approaches for the railway domain and consider how the work in this thesis differs.

3.4.1 RCSD Language

The Railway Control Systems Domain language (RCSD) is a domain specific language for the design of railway control systems developed by Kirsten Mewes [Mew09, Mew10]. Part of this work consists of a very detailed domain analysis in a natural language style that is similar to Bjørner. This analysis forms the basis of RCSD. The developed domain specific language is motivated by a model-driven engineering approach [Ken02] to system design, and has been developed using both MetaEdit+ [KLR96] and UML [Obj11]. There is also a variant of RCSD, known as RCSD-UML, which is developed completely using the UML framework. The language incorporates knowledge from domain engineers about the domain into its static and dynamic semantics as well as using common domain notation for its concrete syntax. This ensures that modelling is simplified for the every day domain engineer. The approach by Mewes also considers the development of test suites over models created in the proposed DSL. To this end, Mewes presents details on the selection of suitable tests based on knowledge of the domain. This allows domain specific constraints to be included into the models and ensures that certain validation checks around correct functionality can be checked at the model level, before software has been developed. Finally, the work also contributes to compiling a collection of domain specific properties that one may consider for validation and verification within the railway domain. Even though the work presented by Mewes is very detailed in the design of the RCSD domain specific language, relatively little is considered with respect to verification. Within this thesis, we concentrate much less on the design of DSLs from the domain perspective, leaving this to the domain engineer. However, we do show how domain knowledge captured within a domain specific specification can be exploited to allow for successful automatic verification.

3.4.2 A Domain Specific Construction Framework

Work by Haxthausen [HP07] has explored the development of a domain specific framework for automated construction and verification of railway control systems. The framework consists of a three tiered approach:

DSL for specification: The top layer is a domain specific language for use by domain engineers. This domain specific language allows specification of railway control systems. The framework also provides static checking of properties such as well-formedness over these specifications.

Model generation: The second tier of the framework is model generation from a given specification. The framework includes a generator that automatically generates a model control program based on the specification given by the design engineer. At this level, the framework provides links to model checkers allowing for bounded model checking of various safety properties.

Code compilation: Finally, the framework allows for code generation from a given system model. This is once again automated, and the framework provides a third level of verification in that checks are performed over the generated code to ensure certain properties are maintained through the generation process.

The work presented is very complete for the railway domain, however it is developed specifically for the railway domain, and as such, does not comment on applications to other domains. In this thesis, we provide a generic, systematic, methodology for DSL construction for any given domain and apply it to the railway domain for illustration purposes.

3.4.3 SafeCap DSL and Toolset

Finally, the SafeCap toolset [IR12b] provides a tooling platform that supports reasoning about railway capacity while ensuring system safety. We note, that this toolset has been developed in a collaboration between the author of this thesis and members of the Computer Science department at Newcastle University. For its base, the toolset uses the SafeCap DSL [IR12a]. This DSL attempts to capture, at a suitable level of abstraction, track topology, route and path definitions and signalling rules. By design, the SafeCap DSL is extensible in that one can dynamically define further attributes for all the predefined language elements and these are then automatically reflected in the SafeCap Toolset. Overall, the toolset aims to allow signalling engineers to design stations and junctions, to check their safety, and to evaluate the potential improvements in capacity whilst applying various alteration patterns that change the railway scheme plans. The platform uses a combination of Event B specification with model checking and SMT solving to verify (bounded) system safety. The toolset also includes several plug-ins that evaluate various capacity parameters.

In a similar manner to the methodology we present (in fact, motivated by the presented methodology), the tool uses Eclipse technology, including the EMF and GMF frameworks. However, in contrast to the presented methodology, the DSL used in the tool has been defined independently by the developers of the tool. This is in part due to the considerations towards capacity analysis at which the tool is focused. Also, such an approach has led to the developed DSL being focused towards Event B analysis. Thus differing to our aim, see Chapter 7, of decoupling the DSL from the formal specification language and in turn allowing tooling environments to be openly extendable for other specification languages.

Chapter 4

Specification Formalisms, Institutions and Proof Support

Contents

In this chapter, we introduce the CASL and MODALCASL specification languages along with the framework of institutions and institution comorphisms. We firstly present each language on the syntactic level, illustrating how the features of each language can be applied to model a small railway example. We then present the technical details of both languages on the semantic level. For this, we give details of the institutions that underlie CASL and MODALCASL and present an institution comorphism for translating MODALCASL specifications to CASL specifications. Finally, we discuss the Heterogeneous Toolset (HETS) and the various theorem provers that are incorporated into this toolset to provide proof support for CASL.

4.1 The Common Algebraic Specification Language

The Common Algebraic Specification Language [Mos04b, BM04], known as CASL, is a specification formalism developed by the CoFI initiative [Mos97] throughout the late 1990's. The aim of the CoFI initiative was to design a *Common Framework for Algebraic Specification and Development* in an attempt to create a *de facto* platform for algebraic specification. The main motivation for the CoFI initiative came from the existence of a number of competing algebraic specification languages with varying levels of tool support and industrial uptake. Examples of such languages include *ACT-ONE/ACT-TWO* [EM85, CEW93], *ASF* [Ber89], *ASF-SDF* [Kli93, DHK96], *OBJ* [GWM+93], *CafeOBJ* [DF98], *Larch* [GH93], *RSL* [RAI93], *Extended ML* [KST97] and *Maude* [MFS+07]. Overall, CASL incorporates many of the features of these other languages including partial functions, sub-sorting and sort generation constraints. In this section, we shall informally introduce and describe the main features of CASL. A full formal description is given in Section 4.3.

4.1.1 Modelling Constructs

CASL allows one to model systems at various levels of abstraction by providing various levels of specification. These include basic (unstructured) specifications, structured specifications and architectural specifications. Basic specifications form the atoms of structured specifications. Structured specifications support the building of complex specifications from simpler specifications, allowing for separation of concerns when modelling large systems. Finally, architectural specifications allow for specifying software architectures, for example, modules within a software package. In this section, we give a brief overview of the constructs CASL provides for writing basic specifications. A more detailed description of the CASL language and its semantics can be found in The CASL Reference Manual [Mos04b]. We illustrate the CASL constructs using specifications for time. The concept of time is central to several of the railway domain elements in Dines Bjørner [Bjø00, Bjø09] domain modelling. These examples are based on work presented in [JR11, O'R12].

In general, a CASL *basic specification* consists of a set of declarations of symbols (i.e. names) for *sorts* (the data types of the specification), symbols and corresponding profiles for *operations* (total and partial functions on the sorts), symbols and profiles for *predicates* (relations on the sorts), and a set of *axioms* and *constraints* which restrict the interpretations of the declared symbols. The axioms are formula's expressed in first order logic with equality, with the usual connectives and universal quantifiers; furthermore, they may make assertions regarding definedness (e.g. of the results of partial functions) and subsorting. We begin with an explanation of these constructs using the following relatively simple specification of discrete time.

```
spec TIME =
    sort  Time
    ops   0 : Time;
          suc : Time → Time;
          pre : Time →? Time
    pred  __<=__ : Time × Time
end
```

The above CASL specification has the name TIME. It specifies that there is exactly one sort symbol, namely *Time*, a constant function symbol (a function symbol with no arguments) 0, a total function symbol *suc* from sort *Time* to sort *Time*, a partial function symbol *pre* from sort *Time* to sort *Time* and a predicate symbol $\_\_<=\_\_$ over *Time* $\times$ *Time*. The intention of these declared symbols is for the *suc* function to increment time by a single time unit, *pre* to decrement time by a single time unit and $\_\_<=\_\_$ to be the binary predicate for less than or equal using infix notation over time. For this specification, the axiom set Φ is empty.

Within CASL, each specification gives rise to a formal object called the signature of the specification. We will go into the formal details of such signatures later in Section 4.5.1. For now, we say that a CASL signature $\Sigma = (S, TF, PF, P, \leq_S)$ is a five tuple where

- S is a set of sort symbols,
- TF and PF are families of sets of total and partial function symbols respectively, each with a profile giving its argument and target sorts,
- P is a family of predicate symbols, each with a profile giving the sorts it ranges over, and
- $\leq_S$ is a reflexive and transitive sub-sort relation.

Using the above specification of TIME as an example, we have the signature $\Sigma_{\text{TIME}} = (S, TF, PF, P, \leq_S)$ where $S = \{Time\}$, $TF = \{0, suc\}$, $PF = \{pre\}$ and $P = \{\_\_ <= \_\_\}$ $\leq_S = \{(Time, Time)\}$.

Remark 4.1 As overloading is not present in this example we have omitted profiles on families of sets for ease of reading. Further details on this are given in Section 4.5.1.

Semantically, each CASL signature has a class of many-sorted models, or algebras, associated with it. The class of models for a given signature Σ is denoted by mod(Σ). Each model gives an interpretation to each of the symbols in a signature. Each sort symbol is interpreted using a non-empty carrier set. Each constant is assigned a particular element of the corresponding carrier set. Function symbols are interpreted by functions over the correct carrier sets. Finally, predicates are interpreted by sets of elements, from the correct carrier set, for which the predicate holds. Given our signature for the TIME specification, one possible model M is given by:

$$\begin{array}{lcl} M_{Time} & = & \{1\} \\ (0)_M & = & 1 \\ (suc)_M(1) & = & 1 \\ (pre)_M(1) & = & 1 \\ (\_\_ <= \_\_)_M & = & \{\} \end{array}$$

where we use the notation M_{Time} for the carrier set of the sort *Time* in model M and similarly for functions and predicates.

Remark 4.2 We have omitted the interpretation of the subsort relation as it does not play a role in this example. This relation is covered in more detail in Section 4.5.2.

This model (often called the one point model) interprets the sort *Time* with a singleton carrier set containing the element 1. As there is only one element in this carrier set, there is no choice for the interpretation of the symbol 0 and also the function symbol *suc* which must map 1 to 1, as both are total functions. As, *pre* is partial, there is a choice between it being undefined for the element 1 or defined as it is in the above model. Similarly, there is a choice in the interpretation of the predicate symbol $\_\_<=\_\_$ where we have chosen to state that it is empty, or false for all elements of the carrier set. Here, we also note that CASL does not allow for empty carrier sets in models, hence the presented model is minimal in terms of the size of the carrier set. The presented model is somewhat counter intuitive to what one would expect from the signature, however it is indeed a valid model. The following is a somewhat more intuitive model:

$$\begin{array}{lcl} N_{Time} & = & \mathbb{N} \\ (0)_N & = & 0 \\ (suc)_N(n) & = & n+1 \\ (pre)_N(n) & = & \begin{cases} n-1 & \text{if } n>0 \\ \textit{undefined} & \text{otherwise} \end{cases} \\ (\_\_<=\_\_)_M & = & \{(n,m) \in \mathbb{N} \times \mathbb{N} \mid n \leqslant m\} \end{array}$$

This model interprets Time using a model of the natural numbers. There are infinitely many more models in the model class for the signature of specification TIME.

To control the number of models a specification has, CASL allows the use of axioms. The specification TIME does not contain any axioms, and hence the formal meaning of all declared symbols is "loose", that is, the symbols can currently take on any possibly meaning within a given model. This in turn means that model class of the specification TIME is the same as the model class of the signature of TIME. However, if we wish to control the meaning of say $\_\_<=\_\_$ then we can add to the above specification the following axiom: $\forall\ n : Time \bullet 0 <= n$. This axiom states that the constant *0* is, in some sense, the first possible time. Adding this axiom leads to the following specification TIME':

spec TIME' =
    sort *Time*
    ops $0 : Time;$
        $suc : Time \rightarrow Time;$
        $pre : Time \rightarrow ?\ Time$
    pred $\_\_<=\_\_ : Time \times Time$
    $\forall\ n : Time \bullet 0 <= n$
end

We denote the set of axioms within a specification by Φ. Interestingly, we can see that the specification *Time'* including this axiom has less models than the specification TIME, as the model M given above is a model of TIME but not a model of TIME'. This is because M does not exhibit the property stated by the axiom. Similarly, we can see that the model N is a model of both the given specifications.

Each specification $SP = (\Sigma, \Phi)$ is a combination of a signature Σ and a set of axioms Φ. As we have seen, like signatures, a specification gives rise to a class of models, namely all the models of its signature which satisfy all of its axioms, that is,

$$\mathbf{Mod}(SP) = \{M \in |\,\mathrm{mod}(\Sigma)| \mid M \models \Phi\}$$

where $M \models \Phi$ if and only if $M \models \varphi$ for all $\varphi \in \Phi$.

Finally, notice that our specification TIME' still contains loosely specified elements, for example there are no axioms concerning the function *pre*. This means that we do not have a single element model class, but instead have many models. We note that we could further specify time, by adding further axioms, to ensure we only have a single model (up to isomorphism). However this loose specification of time is enough for us to demonstrate several features of CASL and, as we shall see later Section 4.7, is enough to carry out some proofs. This follows a general approach to specification, in that the intended model is contained within the model class, but there are also many non standard models within the model class. However, these non standard models often still exhibit the properties one is interested in. The advantage of such an approach is twofold. Firstly, it requires less of a specification effort, and secondly it can lead to better results for proof support, as it provides a smaller axiomatic base for automatic provers to consider.

4.1.2 A Miniature Railway Specification

We now look at a slightly more interesting example of a CASL specification using a subset of the DSL proposed by Bjørner [Bjø00, Bjø09]. This example illustrates some further features of CASL, including subsorting. The following specification captures the notion of linear units, and switches (or points) form Bjørner's DSL.

```
spec RAILWAYELEMENTS =
     TIME'
then sorts Connector;
           Linear, Switch < Unit
     pred  __has_connector__ : Unit × Connector
     ops   c1, c2 : Unit →? Connector;
           c3 : Switch →? Connector
     sort  State
     op    __state_at__ : Unit × Time →? State;
     ∀ s : Switch; l : Linear • ¬ s = l                                    %(1)%
     ∀ u : Unit • ¬ c1(u) = c2(u)                                          %(2)%
     ∀ s : Switch • ¬ c3(s) = c1(s) ∧ ¬ c3(s) = c2(s)                      %(3)%
     ∀ l : Linear; c : Connector •
                  l has_connector c ⇔ c = c1(l) ∨ c = c2(l)               %(4)%
     ∀ s : Switch; c : Connector •
                  s has_connector c ⇔ c = c1(s) ∨ c = c2(s) ∨ c = c3(s)  %(5)%
end
```

Firstly, this specification, named RAILWAYELEMENTS, imports the specification TIME' that was declared earlier. This is via the use of the keyword **then**. This import is one of many forms of specification structuring mechanisms provided by CASL. It makes the symbols and axioms available in TIME' available and applicable within RAILWAYELEMENTS, more details can be found in [Mos04b].

Next, RAILWAYELEMENTS declares four sorts, namely *Connector*, *Linear*, *Switch* and *Unit*. Interestingly, here we see another feature of CASL in the form of sub-sorting using the < symbol. *Linear* and *Switch* are sub-sorts of *Unit*. This tells us that all *Linear* units and *Switch*es are also basic *Unit*s. Semantically, as we shall see later, subsorting is done through injection functions. The < symbol is a shorthand for $s < t$. It declares an injection function from sort s to sort t, a partial projection function from sort t to sort s, and a membership predicate that tests whether elements of sort t have a counterpart in sort s. Here we note that explicit casting is usually not required in formulae as the tools associated with CASL often deal with it automatically. Linear units and switches can be placed and connected together using connectors, from the sort *Connector*. Following Bjørner's DSL, linear units and switches are connected using a single connector between them.

The next element declared is a predicate *__has_connector__* which allows units and connectors to be related: linear units have two connectors and switches have three connectors according to Bjørner [Bjø03]. These restrictions are not present from the definition of the predicate, but will be specified later using axioms. Along with this predicate, there are several total functions. The first two functions, *c1* and *c2*, give us access to the two connectors of a unit, and hence can be applied to both linear units and switches. The *c3* function gives us access to the third connector of a switch and hence cannot be applied to linear units.

Next, we define the sort *State*, this will be used to capture the state of a unit. For now, we leave this sort (along with various other elements) loose. To go with this, we then define an operation, *__state_at__* allowing us to observe the state of a unit at a given point in time.

Finally, we add some axioms to control the interpretations of the symbols we have introduced so far. The axioms state:

1. Linear units and switches are not the same.
2. Connectors on units are distinct.
3. The extra connector for a switch is distinct.
4. Coupling the predicate *__has_connector__* with *c1* and *c2* for units.
5. Coupling the predicate *__has_connector__* with *c3* for switches.

4.1.3 A Note on Free Types

In the above RAILWAYELEMENTS specification, we have modelled *State* as a loose sort. This allows us, if needed, to extend this specification and add more details about the elements of the *State* sort. For example, we may consider the state of a unit to be the position of the physical tracks, e.g. a switch could be in *normal* or *reverse* position, and a linear unit could always be in a fixed position. Alternatively, we could consider *State* to be similar to that of the unit state considered by Bjørner, that is, a pair of connectors dictating which directions along a unit a train can travel. Taking the former as an example, we can use the CASL construct of a free type [Mos04b] to tightly specify what elements are within the *State* sort:

free type *State* ::= *normal* | *reverse* | *moving*

This definition states that there are three constants *normal*, *reverse* and *moving* (between normal and reverse), each representing a state of a unit. The free type also forces all interpretations of *State* to have exactly three carrier set elements, one for each constant. That is the free type captures the notions of "no junk" and "no confusion". Thus, in this case, the free type is just shorthand for expressing that there are three constants, all carrier set elements are accessible via these three constants and finally, that these three constants must have different values.

4.1.4 Modelling a Junction

In the railway domain it is quite common to find so called junctions. This construction is so common that it is useful to model this as a separate specification both for methodological reasons, i.e., reuse of code and theorem proving support (see our work in [JR11]). To allow such structuring, CASL provides several general structuring constructs, namely:

- Naming: Specifications can be named, such as the above TIME' specification.
- Parameters: Named specifications can have parameters that are in fact other specifications that can be instantiated.
- Union: Taking the union of two specifications.
- Extension: Extending a specification with another possibly partial specification. Note that this differs to union, as the second specification can refer to elements declared in the first specification.
- Renaming: Finally, the symbols of a specification can be renamed.

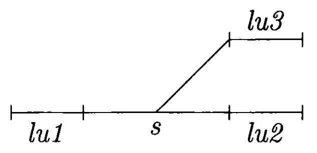


Figure 4.1: A typical railway junction.

To illustrate some of these features, we now show how a junction can be modelled with the railway elements we have introduced so far. To do this, we make use of so called generic specifications [Mos04b]. These allow for the parametrization discussed above. Figure 4.1 illustrates our modelling, where s is a switch and *lu1*, *lu2* and *lu3* are associated linear units.

The following is a generic CASL specification that models such a junction:

spec JUNCTION
    [**ops** *lu1*, *lu2*, *lu3* : *Linear*; *s* : *Switch*
    • ¬ *lu1* = *lu2*
    • ¬ *lu2* = *lu3*
    • ¬ *lu1* = *lu3*]
given RAILWAYELEMENTS =
    • *c2*(*lu1*) = *c1*(*s*)
    • *c2*(*s*) = *c1*(*lu2*)
    • *c3*(*s*) = *c1*(*lu3*)
end

The generic specification JUNCTION has two main parts, namely:

- a *formal parameter* that is a CASL specification. They are written within square brackets, like above. Such a formal parameter declares the elements needed for use later in the body of the specification. For example, a junction requires three linear units (*lu1*, *lu2* and *lu3*) and a switch (s). In our case, the formal parameter also contains axioms that ensure the linear units are all unique. These axioms can be thought of as assumptions over the formal parameter. From a semantic point of view, there may be more linear units and switches in a given actual parameter as these are not captured in the formal parameter using a free type.

- a body which is a partial CASL specification that can use the symbols from the formal parameter. Within this, we declare three axioms that connect the units from the formal parameter in the correct manner.

It also uses the keyword **given** that allows the formal parameter to make use of the symbols available in RAILWAYELEMENTS.

Finally, such a generic specification can be instantiated by providing an actual parameter. For example, we can instantiate the junction specification with the following:

spec JUNCTIONPARAM =
    RAILWAYELEMENTS
then ops *lu1*, *lu2*, *lu3* : *Linear*;
        s1 : *Switch*
    • ¬ *lu1* = *lu2*
    • ¬ *lu1* = *lu3*
    • ¬ *lu2* = *lu3*
end

Any axioms within the formal parameter of the generic specification must be implied by the actual parameter during instantiation. For example, any actual parameter to the junction must guarantee the conditions over the uniqueness of the linear units it provides. This is clearly the case with the above actual parameter.

4.1.5 Implied Properties

The last CASL feature we introduce is that of an implied property [Mos04b]. Such a property or lemma is a first-order formula which should hold for the whole model class of a specification. For example, if we consider the generic junction specification given in the previous section, we may wish to check that there are two distinct connectors for linear unit *lu*1. This should be the case even though we have not specified the connectors. In CASL this can be expresses using the keywords **then %implies**:

then %implies
- $\exists$ *cn1*, *cn2* : *Connector*
  - $\neg$ *cn1* = *cn2* $\wedge$ *lu1 hasConnector cn1*
    $\wedge$ *lu1 hasConnector cn2*

This allows us to extend the specification adding proof obligations that should be implied by the previous axioms in the specification. At this point, new signature elements can not be introduced. The keyword **%implies** is a special CASL comment [Mos04b] that indicates to tools that the axioms in the extension should be implied by the already specified axioms. We will see later in Section 4.7 that this will set up proof obligations to be proven. We also discuss tool support for proving such an implication in Section 4.7.

This concludes our presentation of CASL, the semantic details for CASL are discussed later in Section 4.5. For information on other constructs available in CASL, we refer to [Mos04b, BM04].

4.2 ModalCASL

MODALCASL is an extension of the CASL specification language with constructs for Modal Logic [HC96, Mos04a, MC10]. In this section, we motivate the use of MODALCASL for modelling dynamical aspects of systems, especially within the railway domain. To do this, we mainly consider the use of rigid and flexible operations and predicates as provided by MODALCASL [Mos04a]. Later in Section 4.6 we will consider the semantic details of MODALCASL.

4.2.1 Dynamics in Railways

So far in this chapter, we have introduced a relatively small specification for railway components. Even with this small selection, we have seen that some components of the railway change over time, whereas others do not. For example, the operation *__state_at__* : *Unit* $\times$ *Time* $\rightarrow$? *State* gives, for a given unit, the state it is in at a particular time.

Where time has been specified as a particular sort. Modal Logic [HC96] deals with the concept of Kripke structures [HC96] where there is the view of different worlds with accessibility relations (or modalities) between worlds. For such worlds and the relations between them, one can consider what properties one would like to hold. For example, we can think of such worlds representing different points in time, with the accessibility relation between worlds being the advancement of time. ModalCasl allows us to take such a view, where each world is specified by a specification, and modalities can be used to describe how elements of this specification change between these worlds. The following specification illustrates how ModalCasl provides us with such modal constructs, namely modalities and the key words flexible and rigid, for naturally modelling this kind of dynamical concept.

logic Modal

spec ModalExample =
  modality *time* {}
  ...
  rigid pred *__hasConnector__* : *Unit* × *Connector*
  rigid ops *c1*, *c2* : *Unit* → *Connector*;
      c3 : *Switch* → *Connector*
  flexible op *__state_at*: *Unit* → *State*
end

In the above specification, we can see the use of the ModalCasl keywords **rigid** and **flexible**. Semantically, the interpretation of rigid is that any predicates or operations marked as rigid remain the same throughout all worlds. Considering our example, we can see that the predicate*__hasConnector__* and its corresponding operations *c1*, *c2* and *c3* are marked as rigid, and thus stay the same throughout all time. This naturally models the topological aspects of a track plan perfectly, as they (up to building new railway lines) do not change over time. However, we can see that the operation *__state_at* has a changed profile that has become unary over the sort Unit, and has been marked as **flexible**. This marking means that the value returned by the *__state_at* operation is dependent on the current world, or in our view, the current time.

4.2.2 Modal Properties

In addition to extending Casl with new signature elements, ModalCasl also extends the property building connectives of Casl. Two new connectives are provided, namely: $\langle M\rangle\varphi$ and $[M]\varphi$ where M is some modality sort and φ some property. Both these connectives have the usual modal logic interpretation, that is:

- $\langle M\rangle\varphi$ is interpreted as φ must hold "in some" world that is reachable via the modality M in one step.
- $[M]\varphi$ is interpreted as φ must hold "in every" world that is reachable via the modality M in one step.

The semantic details of this extension are given in Section 4.6.3. For information on how ModalCasl extends Casl, we refer to [Mos04a]. In Chapter 5 we will see further how this natural modelling of system dynamics fits well with the capture of UML class diagrams describing DSLs. We will also see that it is possible to translate ModalCasl specifications into equivalent Casl specification in Section 4.6.4. These two steps not only allow for the natural capturing of DSLs, but also for the strong proof support that is available for Casl to be available for verification purposes over such DSLs.

4.3 Institutions and Institution Comorphisms

The theoretical framework of an institution, as introduced by Goguen and Burstall [GB92], is based on category theory [ML98] and is used for describing logics. Institutions were devised to be used as an abstract framework for capturing the intuitive structure of a logical system, including the syntax, semantics and satisfaction notions within a logic. Informally, an institution consists of a collection of signatures with signature morphisms. Then for each signature there is a collection of sentences, models and a satisfaction relation between the sentences and models such that the satisfaction condition holds. The satisfaction condition captures the idea that "truth is invariant under notational change". Formally, following Mossakowski [Mos02], we define an institution I as:

Definition 4.3 (Institution) An institution $I = (\mathrm{SIGN}^I, \mathrm{sen}^I, \mathrm{mod}^I, \models^I)$ where:

- SIGN^I is the signature category.
- $\mathrm{sen}^I : \mathrm{SIGN}^I \to \mathrm{SET}$ is the sentence functor, where SET is the category where objects are sets and morphisms are total functions between sets.
- $\mathrm{mod}^I : (\mathrm{SIGN}^I)^{op} \to \mathrm{CAT}$ is the model functor, where CAT is the category where objects are categories and morphisms are functors between categories.
- $\models^I_\Sigma \subseteq |\,\mathrm{mod}^I(\Sigma)| \times \mathrm{sen}^I(\Sigma)$ is the satisfaction relation, for each $\Sigma : \mathrm{SIGN}^I$,

such that the satisfaction condition holds: for every signature morphism $\sigma : \Sigma \to \Sigma'$ in SIGN^I,

$$\mathrm{mod}^I(\sigma)(M') \models^I_\Sigma \varphi \iff M' \models^I_{\Sigma'} \mathrm{sen}^I(\sigma)(\varphi)$$

holds for every sentence $\varphi \in \mathrm{sen}^I(\Sigma)$ and for every Σ'-model $M' \in |\,\mathrm{mod}^I(\Sigma')|$.

Remark 4.4 We omit the index I when it is clear from the context. Similarly, for notational convenience, we introduce the common shorthands [Mos02] and write $\sigma(\varphi)$ for $\mathrm{sen}^I(\sigma)(\varphi)$ and $M'|_\sigma$ for $\mathrm{mod}^I(\sigma)(M')$. For example, with these shorthands, the satisfaction condition for an institution becomes: for each signature morphism $\sigma : \Sigma \to \Sigma'$ in SIGN,

$$M'|_\sigma \models_\Sigma \varphi \iff M' \models_{\Sigma'} \sigma(\varphi)$$

for each Σ'-model $M' \in |\,\mathrm{mod}^I(\Sigma')|$ and Σ-sentence $\varphi \in \mathrm{sen}^I(\Sigma)$.

The category SIGN^I contains a collection of signatures and signature morphisms mapping symbols between signatures in a compatible manner. The functor $\mathrm{sen}^I : \mathrm{SIGN}^I \to \mathrm{SET}$ maps each signature $\Sigma \in |\mathrm{SIGN}^I|$ to the set of sentences $\mathrm{sen}^I(\Sigma)$ over the signature Σ. It also maps each signature morphism $\sigma : \Sigma \to \Sigma'$ to the sentence translation function $\mathrm{sen}^I(\sigma) : \mathrm{sen}^I(\Sigma) \to \mathrm{sen}^I(\Sigma')$ that translates sentences formed over Σ to sentences formed over Σ'. The functor $\mathrm{mod}^I : (\mathrm{SIGN}^I)^{op} \to \mathrm{CAT}$ maps each signature $\Sigma \in |\mathrm{SIGN}^I|$ to the category $\mathrm{mod}^I(\Sigma)$ of Σ-models and model morphisms. It also maps each signature morphism $\sigma : \Sigma \to \Sigma'$ to the reduct functor $\mathrm{mod}^I(\sigma) : \mathrm{mod}^I(\Sigma') \to \mathrm{mod}^I(\Sigma)$. This reduct functor reduces models over the signature Σ' to models over the signature Σ. Similarly, model morphisms are reduced to model morphisms between reduced models. Here, notice that morphism composition is reversed as the mod^I functor is contravariant. Finally, the satisfaction condition ensures that the satisfaction relation is preserved across translation of sentences and reducts of models. The overall situation is captured neatly in Figure 4.2.

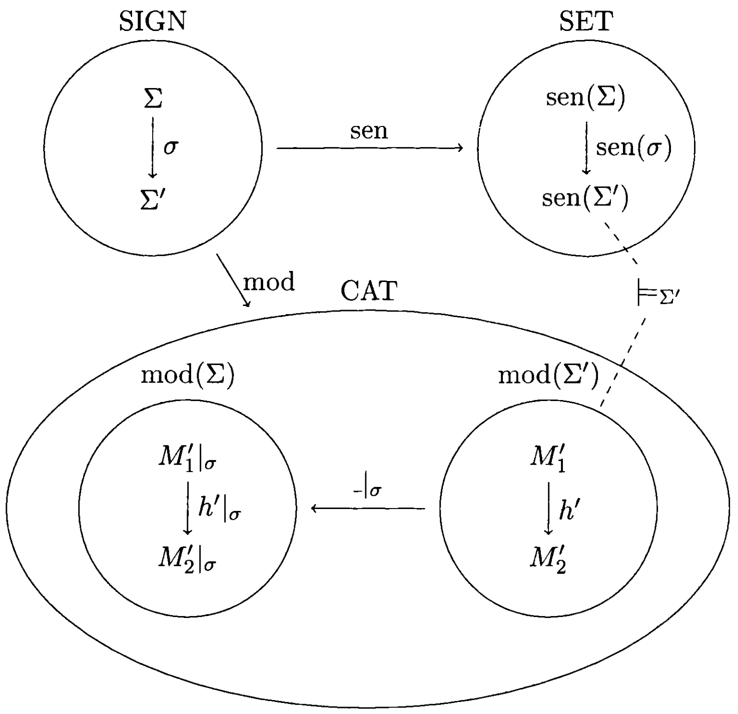


Figure 4.2: Diagram representing the notion of an institution [O'R12].

Given the notion of an institution, one can consider what it means to map between two institutions. The notion of an institution representation [Mos02] or co-morphism between two institutions allows signatures, sentences and models to be translated between institutions. Informally, given two institutions I and J, an institution co-morphism consists of a translation of I signatures to J presentations, a translation of I sentences to J sentences and a contravariant translation of J models to I models such that a translated model satisfies a sentence if, and only if, the original model satisfies the translated sentence. Formally, following Mossakowski [Mos02] we define an institution comorphism as:

Definition 4.5 (Institution Comorphism) Given institutions I and J, an institution co-morphism $\mu = (\Phi, \alpha, \beta) : I \to J$ consists of:

- a functor $\Phi : \text{SIGN}^I \to \text{Pres}^J$,
- a natural transformation $\alpha : \text{Sen}^I \to \text{Sen}^J \circ \Phi$, and
- a natural transformation $\beta : \text{Mod}^J \circ \Phi^{op} \to \text{Mod}^I$, and

such that the following representation condition is satisfied for all $\Sigma \in \text{Sign}^I$, $M' \in \text{Mod}^J \circ \Phi^{op}$ and $\varphi \in \text{Sen}^I$:

$$M' \models^J_{\text{Sig}(\Phi(\Sigma))} \alpha(\varphi) \iff \beta_\Sigma(M') \models^I_\Sigma \varphi.$$

The functor Φ intuitively maps signature of one institution into signatures of another (where axioms can be used to control certain interpretations and hence, the functor maps to presentations). The natural transformation α maps objects and morphisms of the sentence category of the first institution into objects and morphisms of the sentence category of the second institution, taking into account the signature map Φ. Next, the natural transformation β maps the models and model morphisms of the second institution to corresponding models and model morphisms of the first institution again taking into account the signature map Φ. Finally, the satisfaction condition ensures that the above mapping preserves truth of formulae within models.

As examples of institutions, we consider the *SubPCFOL*$^=$ institution that underlies CASL along with the institution for MODALCASL. We then present an institution comorphism from MODALCASL into CASL.

4.4 The *PCFOL*$^=$ Institution

We now present the *PCFOL*$^=$ (partial first order logic with sort generation constraints and equality) institution based on Mossakowski's presentation [Mos02]. This is the institution that forms the basis for *SubPCFOL*$^=$ that underlies CASL. In our presentation, we omit several proofs, and refer the reader to the work of Mossakowski for these details [Mos02].

4.4.1 *PCFOL*$^=$ Signatures

Firstly, we define the signature elements of *PCFOL*$^=$:

Definition 4.6 (*PCFOL*$^=$ Signature) A *PCFOL*$^=$ *signature* $\Sigma = (S, TF, PF, P)$ consists of:

- a set S of sort symbols,

- two $S^* \times S$-sorted families, $TF = (TF_{w,s})_{w \in S^*, s \in S}$ of *total function symbols* and $PF = (PF_{w,s})_{w \in S^*, s \in S}$ of *partial function symbols*, such that $TF_{w,s} \cap PF_{w,s} = \emptyset$ for each $(w, s) \in S^* \times S$,
- a family $P = (P_w)_{w \in S^*}$ of *predicate symbols.*

Remark 4.7 For notational convenience, we write $f : w \to s \in TF$ for $f \in TF_{w,s}$, $f : w \to ?\ s \in PF$ for $f \in PF_{w,s}$ and $p : w \in P$ for $p \in P_w$. Also, given a finite string $w = \langle s_1, \dots, s_n \rangle$ and sets $M_{s_1}, \dots, M_{s_n}$, we write M_w for the Cartesian product $M_{s_1} \times \dots \times M_{s_n}$.

Given two signatures $\Sigma = (S, TF, PF, P)$ and $\Sigma' = (S', TF', PF', P')$, we define the notion of a signature morphism as:

Definition 4.8 (*PCFOL*$^=$ Signature Morphism) A *PCFOL*$^=$ *signature morphism* $\sigma : \Sigma \to \Sigma'$ consists of:

- a map $\sigma^S : S \to S'$,
- a map $\sigma^F_{w,s} : TF_{w,s} \cup PF_{w,s} \to TF'_{\sigma^{S^*}(w), \sigma^S(s)} \cup PF'_{\sigma^{S^*}(w), \sigma^S(s)}$ preserving totality, for each $w \in S^*, s \in S$, and
- a map $\sigma^P : P_w \to P_{\sigma^{S^*}(w)}$.

Finally, defining identities and composition in the obvious way results in the category of *PCFOL*$^=$-signatures.

4.4.2 *PCFOL*$^=$ Models

Given a *PCFOL*$^=$ signature $\Sigma = (S, TF, PF, P)$, we define a model of this signature as:

Definition 4.9 (*PCFOL*$^=$ Model) A *PCFOL*$^=$ Σ-*model* M consists of

- a non-empty carrier set M_s for each $s \in S$,
- a partial function $(f_{w,s})_M$ from M_w to M_s for each $f \in PF_{w,s}$,
- a total function $(f_{w,s})_M$ from M_w to M_s for each $f \in TF_{w,s}$,
- a relation $(p_w)_M \subseteq M_w$ for each $p \in P_w$.

Remark 4.10 For notational convenience, we write f_M for partial/total functions $(f_{w,s})_M$ and for predicates we write p_M for $(p_w)_M$.

Given two models M, N, we define the notion of a model morphism as:

Definition 4.11 (*PCFOL*$^=$ Model Morphism) A *PCFOL*$^=$ Σ-*homomorphism* $h : M \to N$ is a family of functions $h = (h_s : M_s \to N_s)_{s \in S}$ such that:

- for all $f \in TF_{w,s} \cup PF_{w,s}$ and $(a_1, \ldots, a_n) \in M_w$ such that $(f_{w,s})_M(a_1, \ldots, a_n)$ is defined and where $w = \langle s_1, \ldots, s_n \rangle$, it holds that $h_s((f_{w,s})_M(a_1, \ldots, a_n)) = (f_{w,s})_N(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$, and similarly,
- for all $p \in P_w$ and $(a_1, \ldots, a_n) \in M_w$, where $w = \langle s_1, \ldots, s_n \rangle$, it holds that $(a_1, \ldots, a_n) \in (p_w)_M$ implies $(h_{s_1}(a_1), \ldots, h_{s_n}(a_n)) \in (p_w)_N$.

Finally, defining identities and composition in the obvious way results in the category of *PCFOL⁼* Σ-models.

We now consider the notion of model reducts for *PCFOL⁼* models. Let $\sigma : \Sigma \to \Sigma'$ be a *PCFOL⁼* signature morphism and M' be a Σ'-model, model reducts are defined as:

Definition 4.12 (*PCFOL⁼* Model Reduct) Then the *reduct* $M'|_\sigma$ of M' is the Σ-model M with:

- $M_s := M'_{\sigma^S(s)}$ for all $s \in S$,
- $(f_{w,s})_M := (\sigma^F_{w,s}(f))_{M'}$ for all $f \in TF_{w,s} \cup PF_{w,s}$, and
- $(p_w)_M := (\sigma^P_w(p))_{M'}$ for all $p \in P_w$.

Similarly, given a *PCFOL⁼* Σ'-homomorphism $h' : M' \to N'$, we define its reduct as:

Definition 4.13 (*PCFOL⁼* Model Morphism Reduct) The *reduct* of $h'|_\sigma : M'|_\sigma \to N'|_\sigma$ is given by $(h'|_\sigma)_s := h'_{\sigma^S(s)}$ for each $s \in S$.

Finally, once again defining identities and composition in the obvious way, gives us the functor mod.

4.4.3 *PCFOL⁼* Sentences

We now consider sentences for *PCFOL⁼*. Given a *PCFOL⁼* signature $\Sigma = (S, TF, PF, P)$, we define:

Definition 4.14 (Variables) Variables over Σ are given by an S-sorted, pairwise disjoint family $X = (X_s)_{s \in S}$.

Definition 4.15 (Terms) The set of terms $T_\Sigma(X)_s$ of sort $s \in S$, over Σ with variables in X is given by the least set satisfying:

- $x \in T_\Sigma(X)_s$, if $x \in X_s$, and
- $f_{w,s}(t_1, \ldots, t_n) \in T_\Sigma(X)_s$, if $t_i \in T_\Sigma(X)_{s_i}$ $(i = 1 \ldots n)$, $f \in TF_{w,s} \cup PF_{w,s}$, and $w = \langle s_1, \ldots, s_n \rangle$.

Definition 4.16 (Atomic Formulae) The set of atomic formulae, $AF_\Sigma(X)$ with variables in X is given by the least set satisfying:

- predicates: $p_w(t_1, \ldots t_n) \in AF_\Sigma(X)$, if $t_i \in T_\Sigma(X)_{s_i}$, $p_w \in P_w$, and $w = \langle s_1, \ldots, s_n \rangle \in S^*$,
- existential equations: $t_1 \overset{e}{=} t_2 \in AF_\Sigma(X)$, if $t_1, t_2 \in T_\Sigma(X)_s, s \in S$,
- strong equations: $t_1 = t_2 \in AF_\Sigma(X)$if $t_1, t_2 \in T_\Sigma(X)_s, s \in S$, and
- definedness assertions: $def\, t \in AF_\Sigma(X)$, if $t \in T_\Sigma(X)$.

Definition 4.17 (First Order Formulae) The set of first-order Σ-formulae, $FO_\Sigma(X)$, with variables in X is given by the least set satisfying:

- atomic formulae: $AF_\Sigma(X) \subseteq FO_\Sigma(X)$,
- falsity: $F \in FO_\Sigma(X)$,
- conjunction: $\varphi \wedge \psi \in FO_\Sigma(X)$, if$\varphi, \psi \in FO_\Sigma(X)$,
- implication: $\varphi \Rightarrow \psi \in FO_\Sigma(X)$, if $\varphi, \psi \in FO_\Sigma(X)$, and
- universal quantification: $\forall x : s \bullet \varphi \in FO_\Sigma(X)$, if $\varphi \in FO_\Sigma(X \cup \{x : s\}), s \in S$

Remark 4.18 We use the usual abbreviations $\neg\varphi$ for $\varphi \Rightarrow F$, $\varphi \vee \psi$ for $\neg(\neg\varphi \wedge \neg\psi)$, T for $\neg F$ and $\exists x : s \bullet \varphi$ for $\neg\forall x : s \bullet \neg\varphi$. We also omit brackets where possible.

PCFOL$^=$ includes the notion of sort generation constraints, that is, that some set of sorts is generated by some set of functions. A sort generation constraint over a signature Σ can be formally captured as:

Definition 4.19 (Sort Generation Constraint) A sort generation constraint over a signature Σ is given by a triple $(\overset{\bullet}{S}, \overset{\bullet}{F}, \theta)$, where $\theta : \overline{\Sigma} \to \Sigma$ for some signature $\overline{\Sigma} = (\overline{S}, \overline{TF}, \overline{PF}, \overline{P})$, $\overset{\bullet}{S} \subseteq \overline{S}$ and $\overset{\bullet}{F} \subseteq \overline{TF} \cup \overline{PF}$.

Here, the sort generation constraint contains a signature morphism component that is required for them to be translated along signature morphisms without conflicting with the satisfaction condition. We can now define the Σ-sentences for *PCFOL*$^=$.

Definition 4.20 (Sentences) A *PCFOL*$^=$ Σ-sentence is a closed first order formula over Σ or a sort generation constraint over Σ.

Next, to define sentence translation along a signature morphism, we firstly need to consider what is means to translate a variable set. Given a signature morphism $\sigma : \Sigma \to \Sigma'$ and a variable set X over Σ, we can obtain the variable set $\sigma(X)$ over Σ' by $\sigma(X)_{s'} := \bigcup_{\sigma^S(s)=s'} X_s$. This translation can be extended to atomic and first-order formulae via an inductive definition over the structure of the formulae [Mos02]. Finally, the translation of a sort generation constraint $(\overset{\bullet}{S}, \overset{\bullet}{F}, \theta)$ is the sort generation constraint $(\overset{\bullet}{S}, \overset{\bullet}{F}, \sigma \circ \theta)$.

Considering the sentence translation we have outlined above, it is straightforward to see that translation along the identity signature morphism is the identity, and that translation along a composition of two signature morphisms is the composition of the sentence translations along the individual signature morphisms. Hence, this gives us the functor sen.

4.4.4 The *PCFOL*$^=$ Satisfaction Relation

In order to define satisfaction of sentences we first define term evaluation. We note that variable valuations are total, but the value of a term given a variable assignment may be undefined. This could arise thanks to the application of a partial function during evaluation of a term. Given a *total variable valuation* $\nu : X \to M$ we define term evaluation as:

Definition 4.21 (Term Evaluation) The *term evaluation* $\nu^\sharp : T_\Sigma(X) \to ?M$ is inductively defined by:

- $\nu^\sharp_s(x) := \nu(x)$ for all $x \in X_s$ and all $s \in S$.
- $\nu^\sharp_s(f_{w,s}(t_1, \ldots, t_n)) :=$
$$\begin{cases} (f_{w,s})_M(\nu^\sharp_{s_1}(t_1), \ldots, \nu^\sharp_{s_n}(t_n)) & \text{if } \nu^\sharp_{s_i}(t_i) \text{ is defined } (i = 1 \ldots n) \text{ and} \\ & (f_{w,s})_M(\nu^\sharp_{s_1}(t_1), \ldots, \nu^\sharp_{s_n}(t_n)) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$
for all $f \in TF_{w,s} \cup PF_{w,s}$ and $t_i \in T_\Sigma(X)_{s_i}$ $(i = 1 \ldots n)$, where $w = \langle s_1, \ldots, s_n \rangle$.

We can now define the satisfiability of a first-order formula $\varphi \in FO_\Sigma(X)$ with respect to a valuation $\nu : X \to M$:

Definition 4.22 (Satisfaction $\models$) Satisfaction of a first-order formula $\varphi \in FO_\Sigma(X)$ is given by:

- $\nu \models p_w(t_1, \ldots, t_n)$ iff $\nu^\sharp(t_i)$ is defined $(i = 1 \ldots n)$ and $(\nu^\sharp(t_1), \ldots, \nu^\sharp(t_n)) \in (p_w)_M$.
- $\nu \models t_1 \overset{e}{=} t_2$ iff $\nu^\sharp(t_1)$ and $\nu^\sharp(t_2)$ are both defined and equal.
- $\nu \models t_1 = t_2$ iff $\nu^\sharp(t_1)$ and $\nu^\sharp(t_2)$ are either both undefined, or both are defined and equal.
- $\nu \models \mathit{def}\, t$ iff $\nu^\sharp(t)$ is defined.
- not $\nu \models F$.
- $\nu \models \varphi \wedge \psi$ iff $\nu \models \varphi$ and $\nu \models \psi$.
- $\nu \models \varphi \Rightarrow \psi$ iff $\nu \models \varphi$ implies $\nu \models \psi$.
- $\nu \models \forall x : s \bullet \varphi$ iff for all extended valuations $\zeta : X \cup \{x : s\} \to M$ (i.e., valuations where it holds that $\zeta(y) = \nu(y)$ for all $y \in X \setminus \{x : s\}$) we have $\zeta \models \varphi$.

$M \models \varphi$ holds for a many-sorted Σ-model M and a first-order formula φ, iff for all variable valuations ν (into M) we have $\nu \models \varphi$.

A sort generation constraint $(\overset{\bullet}{S}, \overset{\bullet}{F}, \theta)$ is satisfied in a Σ-model M, if the carriers of $M|_\theta$ of the sorts in $\overset{\bullet}{S}$ are generated by the function symbols in $\overset{\bullet}{F}$, that is, for every sort $s \in \overset{\bullet}{S}$ and every value $a \in (M|_\theta)_s$, there is a $\overline{\Sigma}$-term t containing only function symbols from $\overset{\bullet}{F}$ and variables of sorts not in $\overset{\bullet}{S}$ such that $\nu^\sharp(t) = a$ for some assignment ν into $M|_\theta$.

This concludes our presentation of *PCFOL*$^=$. Full details including a proof of the satisfaction condition for *PCFOL*$^=$ are presented by Mossakowski [Mos02]. We now consider *SubPCFOL*$^=$ which extends *PCFOL*$^=$ to an institution with sub-sorting.

4.5 The *SubPCFOL*$^=$ Institution

We now present the *SubPCFOL*$^=$ (partial first order logic with sort generation constraints, equality and sub-sorting) institution which underlies CASL. Our presentation is again based on Mossakowski's work [Mos02].

4.5.1 *SubPCFOL*$^=$ Signatures

The *SubPCFOL*$^=$ institution extends *PCFOL*$^=$ signature with the notion of subsorting. To do this, signatures from *PCFOL*$^=$ are extended with a reflexive and transitive subsort relation.

Definition 4.23 (*SubPCFOL*$^=$ Signature) A *SubPCFOL*$^=$ *signature* $\Sigma = (S, TF, PF, P, \leq_s)$ consists of:

- a *PCFOL*$^=$ signature, and
- a reflexive and transitive subsort relation $\leq_S$ over the set S of sorts.

As this added subsort relation is not antisymmetric, it allows for the definition of isomorphic sorts via injections. Also note, that $\leq_S$ can be extended in a pointwise manner to sequences of sorts. We can then define overloading on subsorts:

Definition 4.24 (Overloading Relations) Let $f : w_1 \to s_1, f : w_2 \to s_2 \in TF \cup PF$. Then $f : w_1 \to s_1 \sim_F f : w_2 \to s_2$ iff there exist $w \in S^*, s \in S$ such that $w \leq_S w_1, w \leq_S w_2, s_1 \leq_S s$, and $s_2 \leq_S s$. Let $p : w_1, p : w_2 \in P$. Then $p : w_1 \sim_P p : w_2$ iff there exists $w \in S^*$ such that $w \leq_S w_1$ and $w \leq_S w_2$.

A signature morphism is nothing but a *PCFOL*$^=$ signature morphism that preserves $\leq_s$, $\sim_F$ and $\sim_P$, i.e. we have:

Definition 4.25 (*SubPCFOL*$^=$ Signature Morphism) A *SubPCFOL*$^=$ signature morphism $\sigma : \Sigma \to \Sigma'$ is a *PCFOL*$^=$ signature morphism that satisfies:

- $s_1 \leq_S s_2$ implies $\sigma^S(s_1) \leq_{S'} \sigma^S(s_2)$ for all $s_1, s_2 \in S$ (preservation of $\leq_S$),
- $f : w_1 \to s_1 \sim_F f : w_2 \to s_2$ implies $\sigma^F_{w_1,s_1}(f) = \sigma^F_{w_2,s_2}(f)$ for all $f \in TF \cup PF$ (preservation of sim_F), and
- $p : w_1 \sim_P p : w_2$ implies $\sigma^P_{w_1}(p) = \sigma^P_{w_2}(p)$ for all $p \in P$ (preservation of sim_P).

Next, the standard way to deal with subsorting on the model level, is to attach to *SubPCFOL*$^=$ signatures both injection and projection functions, and an element-hood predicate. This can be accomplished by formulating a functor $\hat{\ }$: *SubPCFOL*$^=$-signatures $\to$ *PCFOL*$^=$-signatures.

Definition 4.26 ($\hat{\ }$ Functor) Given a *SubPCFOL*$^=$ signature $\Sigma = (S, TF, PF, P, \leq_S)$ we associate to it a *PCFOL*$^=$ signature $\hat{\Sigma} = (\hat{S}, \hat{TF}, \hat{PF}, \hat{P})$ that extends Σ with:

- a total *injection* function symbol $\texttt{inj}_{s,s'} : s \to s'$ for each pair of sorts $s \leq_S s'$,
- a partial *projection* function symbol $\texttt{pr}_{s',s} : s' \to ?s$ for each pair of sorts $s \leq_S s'$, and
- a unary *membership* predicate symbol $\in^s_{s'}: s'$ for each pair of sorts $s \leq_S s'$.

Given a signature morphism $\sigma : \Sigma \to \Sigma'$, we extend it to a signature morphism $\hat{\sigma} : \hat{\Sigma} \to \hat{\Sigma}'$ by mapping the injections, projections and memberships in $\hat{\Sigma}$ to the corresponding injections, projections and memberships in $\hat{\Sigma}'$.

Remark 4.27 From this point onward, we drop the subscripts on the injection and projection functions if they are clear from the context. We also write $t \in s$ instead of $\in^s_{s'} (t)$ if s' is clear from the context.

This now allows a relatively simple definition of *SubPCFOL*$^=$ models.

4.5.2 *SubPCFOL*$^=$ Models

SubPCFOL$^=$ models can now be captured as *PCFOL*$^=$ models of an extended $\hat{\Sigma}$ signature:

Definition 4.28 (*SubPCFOL*$^=$ Models) *SubPCFOL*$^=$ Σ-models are *PCFOL*$^=$ $\hat{\Sigma}$-models satisfying the following set of axioms $\hat{J}(\Sigma)$ (Note: variables are universally quantified):

1. identity: $\texttt{inj}_{s,s}(x) \overset{e}{=} x$ for $s \in S$.
2. embedding injectivity: $\texttt{inj}_{s,s'}(x) \overset{e}{=} \texttt{inj}_{s,s'}(y) \Rightarrow x \overset{e}{=} y$ for $s \leq_S s'$.
3. transitivity: $\texttt{inj}_{s',s''}(\texttt{inj}_{s,s'}(x)) \overset{e}{=} \texttt{inj}_{s,s''}(x)$ for $s \leq_S s' \leq_S s''$.
4. projection: $\texttt{pr}_{s',s}(\texttt{inj}_{s,s'}(x)) \overset{e}{=} x$ for $s \leq_S s'$.
5. projection injectivity: $\texttt{pr}_{s',s}(x) \overset{e}{=} \texttt{pr}_{s',s}(y) \Rightarrow x \overset{e}{=} y$ for $s \leq_S s'$.

6. membership: $\in^{s}_{s'}(x) \Leftrightarrow def\, \mathrm{pr}_{s',s}(x)$ for $s \leq_S s'$.
7. function monotonicity: $\mathtt{inj}_{s',s}(f_{w',s'}(\mathtt{inj}_{s_1,s'_1}(x_1),\ldots,\mathtt{inj}_{s_n,s'_n}(x_n))) =$ $\mathtt{inj}_{s'',s}(f_{w'',s''}(\mathtt{inj}_{s_1,s''_1}(x_1),\ldots,\mathtt{inj}_{s_n,s''_n}(x_n)))$ for $f_{w',s'} \sim_F f_{w'',s''}$, where $w \leq_S w', w''$, $w = \langle s_1,\ldots,s_n\rangle$, $w' = \langle s'_1,\ldots,s'_n\rangle$, $w'' = \langle s''_1,\ldots,s''_n\rangle$, and $s', s'' \leq_S s$.
8. predicate monotonicity: $p_{w'}(\mathtt{inj}_{s_1,s'_1}(x_1),\ldots,\mathtt{inj}_{s_n,s'_n}(x_n)) \Leftrightarrow$ $p_{w''}(\mathtt{inj}_{s_1,s''_1}(x_1),\ldots,\mathtt{inj}_{s_n,s''_n}(x_n))$ for $p_{w'} \sim_P p_{w''}$, where $w \leq_S w', w''$, $w = \langle s_1,\ldots,s_n\rangle$, $w' = \langle s'_1,\ldots,s'_n\rangle$, and $w'' = \langle s''_1,\ldots,s''_n\rangle$.

Model morphisms for *SubPCFOL*$^=$ models are now nothing but *PCFOL*$^=$ morphisms of $\hat{\Sigma}$-models. Similarly, model reducts for *SubPCFOL*$^=$ can be defined to be model reducts in *PCFOL*$^=$ [Mos02].

4.5.3 *SubPCFOL*$^=$ Sentences and Satisfaction

SubPCFOL$^=$ sentences are just ordinary $\hat{\Sigma}$-sentences and Sentence translation along a *SubPCFOL*$^=$ signature morphism σ is just sentence translation along the *PCFOL*$^=$ signature morphism $\hat{\sigma}$.

Finally, the satisfaction relation and satisfaction condition can be inherited from *PCFOL*$^=$ as models and sentences are taken from *PCFOL*$^=$.

This draws to a close our presentation of the *SubPCFOL*$^=$ institution underlying CASL.

4.6 An Institution for Modal Logic

As described in Section 4.2, MODALCASL [Mos04a] extends CASL with elements of Modal Logic [HC96, Mos04a] . Here we present the institution underlying MODALCASL. We shall call this the *ModalSubPCFOL*$^=$ institution. Again, we omit proof details and refer the reader to [Mos04a, MC10].

4.6.1 *ModalSubPCFOL*$^=$ Signatures

MODALCASL extends CASL signatures with the notion of "flexible" operations, predicates and modalities. Modalities are introduced in two forms, firstly as basic modalities and secondly as sets of modalities defined by some terms. Formally we have:

Definition 4.29 (*ModalSubPCFOL*$^=$ Signatures) A *ModalSubPCFOL*$^=$ signature $\Sigma_M = (S, TF, PF, P, \leq_s, F_R, P_R, M, SM)$ consists of:

- a *SubPCFOL*$^=$ signature $(S, TF, PF, P, \leq_s)$,
- a predicate $F_R \subseteq TF \cup PF$ marking functions as rigid,
- a predicate $P_R \subseteq P$ marking predicates as rigid,

- a set of modalities M, and
- a set of modality sorts $SM \subseteq S$.

Given two *ModalSubPCFOL*$^=$ signatures $\Sigma_M = (S, TF, PF, P, \leq_s, F_R, P_R, M, SM)$ and $\Sigma'_M = (S', TF', PF', P', \leq'_s, F'_R, P'_R, M', SM')$, we can define a signature morphism as:

Definition 4.30 (*ModalSubPCFOL*$^=$ Signature Morphism) A *ModalSubPCFOL*$^=$ signature morphism $\sigma_M : \Sigma_M \to \Sigma'_M$ consists of:

- a *SubPCFOL*$^=$ signature morphism
  $\sigma : (S, TF, PF, P, \leq_S) \to (S', TF', PF', P', \leq'_S)$,
- a mapping $\sigma_M : M \to M'$ between modalities, and
- a mapping $\sigma_{SM} : SM \to SM'$ between modality sorts

such that rigidity and modalities are preserved, i.e.

- $\sigma(F_R) = F'_R$ (preservation of rigid functions)
- $\sigma(P_R) = P'_R$ (preservation of rigid predicates)
- $\sigma_M(M) = M'$ (preservation of modalities), and
- $\sigma_{SM}(SM) = SM'$ (preservation of modality sorts).

Defining identities and composition in the obvious way leads to the category of *ModalSubPCFOL*$^=$-signatures. Next, we consider the notion of a *ModalSubPCFOL*$^=$ Model.

4.6.2 *ModalSubPCFOL*$^=$ Models

Modal Logic is built upon the idea of worlds [HC96, Mos04a], each of which is defined (usually) by some first order model. Between such worlds, there is then an accessibility relation for each given modality. These elements are reflected directly in the *ModalSubPCFOL*$^=$ models.

Definition 4.31 (*ModalSubPCFOL*$^=$ Models) A *ModalSubPCFOL*$^=$ model is a tuple $K = (W, i, \{a_m\}_{m \in M}, \{a_c\}_{c \in M_W(s), s \in SM}, \{M_w\}_{w \in W})$ consisting of:

- a set of worlds W,
- an initial world $i \in W$,
- a binary accessibility relation $a_m \subseteq W \times W$ for each modality $m \in M$,
- a binary accessibility relation $a_c \subseteq W \times W$ for each carrier element c of the interpretation of a modality sort $s \in SM$, and

- a *SubPCFOL*$^=$ model M_w for each world $w \in W$,

such that the carrier sets and interpretations of rigid function and predicate symbols is the same in all *SubPCFOL*$^=$ models, i.e.

- carrier sets: $M_w(s) = M_v(s)$ for all $w, v \in W, s \in S$,
  (we sometimes just write $M(s)$, as carrier sets are independent of the worlds.)
- functions: $M_w(f) = M_v(f)$ for all $w, v \in W, f \in F_R : t \to s \cup P_R : t \to s, t \in S^*, s \in S$, and
- predicates: $M_w(p) = M_v(p)$ for all $w, v \in W, p \in P_R : t, t \in S^*$

and the accessibility relations a_c are preserved under the embedding of a carrier element along subsort injection, i.e., for all $s, s' \in S$ with $s \leq s'$ it holds that: for all $c \in M(s), c' \in M(s')$ with $c' = M(inj_{s,s'})(c)$ we have $(v, w) \in a_c \Rightarrow (v, w) \in a_{c'}$ for all $v, w \in W$.

Remark 4.32 Here we explicitly include a distinguished initial world into the models. This differs from Mossakowski and Girlea [MC10, Mos04a], but is required later in Section 5.4.

With respect to *ModalSubPCFOL*$^=$ model morphisms, we ensure that modality relations between worlds are preserved, along with the first order structure of a world:

Definition 4.33 Given two *ModalSubPCFOL*$^=$ models over the same signature, say $K = (W, i, \{a_m\}_{m \in M}, \{a_c\}_{c \in M(s), s \in SM}, \{M_w\}_{w \in W})$ and $L = (V, j, \{b_m\}_{m \in M}, \{b_c\}_{c \in N(s), s \in SM}, \{N_v\}_{v \in V})$, a model morphism is a mapping $(f : W \to V, \{f_w\}_{w \in W}) : K \to L$ such that:

- $f(i) = j$, that is, initial worlds are preserved,
- $(w, v) \in a_m \Rightarrow (f(w), f(v)) \in b_m$ for all $w, v \in W, m \in M$, that is, modality relations are preserved,
- $(w, v) \in a_c \Rightarrow (f(w), f(v)) \in b_{f_w(c)}$ for all $w, v \in W, c \in M(s), s \in SM$, that is, modality relations are preserved for carrier elements of modality sorts, and
- $f_w : M_w \to N_{f(w)}$, for all $w \in W$ is a *SubPCFOL*$^=$ model morphism.

Together with the standard definition of identity and composition, *ModalSubPCFOL*$^=$ models and model morphisms form the category *ModalSubPCFOL*$^=$-models.

Next we consider model reducts:

Definition 4.34 (*ModalSubPCFOL*$^=$ Model Reduct) Given a *ModalSubPCFOL*$^=$ signature morphism $\sigma : \Sigma \to \Sigma'$, the reduct of a *ModalSubPCFOL*$^=$ model $K = (V, j, \{b_m\}_{m \in M}, \{b_c\}_{c \in N(s), s \in SM}, \{N_v\}_{v \in V})$ to a model $L = (W, i, \{a_m\}_{m \in M}, \{a_c\}_{c \in M(s), s \in SM}, \{M_w\}_{w \in W})$ along σ is given by:

- $W = V$,
- $i = j$,
- $a_m = b_{\sigma_M(m)}$ for all $m \in M$
- $a_c = b_c$ where $c \in M(\sigma_{SM}(s))$, for all $s \in SM$, and
- $M_w = (N_w)|_\sigma$ for all $w \in W$.

This definition straightforwardly keeps the initial world and relations between sets of worlds in the reduced model intact. Similarly, for model morphisms the mappings between sets of worlds remains, whilst the mapping of the first-order structure of each world is reduced along the corresponding *SubPCFOL*$^=$ morphism reduct.

Definition 4.35 (*ModalSubPCFOL*$^=$ Model Morphism Reduct) Given a *ModalSubPCFOL*$^=$ signature morphism $\sigma : \Sigma \to \Sigma'$, the reduct of a model morphism $(f', \{f'_w\}_{w \in W'}) : K' \to L'$ is $(f'|_\sigma, \{f'_w|_\sigma\} w \in W')$ where:

- $f'|_\sigma = f'$ and
- $f'_w|_\sigma$ is the standard *SubPCFOL*$^=$ model morphism reduct for all $w \in W'$.

4.6.3 *ModalSubPCFOL*$^=$ Sentences

Sentences for *ModalSubPCFOL*$^=$ are an extension of *SubPCFOL*$^=$ sentences with two new connectives, namely $[m]$ and $\langle m \rangle$ where m is a modality or a term in a modal sort. These connectives represent the usual modal operators *b*ox and *d*iamond. We also note that the sentences of *SubPCFOL*$^=$ are extended to allow definitions for terms of modal sorts. As this definition is straightforward but lengthy, we refer the reader to [MC10]. With respect to sentence translation, we extend *SubPCFOL*$^=$ sentence translation. Below we present the rules for sentence translation of these new connectives:

Definition 4.36 (*ModalSubPCFOL*$^=$ Sentence Translation) Given a *ModalSubPCFOL*$^=$ signature morphism $\sigma : \Sigma \to \Sigma'$ we define the translation of sentences by:

- $Sen(\sigma)(\langle m \rangle' \varphi) = \langle \sigma(m) \rangle' Sen(\sigma)(\varphi)$
- $Sen(\sigma)([m]' \varphi) = [\sigma(m)]' Sen(\sigma)(\varphi)$

Finally, we do not present the formal detail of satisfaction as it is somewhat lengthy. Instead, we refer the reader to [MC10, Mos04a]. We note that the ModalCasl satisfaction is based on the satisfaction for Casl outlined in Section 4.4.4 where intuitively the extension states:

- $\langle m \rangle \varphi$ holds if φ holds in some world reachable in one step via m.
- $[m]\varphi$ holds if φ holds in all worlds reachable in one step via m.

This concludes our presentation of the ModalCasl institution. We now consider a simple comorphism from ModalCasl to Casl.

4.6.4 An Institution Comorphism from Modal CASL to CASL

To provide an insight into translations between institutions, we now give and overview of how ModalCasl specifications can be turned into Casl specifications. Full details are given by Mossakowski in [Mos04a, Mos02]. The comorphism consists of the following:

Signature translation: Sorts are unchanged, but a sort W (for "worlds") is added, as well as a constant *init* : W and a binary relation $R : W \times W$. Rigid operation and predicate symbols are kept unchanged. For flexible operation and predicate symbols, W is added as an extra argument.

Sentence translation: Sentences are translated according to the standard translation.

Model translation: A Casl model is turned into a ModalCasl by keeping the carrier sets (note that ModalCasl enforces constant domains) as well as the rigid operations and predicates. The set of worlds and the accessibility relation are obtained by the interpretation of w and R respectively. The flexible operations and predicates are obtained by using the current world as the extra argument of the Casl operation respectively predicate.

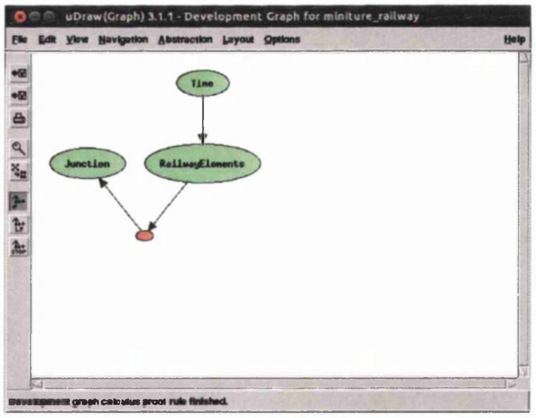
This comorphism has been implemented in the Hets toolset. Considering our example ModalCasl specification from Section 4.2, applying the above translation leads to the following Casl specification:

spec ModalInCASL =
sorts *Connector*, *Linear*, *State*, *Switch*, *Unit*, *g_World*
sorts *Linear*, *Switch* < *Unit*
op *init* : *g_World*
op *c1* : *Unit* → *Connector*
op *c2* : *Unit* → *Connector*
op *c3* : *Switch* → *Connector*
op *state_at* : *g_World* × *Unit* → *State*
pred *__hasConnector__* : *Unit* × *Connector*
pred *g_R[time]* : *g_World* × *g_World*
end

In this specification, we can clearly see that a sort *g_World* has been added, along with the distinguished initial world *init* and a binary predicate *g_R[time]* over worlds. We can then see that all rigid operations and predicates have been translated to corresponding operations and predicates with no change to their profile. Finally, we can see that the profile of the flexible operation *state_at* has been changed to *g_World* × *Unit* → *State*, which now includes the sort *g_World*.

4.7 Tool Support: Hets and Automatic Theorem Proving

Over the last few sections, we have introduced both CASL and MODALCASL, and given modelling scenarios to which each are suited. Such an approach of using various different languages for different modelling tasks is known as Heterogeneous Specification [SAA02]. This is widely accepted for the specification of large systems where heterogeneous multi-logic specifications are needed due to the fact that complex problems often have different aspects that are best specified in different logics. With respect to CASL and MODALCASL, tool support comes in the form of HETS, the Heterogeneous Tool Set [MML07],. HETS is a proof environment centered around CASL and languages that are restrictions or extensions of CASL. HETS is based around implementations of logics that form institutions. Translations between these logics are then given by implementations of institution comorphisms and morphisms. HETS also uses development graphs [MAH06] to track the structural information between heterogeneous specifications. It supports parsing and static analysis of specifications written in CASL and MODALCASL, along with many other specification languages. Figure 4.3 shows the HETS development graph for the miniature railway introduced in Section 4.1.2. In this graph we can see the structure of the specifications we have introduced. Each bubble represents a specification, the various arrows represent imports, and the red bubble represents that there are open proof goals, namely the implied property we introduced in Section 4.1.5. Finally, HETS also acts as a broker to various proof assistants and automatic theorem provers some of which we introduce below.



Development graph calculus proof rule finished.

Figure 4.3: The HETS development graph for out miniature railway specification.

4.7.1 Theorem Provers

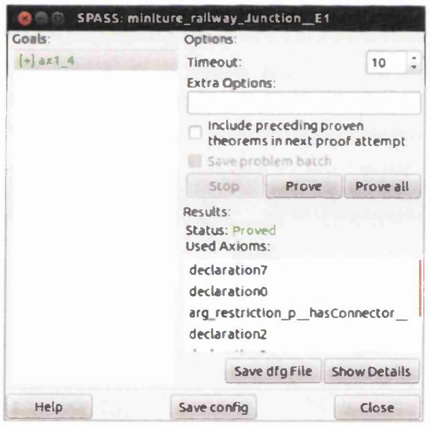
Within HETS, there are several theorem provers that can be called to discharge proofs. For example, the implication given in Section 4.1.5 can easily be discharged by SPASS [WBH+02] and Vampire [RV01]. Translations to the underlying logics of the various provers are again implemented as institution comorphisms. Although HETS has support for interactive theorem provers such as Isabelle [NPW02] for higher order logic, we will make use of the following (fairly similar) automatic provers:

SPASS [WBH$^+$02] which is an automatic theorem prover for first-order logic with equality. SPASS is a resolution based solver with the ability to produce proof trees.

Vampire [RV01] which is another automatic theorem prover for first-order logic both with or without equality. Vampire is also a resolution based solver which has a high focus towards efficiency.

The E Theorem Prover [Sch02] which is a first-order logic with equality theorem prover that applies saturation techniques. E can also produce a list of proof steps taken to prove a given goal.

The interface provided to such provers is shown in Figure 4.4, where we have the proof goal from our miniature railway example loaded ready for proving with SPASS. HETS translates the axiom base of the specification and the proof obligation into SPASS'


SPASS: miniture_railway_Junction__E1

Goals:
[+] ax1_4

Options:
Timeout: 10
Extra Options:
Include preceding proven theorems in next proof attempt
Save problem batch

Results:
Status: Proved
Used Axioms:
declaration7
declaration0
arg_restriction_p__hasConnector__
declaration2

Figure 4.4: SPASS prover interface from HETS.

input language and then passes them onto SPASS. The window shown in Figure 4.4 allows us to attempt to discharge the proof obligation via SPASS. The open proof goals are shown in the upper left area, here we only have the one property we have specified as an open goal. Next, we can see that SPASS is able to prove this simple implication automatically by the green plus symbol next to the axiom on the top left of the figure. Finally, the lower right portion of the window shows us axioms used for the proof (most are unnamed as we did not label them in the previous specifications).

Part II

Methodology Construction

Chapter 5

From DSLs to CASL

Contents

In this chapter, we present the technical framework for allowing domain specific languages formulated using UML class diagrams to be captured in CASL. In particular, we present a new institution for UML class diagrams through extension of the UML class diagram institution by Cengarle and Knapp [CK08] with numerous concepts typically appearing in applications. Along with this, we give a comorphism mapping from this institution into MODALCASL. As part of this mapping, we present the institution construction of a pointed power set institution. This construction factors out a general principle necessary for connecting UML class diagrams with an arbitrary institution capturing system dynamics. Finally, for the sake of better proof support, we use an already established comorphism to translate from MODALCASL to CASL. Throughout the chapter, we use examples from Bjørner's DSL for illustration. Overall, this approach allows us to directly import DSL specifications from industry into the methodology proposed in this thesis.

5.1 UML Class Diagrams for DSLs

As we have seen in chapter 2 and 3, UML Class Diagrams [Obj11] 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. A typical example of such an endeavour is given by the Data Model [Rai10] of our research partner Invensys Rail which aims to describe all elements within the railway domain.

In this chapter, we describe how to utilise industrial DSLs from the railway domain, formulated as UML class diagrams, for verification. As UML class diagrams only capture the static system aspects, we make the realistic assumption that the class diagram is accompanied with some natural language specification describing the dynamic system aspects. For Railways this situation is given thanks to generally accepted standard literature, e.g., [KR01].

Our work closely relates to the various approaches that utilise UML as a graphical frontend for formal methods. In this respect, we follow the approach of Cengarle and Knapp [CKTW08, CK08]. In their approach, a UML diagram can be described in its "natural" semantics and its relations to other UML diagram types is expressed by appropriate translations. Overall, this results in a compositional semantics for UML as each diagram type is treated individually. It also separates concerns: first a semantics is given, then a translation is defined. This differs from monolithic constructions, which first select a fixed, usually small number of UML diagrams and then give a semantics by translation into an established formalism. A first such attempt with CASL as the underlying formalism was given in [HCB00]. Yet another example of this second approach is the UML-RSDS method [LCA04]. UML-RSDS translates specifications consisting of class diagrams, state machines and OCL into B. Class diagram annotations in the form of stereotypes steer the translation. Lano et al. [LCA04] give a railway example, however, not to the extent of defining a DSL. Another example of this kind is the approach taken within the INESS project [dSWP11]. The INESS project defines a DSL whose components are mapped into xUML. These are then translated to Promela in order to verify railway systems with the SPIN model checker. Besides the different approach to semantics, our construction allows for theorem proving technology rather than for model checking. A third approach is given by Meng and Aichernig [MA03], who define various semantics for class diagrams depending on their use in software development. Their overall approach is of a co-algebraic nature. The most abstract level, defined as the "object type" semantics, is close to the view we take on class diagrams. The more concrete levels equip objects with a (hidden) state and a transition structure, and visibility tags for attributes and methods.

We begin by presenting an institution for capturing proper UML class diagrams, as used in Figure 2.3. To do this, we first consider the elements of UML class diagrams that we are going to capture with our constructions. We then discuss the introduction of Stereotypes [Obj11] to such class diagrams for allowing the capture of dynamical aspects and present a general construction on institutions for capturing this stereotype.

5.1.1 Constructs from UML Class Diagrams

Before presenting our institution for class diagrams, we first consider the various elements of UML class diagrams that we capture. We also highlight the main restrictions of our construction.

To begin, we consider the elements of class diagrams that are considered by Holt [Hol04]. The work we present captures all of the basic class diagram elements discuss by Holt. For example, we capture classes, arrows, relationships, generalisation, properties etc. However, we refrain from a distinction between aggregation and composition, and only capture a particular notion of composition. We also capture a restricted version of operations, namely operations that do not alter the state of an object. Such operations are often called query operations. Finally, we exclude constraints from our class diagram institution as these are usually written using natural language or another UML diagram type. Such constraints have also not been required in the applications we have considered.

Finally, with respect to the capture of data types, we consider a restricted type system compared to, for example, the work of Cengarle and Knapp [CKTW08, CK08]. In their work, they use a notationally involved natural transformation to extend a class diagram allowing for the capture of recursive type constructs. However, for our work, such types are not needed, thus instead we use "built-in" types capturing the basic types and type formers we require. Concretely we allow for Booleans, Lists, Sets and Pairs.

5.2 An Institution for UML Class Diagrams

At a high level, following the UML class diagram standard interpretation [Obj11], the UML class diagram institution is constructed as follows: The signatures are used to capture all classes, relationships, and features of a UML class diagram. The models comprise of all objects and links between objects that comply to the UML class diagram. Finally, the sentences describe the multiplicities of the associations and compositions within the class diagram. We note, that in order to provide generic access to primitive types, like *Boolean*, and type constructors like *List*, we treat these as built-in types with a standard meaning. We also note, that for technical reasons concerning empty sorts in CASL, we assume that all other classes are inhabited, i.e., to contain at least one object, for example the *null* object.

5.2.1 Class Diagram Signatures

Firstly we define the objects of the UML class diagram signature category. The signatures of the UML class diagram institution are given by *class nets*:

Definition 5.1 (Class Nets) A *class net* $\Sigma = ((C, \leq_C), K, P, M, A)$ comprises of

- a *class hierarchy* $(C, \leq_C)$, i.e., a partial order where C is a set of *class names* and the partial ordering relation $\leq_C$ represents a *generalisation relation* on C, where we say that c_1 is a *sub-class* of c_2 if $c_1 \leq_C c_2$. This partial order is also closed with respect to the built-in type $\mathsf{Boolean}$ (i.e., $\mathsf{Boolean} \in C$), and "downwards" closed with respect to the unary built-in type formers List and Set, and the binary built-in type former Pair (i.e., if $\mathsf{List}[c] \in C$ or $\mathsf{Set}[c] \in C$, then $c \in C$; and if $\mathsf{Pair}[c_1, c_2] \in C$, then $c_1, c_2 \in C$);

- a set K of *instance specifications declarations* of the form $k : c$ with k an *instance specification name* and $c \in C$;

- a set P of *property declarations* of the form $c.p(x_1 : c_1, \ldots, x_n : c_n) : c'$ with $c, c_1, \ldots, c_n, c' \in C$, $n \geq 0$, $x_1, \ldots, x_n$ *formal parameter names* and p a *property name* (where the parentheses are dropped if the property declaration has no arguments);

- a set M of *composition declarations* of the form $c{\bullet}r : c'$ with $c, c' \in C$ and r a *composition role name.* Roles are used to describe the part played by a class within a composition. Even though roles are optional within UML class diagrams, here we make them explicit. If no role name is given, then we simply use the name of the class itself;

- and a set A of *association declarations* of the form $a\{r_1 : c_1, \ldots, r_n : c_n\}$ with $n \geq 2$, $c_1, \ldots, c_n \in C$, a an *association name*, and $r_1, \ldots, r_n$ *association role names.* Again, roles are used to highlight the part played by a class within an association;

such that the following properties are satisfied:

- firstly we require that supertypes of built-in types and type formers are forbidden, for example, forbidding *Boolean* $\leq c$ for all $c \neq$ *Boolean.*

- we require uniqueness of role names for several elements within the class net, namely:

  1. instance specification names are unique: if $k_1 : c_1$ and $k_2 : c_2$ are different instance specification declarations in K, then $k_1 \neq k_2$;
  2. property names are unique along the generalisation relation: if $c_1.p_1(x_{11} : c_{11}, \ldots, x_{1n_1} : c_{1n_1}) : c'_1$ and $c_2.p_2(x_{21} : c_{21}, \ldots, x_{2n_2} : c_{2n_2}) : c'_2$ are different property declarations in P and $c_1 \leq_C c_2$, then $p_1 \neq p_2$;
  3. composition role names are unique along the generalisation relation: if $c_1{\bullet}r_1 : c'_1$ and $c_2{\bullet}r_2 : c'_2$ are different composition declarations in M and $c_1 \leq_C c_2$, then $r_1 \neq r_2$;
  4. nullary property and composition role names are unique along the generalisation relation: if $c_1.p_1 : c'_1$ is a nullary property declaration in P and $c_2{\bullet}r_2 : c'_2$ is a composition declaration in M and $c_1 \leq_C c_2$, then $p_1 \neq r_2$; and if $c_1{\bullet}r_1 : c'_1$ is a composition declaration in M and $c_2.p_2 : c'_2$ is a nullary property declaration in P and $c_1 \leq_C c_2$, then $r_1 \neq p_2$;
  5. association role names are unique: if $a\{r_1 : c_1, \ldots, r_n : c_n\}$ is an association declaration in A and $i \neq j$, $1 \leq i, j \leq n$, then $r_i \neq r_j$;

- Finally, we require that composition declarations are cycle-free, that is, if $c_1{\bullet}r_1 : c_2, \ldots, c_n{\bullet}r_n : c_{n+1} \in M$, then $c_{n+1} \neq c_1$.

As an example, considering the UML class diagram in Figure 2.3, we gain the following class net:

Classes:	$\{$*Net*, *Station*, …, *Pair*[*Unit*, *Path*], *List*[*Pair*[*Unit*, *Path*]]$\}$
Generalisations:	*Point* $\leq$ *Unit*, …, *Route* $\leq$ *List[Pair[Unit, Path]]*
Properties:	$\{$*Net.id* : *UID*, …, *Route.isOpen*(*r* : *Route*):*Boolean*$\}$
Compositions:	$\{$*Station* ◆*has* : *Unit*, *Station* ◆*has*: *Track*$\}$
Associations:	$\{$*stateAt*(*unit*: *Unit*, *state* : *UnitState*), …$\}$

Here we can see the inclusion of default role names, for example, for the *stateAt* association. It has a role name *unit* for the *Unit* class involved in the association and similarly a *state* role name for the *UnitState*.

Next, for morphisms between class nets, we define:

Definition 5.2 (Class Net Morphisms) A *class net morphism* $\sigma = (\gamma, \kappa, \pi, \mu, \alpha) : \Sigma = ((C, \leq_C), K, P, M, A) \to \mathrm{T} = ((D, \leq_D), L, Q, N, B)$ is given by

- a class hierarchy map $\gamma : (C, \leq_C) \to (D, \leq_D)$ such that γ is a monotone map from $(C, \leq_C)$ to $(D, \leq_D)$, i.e., $\gamma(c) \leq_D \gamma(c')$ if $c \leq_C c'$ and homomorphic with respect to types and type formers, i.e., $\gamma(\mathsf{Boolean}) = \mathsf{Boolean}$, $\gamma(\mathsf{List}[c]) = \mathsf{List}[\gamma(c)]$ for all $c \in C$, etc.

- an instance specification map $\kappa : K \to L$ such that if $\kappa(k : c) = l : d \in L$, then $d = \gamma(c)$;

- a property declaration map $\pi : P \to Q$ such that if $\pi(c.p(x_1 : c_1, \ldots, x_m : c_m) : c') = d.q(y_1 : d_1, \ldots, y_n : d_n) : d' \in Q$, then $m = n$, $d = \gamma(c)$, $d_i = \gamma(c_i)$ for all $1 \leq i \leq m$, and $d' = \gamma(d')$;

- a composition declaration map $\mu : M \to N$ such that if $\mu(c{\blacklozenge}r : c') = d{\blacklozenge}s : d' \in M$, then $d = \gamma(c)$ and $d' = \gamma(d')$;

- an association declaration map $\alpha : A \to B$ such that n-ary association declarations are mapped to n-ary association declarations along γ, that is, $\alpha(a\{r_1 : c_1, \ldots, r_n : c_n\}) = b\{s_1 : d_1, \ldots, s_m : d_m\} \in B$, then there is a bijective map $\rho : \{r_1, \ldots, r_n\} \to \{s_1, \ldots, s_m\}$ with $d_j = \gamma(c_i)$ if $\rho(r_i) = s_j$; we denote this bijective map ρ also by ρ_α.

We define the composition of signature morphisms to be pointwise composition of each component function. That is:

Definition 5.3 (Class Net Morphism Composition) Given $\sigma = (\gamma, \kappa, \pi, \mu, \alpha) : \Sigma = ((C, \leq_C), K, P, M, A) \to \Sigma' = ((C', \leq_C)', K', P', M', A')$ and $\sigma' = (\gamma', \kappa', \pi', \mu', \alpha') : \Sigma' = ((C', \leq_C)', K', P', M', A') \to \Sigma'' = ((C'', \leq_C)'', K'', P'', M'', A'')$. We define a candidate: $\sigma' \circ \sigma = (\gamma' \circ \gamma, \kappa' \circ \kappa, \pi' \circ \pi, \mu' \circ \mu, \alpha' \circ \alpha) : \Sigma = ((C, \leq_C), K, P, M, A) \to \Sigma'' = ((C'', \leq_C)'', K'', P'', M'', A''))$

Identity morphisms are defined in the expected way:

Definition 5.4 (Class Net Identity Morphisms) A candidate class net identity morphism $id_\sigma = (id_\gamma, id_\kappa, id_\pi, id_\mu, id_\alpha) : \Sigma = ((C, \leq_C), K, P, M, A) \to \Sigma = ((C, \leq_C), K, P, M, A)$ is given by:

- a class hierarchy map $\gamma : (C, \leq_C) \to (C, \leq_C)$ mapping $C \mapsto C$ and $\leq_C \mapsto \leq_C$;
- an instance specification map $\kappa : K \to K$ mapping $k : c \mapsto k : c$;
- a property declaration map $id_\pi : (P \to P)$ mapping $c.p(x_1 : c_1, \ldots, x_n : c_n) : c' \mapsto c.p(x_1 : c_1, \ldots, x_n : c_n) : c'$;
- a composition declaration map $id_\mu : (M \to M)$ mapping $c\bullet r : c' \mapsto c\bullet r : c'$
- an association map $id_\alpha : (\alpha \to \alpha)$ mapping $a\{r_1 : c_1, \ldots, r_n : c_n\} \mapsto a\{r_1 : c_1, \ldots, r_n : c_n\}$

Lemma 5.5 (The Category Cl of Class Nets) Class nets and class net morphisms with composition and identity as defined by the above candidates form the category of *class nets*, denoted by Cl.

Proof. Considering composition, given $\sigma = (\gamma, \kappa, \pi, \mu, \alpha) : \Sigma = ((C, \leq_C), K, P, M, A) \to \Sigma' = ((C', \leq_C)', K', P', M', A'), \sigma' = (\gamma', \kappa', \pi', \mu', \alpha') : \Sigma' = ((C', \leq_C)', K', P', M', A') \to \Sigma'' = ((C'', \leq_C)'', K'', P'', M'', A'')$ and $\sigma' \circ \sigma = (\gamma' \circ \gamma, \kappa' \circ \kappa, \pi' \circ \pi, \mu' \circ \mu, \alpha' \circ \alpha)$ we show that composition forms a valid morphism:

- $(\gamma' \circ \gamma)$ trivially preserves monotonicity of $\leq_c$ as it is nothing but function composition.
- If $(\kappa' \circ \kappa)(k : c) = k'' : c'' \in K''$ then we require that $c'' = (\gamma' \circ \gamma)(c)$. This follows as we know $\kappa(k : c) = k' : c'inK'$ s.t. $k' : c' \in K'$ and that $\kappa'(k' : c') = k'' : c'' \in K'')$ s.t. $k'' : c'' \in K''$. Hence we are done.
- Cases for conditions on $\pi' \circ \pi$ and $\mu' \circ \mu$ follow similarly.
- Finally, $\alpha' \circ \alpha$ trivially preserves bijectivity (as it is nothing but function composition).

Hence composition morphisms are morphisms in Cl. Composition is also trivially associative as composition is defined as pointwise function composition. Finally, identity morphisms as defined are trivially morphisms which are left and right unit in Cl. □

5.2.2 Class Diagram Models

A Σ-model of the UML class diagram for a class net is given by an instance net, defined as follows:

Definition 5.6 (Instance Nets) Given a class net $\Sigma = ((C, \leq_C), K, P, M, A)$ a model of this net is a Σ-*instance net* $\mathcal{I} = (C^\mathcal{I}, K^\mathcal{I}, P^\mathcal{I}, M^\mathcal{I}, A^\mathcal{I})$ consisting of interpretations for all declarations where

- $C^{\mathcal{I}}$ is a class interpretation mapping each $c \in C$ to a non-empty set such that $C^{\mathcal{I}}(c_1) \subseteq C^{\mathcal{I}}(c_2)$ if $c_1 \leq_C c_2$ with standard mappings for types and type formers, that is, $C^{\mathcal{I}}(\mathsf{Boolean}) = \{\mathit{ff}, \mathit{tt}\}$, $C^{\mathcal{I}}(\mathsf{List}[c]) = (C^{\mathcal{I}}(c))^*$ (where V^* denotes the finite sequences over V), $C^{\mathcal{I}}(\mathsf{Set}[c]) = \wp(C^{\mathcal{I}}(c))$ (where $\wp(V)$ denotes the powerset of V), and $C^{\mathcal{I}}(\mathsf{Pair}[c_1, c_2]) = C^{\mathcal{I}}(c_1) \times C^{\mathcal{I}}(c_2)$;
- $K^{\mathcal{I}}$ is an instance specification interpretation mapping each $k : c \in K$ to an element of $C^{\mathcal{I}}(c)$;
- $P^{\mathcal{I}}$ is a property declaration interpretation mapping each $c.p(x_1 : c_1, \ldots, x_n : c_n) : c' \in P$ to a partial map $C^{\mathcal{I}}(c) \times C^{\mathcal{I}}(c_1) \times \cdots \times C^{\mathcal{I}}(c_n) \rightarrow ? \, C^{\mathcal{I}}(c')$;
- $M^{\mathcal{I}}$ is a composition declaration interpretation mapping each $c{\bullet}r : c' \in M$ to a map $C^{\mathcal{I}}(c) \rightarrow \wp(C^{\mathcal{I}}(c'))$;
- $A^{\mathcal{I}}$ is an association declaration interpretation mapping each $a(r_1 : c_1, \ldots, r_n : c_n) \in A$ to a subset of $\prod_{1 \leq i \leq n} C^{\mathcal{I}}(c_i)$,

such that

- instance specifications $k_1 : c$ and $k_2 : c$ with $k_1 \neq k_2$ are interpreted differently: $K^{\mathcal{I}}(k_1 : c) \neq K^{\mathcal{I}}(k_2 : c)$, and
- each instance has at most one owner: if $o' \in M^{\mathcal{I}}(c_1{\bullet}r_1 : c_1')(o_1) \cap M^{\mathcal{I}}(c_2{\bullet}r_2 : c_2')(o_2)$, then $o_1 = o_2$.

An example excerpt of an instance net for the class net given in Section 2.5 (namely Figure 2.3) is:

$$
\begin{array}{lcl}
C^{\mathcal{I}}(\mathsf{Net}) & = & \{\texttt{LondonUnderground}\} \\
C^{\mathcal{I}}(\mathsf{Station}) & = & \{\texttt{Charing}, \ldots, \texttt{Picadilly}\} \\
C^{\mathcal{I}}(\mathsf{Connector}) & = & \{\texttt{c1}, \texttt{c2}, \texttt{c3}, \ldots\} \\
C^{\mathcal{I}}(\mathsf{Pair[Connector, Connector]}) & = & \{(\texttt{c1}, \texttt{c1}), (\texttt{c1}, \texttt{c2}), (\texttt{c2}, \texttt{c1}), (\texttt{c2}, \texttt{c2}), \ldots\} \\
P^{\mathcal{I}}(\mathsf{Net.ID : UID}) & : & C^{\mathcal{I}}(\mathsf{Net}) \rightarrow ? \, C^{\mathcal{I}}(\mathsf{UID}) \quad \text{where} \\
 & & \texttt{LondonUnderground} \mapsto \texttt{"LUG"} \quad \text{etc.} \\
P^{\mathcal{I}}(\mathsf{Route.isOpen : Boolean}) & : & C^{\mathcal{I}}(\mathsf{Route}) \rightarrow ? \, C^{\mathcal{I}}(\mathsf{Boolean}) \quad \text{where} \\
 & & \mathsf{R1} \mapsto \mathit{tt} \quad \text{etc.} \\
M^{\mathcal{I}}(\mathsf{Net{\bullet}has : Station}) & : & C^{\mathcal{I}}(\mathsf{Net}) \rightarrow \wp(C^{\mathcal{I}}(\mathsf{Station})) \quad \text{where} \\
 & & \mathsf{LondonUnderground} \mapsto \{\texttt{Charing}, \ldots, \texttt{Picadilly}\} \\
A^{\mathcal{I}}(\mathsf{stateAt}(\ldots)) & = & \{(\texttt{LU1}, \{(\texttt{c1}, \texttt{c2})\}), (\texttt{LU2}, \{(\texttt{c2}, \texttt{c3})\}), \ldots\}
\end{array}
$$

For model morphisms, we define Σ-*instance net morphisms* as:

Definition 5.7 (Instance Net Morphisms) A Σ-*instance net morphism* $\zeta : \mathcal{I} = (C^{\mathcal{I}}, K^{\mathcal{I}}, P^{\mathcal{I}}, M^{\mathcal{I}}, A^{\mathcal{I}}) \rightarrow \mathcal{J} = (C^{\mathcal{J}}, K^{\mathcal{J}}, P^{\mathcal{J}}, M^{\mathcal{J}}, A^{\mathcal{J}})$ over a class net $\Sigma = ((C, \leq_C), K, P, M, A)$ is given by a map $\zeta \in \Pi_{c \in C} \,.\, C^{\mathcal{I}}(c) \rightarrow C^{\mathcal{J}}(c)$ such that all interpretations of declarations are mapped homomorphically, that is:

- including interpretations of all types and type formers, for example, $\zeta(\mathsf{Pair}[c_1, c_2])(o_1, o_2) = (\zeta(c_1)(o_1), \zeta(c_2)(o_2))$ for all $c_1, c_2 \in C$, $o_1 \in C^{\mathcal{I}}(c_1)$, $o_2 \in C^{\mathcal{I}}(c_2)$

- $K^{\mathcal{J}}(k : c) = \zeta(c)(K^{\mathcal{I}}(k : c))$ for each $k : c \in K$;

- $P^{\mathcal{J}}(c.p(x_1 : c_1, \ldots, x_n : c_n) : c')(\zeta(c)(o), \zeta(c_1)(o_1), \ldots, \zeta(c_n)(o_n)) \cong \zeta(c')(P^{\mathcal{I}}(c.p(x_1 : c_1, \ldots, x_n : c_n) : c')(o, o_1, \ldots, o_n))$ for each $c.p : c' \in P$ and $o \in C^{\mathcal{I}}(c)$, $o_1 \in C^{\mathcal{I}}(c_1), \ldots, o_n \in C^{\mathcal{I}}(c_n)$ (where $\cong$ means that if either side is defined, then also the other side is defined and equal);

- $M^{\mathcal{J}}(c{\bullet}r : c')(\zeta(c)(o)) = \zeta(c')(P^{\mathcal{I}}(c{\bullet}r : c')(o))$ for each $c{\bullet}r : c' \in M$ and $o \in C^{\mathcal{I}}(c)$;

- $A^{\mathcal{J}}(a\{r_1 : c_1, \ldots, r_n : c_n\}) = \{\{r_1 \mapsto \zeta(c_1)(t(r_1)), \ldots, r_n \mapsto \zeta(c_n)(t(r_n))\} \mid t \in A^{\mathcal{I}}(a\{r_1 : c_1, \ldots, r_n\})\}$.

Once again, we define composition of morphisms to be pointwise composition of each component map:

Definition 5.8 (Composition of Instance Net Morphisms) Given $\zeta : \mathcal{I} = (C^{\mathcal{I}}, K^{\mathcal{I}}, P^{\mathcal{I}}, M^{\mathcal{I}}, A^{\mathcal{I}}) \to \mathcal{I}' = (C^{\mathcal{I}'}, K^{\mathcal{I}'}, P^{\mathcal{I}'}, M^{\mathcal{I}'}, A^{\mathcal{I}'})$ and $\zeta' : \mathcal{I}' = (C^{\mathcal{I}'}, K^{\mathcal{I}'}, P^{\mathcal{I}'}, M^{\mathcal{I}'}, A^{\mathcal{I}'}) \to \mathcal{I}'' = (C^{\mathcal{I}''}, K^{\mathcal{I}''}, P^{\mathcal{I}''}, M^{\mathcal{I}''}, A^{\mathcal{I}''})$. We define a candidate: $(\zeta' \circ \zeta)(c) = \zeta'(c) \circ \zeta(c)$ for all $c \in C$.

Finally, identity morphism are defined in the expected way:

Definition 5.9 (Identity Instance Net Morphisms) An candidate instance net identity morphism $id_{\mathcal{I}} : \mathcal{I} = (C^{\mathcal{I}}, K^{\mathcal{I}}, P^{\mathcal{I}}, M^{\mathcal{I}}, A^{\mathcal{I}}) \to \mathcal{I} = (C^{\mathcal{I}}, K^{\mathcal{I}}, P^{\mathcal{I}}, M^{\mathcal{I}}, A^{\mathcal{I}})$ is given by $id_{\mathcal{I}}(c) = id_{C^{\mathcal{I}}(c)}$ for all $c \in C$.

Lemma 5.10 (Instance Nets form the Category Inst(Σ)) Σ-instance nets as object and Σ-instance net morphisms with composition and identity as defined above form the category of *Σ-instance nets*, denoted by Inst(Σ).

Proof. Considering composition, given $\zeta : \mathcal{I} = (C^{\mathcal{I}}, K^{\mathcal{I}}, P^{\mathcal{I}}, M^{\mathcal{I}}, A^{\mathcal{I}}) \to \mathcal{I}' = (C^{\mathcal{I}'}, K^{\mathcal{I}'}, P^{\mathcal{I}'}, M^{\mathcal{I}'}, A^{\mathcal{I}'})$, $\zeta' : \mathcal{I}' = (C^{\mathcal{I}'}, K^{\mathcal{I}'}, P^{\mathcal{I}'}, M^{\mathcal{I}'}, A^{\mathcal{I}'}) \to \mathcal{I}'' = (C^{\mathcal{I}''}, K^{\mathcal{I}''}, P^{\mathcal{I}''}, M^{\mathcal{I}''}, A^{\mathcal{I}''})$ we show that the composition $\zeta' \circ \zeta$ forms a valid morphism:

- $K^{\mathcal{I}''}(k : c) = (\zeta' \circ \zeta)I(c)(K^{\mathcal{I}}(k : c))$ holds as $(\zeta' \circ \zeta)(c)(K^{\mathcal{I}}(k : c)) = \zeta'(\zeta(c))(K^{\mathcal{I}}(k : c))$ by definition of composition and $K^{\mathcal{I}''}(k : c) = \zeta'(\zeta(c))(K^{\mathcal{I}}(k : c))$ by definition of composition. Hence we are done.

- Cases for conditions on P, M and A follow similarly.

Hence composed morphisms are morphisms in Inst(Σ). Composition is also trivially associative as composition is defined as pointwise function composition. Finally, identity morphisms as defined are trivially morphisms which are left and right unit in Inst(Σ). □

For reducts of instance nets and instance net morphisms, let $\sigma = (\gamma, \kappa, \pi, \mu, \alpha) : \Sigma = ((C, \leq_C), K, P, M, A) \to \mathrm{T} = ((D, \leq_D), L, Q, N, B)$ be a class net morphism. Then we have:

Definition 5.11 The *reduct* of a T-instance net $\mathcal{J} = (D^{\mathcal{J}}, L^{\mathcal{J}}, Q^{\mathcal{J}}, N^{\mathcal{J}}, B^{\mathcal{J}})$ along σ is the Σ-instance net $\mathcal{J}|\sigma = (C^{\mathcal{J}|\sigma}, K^{\mathcal{J}|\sigma}, P^{\mathcal{J}|\sigma}, M^{\mathcal{J}|\sigma}, A^{\mathcal{J}|\sigma})$ formed by component-wise reducts, that is:

- $C^{\mathcal{J}|\sigma}(c) = D^{\mathcal{J}}(\gamma(c))$;
- $K^{\mathcal{J}|\sigma}(k : c) = L^{\mathcal{J}}(\kappa(k : c))$;
- $P^{\mathcal{J}|\sigma}(c.p(x_1 : c_1, \ldots, x_n : c_n) : c') = Q^{\mathcal{J}}(\pi(c.p(x_1 : c_1, \ldots, x_n : c_n) : c'))$;
- $M^{\mathcal{J}|\sigma}(c \bullet r : c') = N^{\mathcal{J}}(\mu(c \bullet r : c'))$;
- $A^{\mathcal{J}|\sigma}(a\{r_1 : c_1, \ldots, r_n : c_n\}) = B^{\mathcal{J}}(\alpha(a\{r_1 : c_1, \ldots, r_n : c_n\}))$.

Lemma 5.12 Model reducts are Σ-models.

Proof. Let $\sigma = (\gamma, \kappa, \pi, \mu, \alpha) : \Sigma = ((C, \leq_C), K, P, M, A) \to \mathrm{T} = ((D, \leq_D), L, Q, N, B)$ be a class net morphism. Let $\mathcal{J} = (D^{\mathcal{J}}, L^{\mathcal{J}}, Q^{\mathcal{J}}, N^{\mathcal{J}}, B^{\mathcal{J}})$ be a T-instance net. Considering the reduct $\mathcal{J}|\sigma = (C^{\mathcal{J}|\sigma}, K^{\mathcal{J}|\sigma}, P^{\mathcal{J}|\sigma}, M^{\mathcal{J}|\sigma}, A^{\mathcal{J}|\sigma})$ we have:

- $C^{\mathcal{J}|\sigma}(c_1) \subseteq C^{\mathcal{J}|\sigma}(c_2)$ if $c_1 \leq_C c_2$ as we know $\gamma(c1) \leq \gamma(c2)$ as γ is monotonic and $D^{\mathcal{J}}(\gamma(c1)) \leq D^{\mathcal{J}}(\gamma(c2))$ as $\mathcal{J}$ is a T instance net.
- $K^{\mathcal{J}|\sigma}(k : c) \in C^{\mathcal{J}|\sigma}(c)$ as $C^{\mathcal{J}}(c) = D^{\mathcal{J}}(\gamma(c))$ and $K^{\mathcal{J}|\sigma}(k : c) = L^{\mathcal{J}}(\kappa(k : c))$ and we know that $L^{\mathcal{J}}(\kappa(k : c)) \in D^{\mathcal{J}}(\gamma(c))$.
- $P^{\mathcal{J}|\sigma}$ is trivially a map mapping each $c.p(x_1 : c_1, \ldots, x_n : c_n) : c' \in P$ to a partial map $C^{\mathcal{J}|\sigma}(c) \times C^{\mathcal{J}|\sigma} \times \cdots \times C^{\mathcal{J}|\sigma} \to ? \; C^{\mathcal{J}|\sigma}(c')$ as $Q^{\mathcal{J}}$ has this property.
- Conditions for $M^{\mathcal{J}|\sigma}$ and $A^{\mathcal{J}|\sigma}$ follow similarly to the conditions for $P^{\mathcal{J}|\sigma}$.

Finally, it follows by definition that each instance in $\mathcal{J}|\sigma$ "has at most one owner". □

In a similar manner, for instance net morphisms we define reducts as:

Definition 5.13 (Instance Net Morphsism Reduct) The *reduct* of a T-instance net morphism $\zeta : \mathcal{J}_1 \to \mathcal{J}_2$ along σ is the Σ-instance net morphism $\zeta|\sigma : \mathcal{J}_1|\sigma \to \mathcal{J}_2|\sigma$ with $\zeta|\sigma(c) = \zeta(\gamma(c))$.

Lemma 5.14 Model reduct morphisms are morphisms in Cat^{op}.

Proof. Given $\zeta : \mathcal{J}_1 = (D^{\mathcal{J}_1}, L^{\mathcal{J}_1}, Q^{\mathcal{J}_1}, N^{\mathcal{J}_1}, B^{\mathcal{J}_1}) \to \mathcal{J}_2 = (D^{\mathcal{J}_2}, L^{\mathcal{J}_2}, Q^{\mathcal{J}_2}, N^{\mathcal{J}_2}, B^{\mathcal{J}_2})$ as a T-instance net morphism. We show $\zeta|\sigma : \mathcal{J}_1|\sigma = (D^{\mathcal{J}_1|\sigma}, L^{\mathcal{J}_1|\sigma}, Q^{\mathcal{J}_1|\sigma}, N^{\mathcal{J}_1|\sigma}, B^{\mathcal{J}_1|\sigma}) \to \mathcal{J}_2|\sigma = (D^{\mathcal{J}_2|\sigma}, L^{\mathcal{J}_2|\sigma}, Q^{\mathcal{J}_2|\sigma}, N^{\mathcal{J}_2|\sigma}, B^{\mathcal{J}_2|\sigma})$ is a valid morphism. Firstly, by definition we know $\zeta|\sigma(c) = \zeta(\gamma(c))$, hence:

- $K^{\mathcal{J}_2|\sigma}(k : c) = \zeta(c)(K^{\mathcal{J}_1|\sigma}(k : c))$ as $\zeta(c)(K^{\mathcal{J}_1|\sigma}(k : c)) = \zeta(\gamma(c))K^{\mathcal{J}_1|\sigma})$ and we know $K^{\mathcal{J}}(k : c) = \zeta(\gamma(c))(K^{(I)}(k : c))$ is a valid morphism. Hence we are done for this case.
- Cases for $P^{\mathcal{J}}$ through to $A^{\mathcal{J}}$ follow in a similar manner. □

We can now formulate the model reduct functor.

Lemma 5.15 (Instance Reduct Functor $-|\sigma$) The σ-reducts of T-instance nets and T-instance net morphisms yield a functor $-|\sigma : \mathrm{Inst(T)} \to \mathrm{Inst}(\Sigma)$.

Proof. Given Lemma 5.12 and Lemma 5.14 it follows that identities are preserved. Therefore it remains to show $-|\sigma$ is functorial. That is, $(\zeta' \circ \zeta)|\sigma(c) = \zeta'|\sigma(c) \circ \zeta|\sigma(c)$. By definition $(\zeta' \circ \zeta)|\sigma(c) = (\zeta' \circ \zeta)(\gamma(c))$ and $\zeta'(\gamma(c)) \circ \zeta(\gamma(c))$ hence we are done. □

Similarly, we can now show that:

Lemma 5.16 (Inst is a functor.) $\mathrm{Inst} : \mathrm{Cl} \to \mathrm{Cat}^{op}$ is a functor.

Proof. Follows from Lemma 5.10 and Lemma 5.15. As Lemma 5.10 shows that $\mathrm{Inst}(\Sigma)$ forms a category and Lemma 5.15 shows that $(-)|\sigma$ is a morphism between such categories. Finally, it is trivial to see that identities are preserved. □

5.2.3 Multiplicity Formulae as Sentences

For the sentences of the UML class diagram institution we use the multiplicity constraints from the class diagram. We define:

Definition 5.17 (Multiplicity Formulae) The *multiplicity formulae* are strings formed over the following grammar:

$$\begin{aligned}
\mathit{Frm} &::= \mathit{NumLit} \leq \mathit{FunExpr} \mid \mathit{FunExpr} \leq \mathit{NumLit} \mid \mathit{Composition}\ ! \\
\mathit{FunExpr} &::= \#\ \mathit{Composition} \mid \#\ \mathit{Association}\ [\ \mathit{Role}(,\ \mathit{Role})^*\ ] \\
\mathit{Composition} &::= \mathit{Class}{\blacklozenge}\mathit{Role} : \mathit{Class} \\
\mathit{Association} &::= \mathit{Name}\{\mathit{Role} : \mathit{Class}(,\ \mathit{Role} : \mathit{Class})^*\} \\
\mathit{Class} &::= \mathit{Name} \\
\mathit{Role} &::= \mathit{Name} \\
\mathit{NumLit} &::= 0 \mid 1 \mid \cdots
\end{aligned}$$

where *Name* is a set of strings. The $\leq$-formulae express constraints on the cardinalities of composition and association declarations, i.e., how many instances are allowed to be in a certain relation with others. The #-expressions return the number of links in an association when some roles are fixed. The !-formulae for compositions express that the owning end must not be empty.

The set of sentences or Σ-*multiplicity constraints* $\mathit{Mult}(\Sigma)$ for a class net Σ is given by the multiplicity formulae in *Frm* such that all mentioned elements of *Composition* and *Association* correspond to composition declarations and association declarations of

Σ respectively, and the *Role* names mentioned in the last clause of *FunExpr* occur in the mentioned association.

Some of the cardinality constraints of the running example in Figure 2.3 can be expressed with multiplicity formulae as follows:

- Station•has : Track ! — each Track is owned by a Station via has,
- $2 \leq$ #has(connector : Connector, unit : Unit)[unit] — association has links each Unit to at least two Connectors,
- #stateAt(unitState : UnitState, unit : Unit)[unit] $= 1$ — association stateAt links each Unit to exactly one UnitState.

Note that in the above, we have used formulae with $=$ instead of two formulae with $\leq$ where the left hand side and the right hand side are switched.

The *translation* of a formula $\varphi \in \mathit{Mult}(\Sigma)$ along a class net morphism $\sigma = (\gamma, \kappa, \pi, \mu, \alpha) : \Sigma \to \mathrm{T}$, written as $\sigma(\varphi)$, is given by simply applying σ to compositions, associations, and role names as follows:

Definition 5.18 (Sentence Translation for Multiplicity Formulae) The *translation* of a formula $\varphi \in \mathit{Mult}(\Sigma)$ along a class net morphism $\sigma = (\gamma, \kappa, \pi, \mu, \alpha) : \Sigma \to \mathrm{T}$ is given by:

$$\begin{array}{lcl}
\sigma(\ell \leq \#c{\bullet}r : c') & = & \ell \leq \#\mu(c{\bullet}r : c') \\
\sigma(\#c{\bullet}r : c' \leq \ell) & = & \#\mu(c{\bullet}r : c') \leq \ell \\
\sigma(c{\bullet}r : c'!) & = & \mu(c{\bullet}r : c')! \\
\sigma(\ell \leq \#a\{r_1 : c_1, \ldots, r_n : c_n\}[r_{i_1}, \ldots, r_{i_m}]) & = & \ell \leq \#\alpha(a\{r_1 : c_1, \ldots, r_n : c_n\}) \\
 & & [\rho_\alpha(r_{i_1}), \ldots, \rho_\alpha(r_{i_m})] \\
\sigma(\#a\{r_1 : c_1, \ldots, r_n : c_n\}[r_{i_1}, \ldots, r_{i_m}] \leq \ell) & = & \#\alpha(a\{r_1 : c_1, \ldots, r_n : c_n\}) \\
 & & [\rho_\alpha(r_{i_1}), \ldots, \rho_\alpha(r_{i_m})] \leq \ell
\end{array}$$

Finally, we gain the fact that *Mult* forms a functor.

Lemma 5.19 *Mult* : Cl $\to$ Set is a functor.

Proof. We know by definition that $\mathit{Mult}(\Sigma)$ is a set and hence an object in Set. We also know by definition that sentence translation is a function and hence a morphism in Set. It remains to show that *Mult* is functorial. Given $\sigma : \Sigma \to \Sigma'$ and $\sigma' : \Sigma' \to \Sigma''$ and a sentence φ we have that $\mathit{Mult}(\sigma' \circ \sigma)(\varphi) = (\mathit{Mult}(\sigma') \circ \mathit{Mult}(\sigma))(\varphi)$ as $(\mathit{Mult}(\sigma') \circ \mathit{Mult}(\sigma))(\varphi) = \mathit{Mult}(\sigma')(\mathit{Mult}(\sigma)(\varphi))$ and composition of morphisms is nothing but component-wise function composition. $\square$

Finally, we come to the satisfaction relation.

Definition 5.20 (Satisfaction Relation for Class Nets) The Σ-*satisfaction relation* $- \models_\Sigma - \subseteq |\mathrm{Inst}(\Sigma)| \times \mathit{Sen}(\Sigma)$ of the UML class diagram institution is defined for each Σ-instance net $\mathcal{I} = (C^\mathcal{I}, K^\mathcal{I}, P^\mathcal{I}, M^\mathcal{I}, A^\mathcal{I})$ as

$$\begin{array}{lcl}
\mathcal{I} \models_\Sigma \ell \leq \#c{\bullet}r : c' & \Longleftrightarrow & \forall o \in C^{\mathcal{I}}(c) \,.\, [\![\ell]\!] \leq |M^{\mathcal{I}}(c{\bullet}r : c')(o)| \\
\mathcal{I} \models_\Sigma \#c{\bullet}r : c' \leq \ell & \Longleftrightarrow & \forall o \in C^{\mathcal{I}}(c) \,.\, |M^{\mathcal{I}}(c{\bullet}r : c')(o)| \leq [\![\ell]\!] \\
\mathcal{I} \models_\Sigma c{\bullet}r : c'! & \Longleftrightarrow & \forall o' \in C^{\mathcal{I}}(c') \,.\, \exists o \in C^{\mathcal{I}}(c) \,. \\
 & & \quad M^{\mathcal{I}}(c{\bullet}r : c')(o) = o' \\
\mathcal{I} \models_\Sigma \ell \leq \#a(r_1 : c_1, \ldots, & & \\
r_n : c_n)[r_{i_1}, \ldots, r_{i_m}] & \Longleftrightarrow & \forall o_{i_1} \in C^{\mathcal{I}}(c_{i_1}), \ldots, o_{i_m} \in C^{\mathcal{I}}(c_{i_m}) \,. \\
 & & \quad [\![\ell]\!] \leq |\{t \in A^{\mathcal{I}}(a(r_1 : c_1, \ldots, r_n : c_n)) \mid \\
 & & \quad t_{i_1} = o_{i_1}, \ldots, t_{i_m} = o_{i_m}\}| \\
\mathcal{I} \models_\Sigma \#a(r_1 : c_1, \ldots, r_n : c_n) & & \\
[r_{i_1}, \ldots, r_{i_m}] \leq \ell & \Longleftrightarrow & \forall o_{i_1} \in C^{\mathcal{I}}(c_{i_1}), \ldots, o_{i_m} \in C^{\mathcal{I}}(c_{i_m}) \,. \\
 & & \quad |\{t \in A^{\mathcal{I}}(a(r_1 : c_1, \ldots, r_n : c_n)) \mid \\
 & & \quad t_{i_1} = o_{i_1}, \ldots, t_{i_m} = o_{i_m}\}| \leq [\![\ell]\!]
\end{array}$$

where $[\![-]\!] : \mathit{NumLit} \to \mathbb{Z}$ maps a numerical literal to an integer.

Using this definition, we can show the satisfaction condition for the UML class diagram institution, that is:

Lemma 5.21 (Satisfaction Condition for Class Diagrams) Let $\Sigma = ((C, \leq_C), K, P, M, A)$ and $\mathrm{T} = ((D, \leq_D), L, Q, N, B)$ be class nets, let $\sigma = (\gamma, \kappa, \pi, \mu, \alpha) : \Sigma \to \mathrm{T}$ be a class net morphism, let $\mathcal{J} = (D^{\mathcal{J}}, L^{\mathcal{J}}, Q^{\mathcal{J}}, N^{\mathcal{J}}, B^{\mathcal{J}})$ be a T-instance net, and let $\varphi \in \mathit{Mult}(\Sigma)$. Then the *satisfaction condition* holds:

$$\mathcal{J} \models_{\mathrm{T}} \sigma(\varphi) \iff \mathcal{J}|\sigma \models_\Sigma \varphi$$

for each $\sigma : \Sigma \to \mathrm{T}$ in Cl, each $\mathcal{J} \in |\mathrm{Inst}(\mathrm{T})|$, and each $\varphi \in \mathrm{Mult}(\Sigma)$.

Proof. We make a case distinction over the kind of multiplicity constraint φ.

Case: $\varphi \equiv \ell \leq \#c{\bullet}r : c'$ is given by:

$$\begin{array}{lcr}
\mathcal{J} \models_{\mathrm{T}} \sigma(\ell \leq \#c{\bullet}r : c') & \Longleftrightarrow & \text{(sentence translation)} \\
\mathcal{J} \models_{\mathrm{T}} \ell \leq \#\mu(c{\bullet}r : c') & \Longleftrightarrow & \text{(ass. } \mu(c{\bullet}r : c') = (d{\bullet}s : d')\text{)} \\
\mathcal{J} \models_{\mathrm{T}} \ell \leq \#(d{\bullet}s : d') & \Longleftrightarrow & \text{(sat. rel. and } d = \gamma(c)\text{)} \\
\forall p \in D^{\mathcal{J}}(\gamma(c)) \,.\, [\![\ell]\!] \leq |N^{\mathcal{J}}(d{\bullet}s : d')(p)| & \Longleftrightarrow & \text{(defn. } \mu\text{)} \\
\forall p \in D^{\mathcal{J}}(\gamma(c)) \,.\, [\![\ell]\!] \leq |N^{\mathcal{J}}(\mu(c{\bullet}r : c'))(p)| & \Longleftrightarrow & \text{(model reduct)} \\
\forall o \in C^{\mathcal{J}|\sigma}(c) \,.\, [\![\ell]\!] \leq |M^{\mathcal{J}|\sigma}(c{\bullet}r : c')(o)| & \Longleftrightarrow & \text{(sat. rel.)} \\
\mathcal{J}|\sigma \models_\Sigma \ell \leq \#c{\bullet}r : c' & &
\end{array}$$

Case: $\varphi \equiv \#c{\bullet}r : c' \leq \ell$ is analogous to the above case.

Case: $\varphi \equiv c{\bullet}r : c'!$ is given by:

$$\begin{array}{lcr}
\mathcal{J} \models_{\mathrm{T}} \sigma(c{\bullet}r : c'!) & \iff & \text{(sentence translation)} \\
\mathcal{J} \models_{\mathrm{T}} \mu(c{\bullet}r : c')! & \iff & \text{(ass. } \mu(c{\bullet}r : c') = (d{\bullet}s : d')\text{)} \\
\mathcal{J} \models_{\mathrm{T}} d{\bullet}s : d'! & \iff & \text{(sat. rel. and } d = \gamma(c)\text{)} \\
\forall p' \in D^{\mathcal{J}}(\gamma(c')) \,.\, \exists p \in D^{\mathcal{J}}(\gamma(c)) \,. & & \\
\quad N^{\mathcal{J}}(d{\bullet}s : d'))(p) = p' & \iff & \text{(defn. } \mu\text{)} \\
\forall p' \in D^{\mathcal{J}}(\gamma(c')) \,.\, \exists p \in D^{\mathcal{J}}(\gamma(c)) \,. & & \\
\quad N^{\mathcal{J}}(\mu(c{\bullet}r : c'))(p) = p' & \iff & \text{(model reduct)} \\
\forall o' \in C^{\mathcal{J}|\sigma}(c') \,.\, \exists o \in C^{\mathcal{J}|\sigma}(c) \,. & & \\
M^{\mathcal{J}|\sigma}(c{\bullet}r : c')(o) = o' & \iff & \text{(sat. rel.)} \\
\mathcal{J}|\sigma \models_{\Sigma} c{\bullet}r : c'! & &
\end{array}$$

Case: $\varphi \equiv \ell \leq \#a\{r_1 : c_1, \ldots, r_n : c_n\}[r_{i_1}, \ldots, r_{i_m}]$ is given by:

$$\begin{array}{l}
\mathcal{J} \models_{\mathrm{T}} \sigma(\ell \leq \#a\{r_1 : c_1, \ldots, r_n : c_n\}[r_{i_1}, \ldots, r_{i_m}]) \\
\iff \quad \text{(sentence translation)} \\
\mathcal{J} \models_{\mathrm{T}} \ell \leq \#\alpha(a\{r_1 : c_1, \ldots, r_n : c_n\})[\rho_\alpha(r_{i_1}), \ldots, \rho_\alpha(r_{i_m})] \\
\iff \quad \text{(ass. } \alpha(a\{r_1 : c_1, \ldots, r_n : c_n\}) = b\{s_1 : d_1, \ldots, s_n : d_n\}\text{))} \\
\mathcal{J} \models_{\mathrm{T}} \ell \leq \#(b\{s_1 : d_1, \ldots, s_n : d_n\})[s_{i_1}, \ldots, s_{i_m}] \\
\iff \quad \text{(sat. rel. and } d_j = \gamma(c_j)\text{)} \\
\forall p_{i_1} \in D^{\mathcal{J}}(\gamma(c_{i_1})), \ldots, p_{i_m} \in D^{\mathcal{J}}(\gamma(c_{i_m})) \,. \\
\quad [\![\ell]\!] \leq |\{u \in B^{\mathcal{J}}(b\{s_1 : d_1, \ldots, s_n : d_n\}) \mid u(s_{i_1}) = p_{i_1}, \ldots, u(s_{i_m}) = p_{i_m}\}| \\
\iff \text{(defn. } \alpha\text{)} \\
\forall p_{i_1} \in D^{\mathcal{J}}(\gamma(c_{i_1})), \ldots, p_{i_m} \in D^{\mathcal{J}}(\gamma(c_{i_m})) \,. \\
\quad [\![\ell]\!] \leq |\{u \in B^{\mathcal{J}}(\alpha(a\{r_1 : c_1, \ldots, r_n : c_n\})) \\
\qquad \mid u(\rho_\alpha(r_{i_1})) = p_{i_1}, \ldots, u(\rho_\alpha(r_{i_m})) = p_{i_m}\}| \\
\iff \text{(model reduct)} \\
\forall o_{i_1} \in C^{\mathcal{J}|\sigma}(c_{i_1}), \ldots, o_{i_m} \in C^{\mathcal{J}|\sigma}(c_{i_m}) \,. \\
\quad [\![\ell]\!] \leq |\{t \in A^{\mathcal{J}|\sigma}(a\{r_1 : c_1, \ldots, r_n : c_n\}) \mid t(r_{i_1}) = o_{i_1}, \ldots, t(r_{i_m}) = o_{i_m}\}| \\
\iff \text{(sat. rel.)} \\
\mathcal{J}| \models_{\Sigma} \ell \leq \#a\{r_1 : c_1, \ldots, r_n : c_n\}[r_{i_1}, \ldots, r_{i_m}]
\end{array}$$

Case: $\varphi \equiv \#a\{r_1 : c_1, \ldots, r_n : c_n\}[r_{i_1}, \ldots, r_{i_m}] \leq \ell$ is analogous to the above case.

□

From this, we obtain our class diagram institution:

Theorem 5.22 (ClDiag is an Institution) ClDiag = (Cl, Inst, *Mult*, $\models$) forms an institution.

Proof. Follows from Lemma 5.5, Lemma 5.16 and Lemma 5.21. □

5.3 Stereotypes for Dynamical Aspects

As we have seen in Section 3.2.1, the railway domain has a variety of concepts that can change over time. It also has a variety of concepts that are static for all time. For

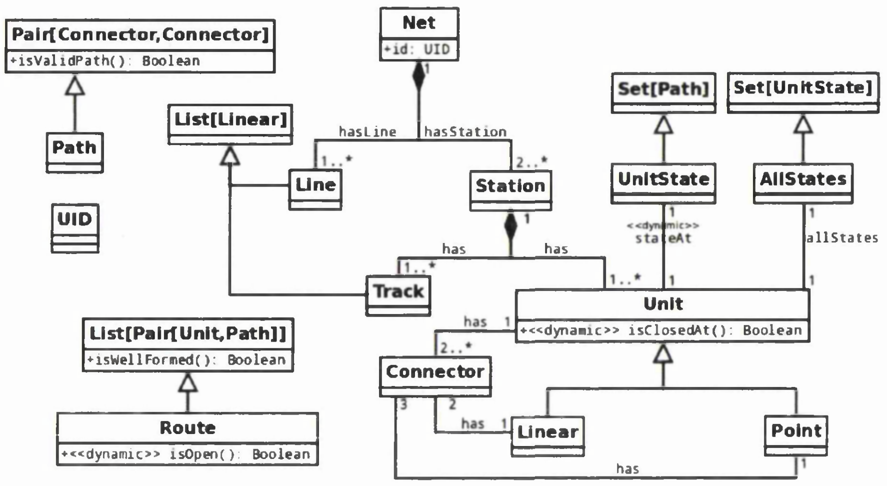


Figure 5.1: A UML class diagram with Stereotypes for Bjørner's DSL.

example, consider the track plan in Figure 3.1 and the UML class diagram in Figure 5.1. Here, the has relation between Connectors and Linear units clearly remains the same over the time we are interested in the system[1]. However, if we consider, for example the StateAt relation between Units and UnitStates, it is clear that the state of a point may change over time, e.g. see Figure 3.6. To allow UML Class Diagrams to capture this dual nature of systems, we introduce a so-called UML *stereotype*, as is common practice for UML [Obj11]. To the "pure" UML class diagram for Bjørner's DSL given in Figure 2.3 we add a stereotype ≪dynamic≫. In Figure 5.1 this appears for the association stateAt and the two operations isClosedAt for routes and isOpen for units. All other elements are considered to be ≪rigid≫. These domain-specific stereotypes are intended to make clear which parts of an object structure complying to the class diagram can change over time, i.e., are ≪dynamic≫ like stateAt, and which parts have to be kept fixed over time, i.e., are ≪rigid≫, e.g., the objects of Net, Station, Line, etc.

To capture these stereotypes within our UML institution, we have developed the general institution construction of a pointed power set institution. This construction factors out a general principle necessary for connecting UML class diagrams with an arbitrary institution capturing system dynamics by capturing the nature of the dynamic stereotype. Namely, this situation of dynamics arises for any reactive formalism that one wants to link to UML class diagrams and that is able to change the system configuration. Therefore, we decided to factor out the mediating construction into an institution independent powerset construction with regards to the models. Here we work with pointed powersets in order to represent initial states. After presenting this construction, we show how we can use it with our class diagram institution to give the required capturing of dynamic and rigid stereotypes.

[1] That is, we do not consider modelling, for example, construction changes to the railway.

5.3.1 An Institution Capturing Dynamics

Let $\mathscr{I} = (\mathrm{Sig}^{\mathscr{I}}, \mathit{Sen}^{\mathscr{I}}, \mathrm{Mod}^{\mathscr{I}}, \models^{\mathscr{I}})$ be an institution. We define an institution $\wp^{\mathrm{i}}_{*}\mathscr{I}$ over $\mathscr{I}$ that inherits the sentences from $\mathscr{I}$. It then has as models pointed powersets $(M_0, \mathcal{M})$ of models of $\mathscr{I}$. For each of these models, we then require that:

1. All inherited sentences have to hold in all models of $\mathcal{M}$, and

2. in each model $(M_0, \mathcal{M})$ over a signature Σ all elements of $\mathcal{M}$ "behave" like M_0 for a certain part Γ of the signature Σ.

The intuitive insight behind this definition, is that the certain part Γ of the signature, for which all models "behave" the same, can be used to capture the rigid elements of a class diagram. Whereas the dynamical aspects are free to change their interpretation throughout the models. Formally we have:

Remark 5.23 Throughout our definition we use the explicit form of a model as a pair $(M_0, \mathcal{M})$. We note that the distinguished model M_0 is included for clarity within the construction. It can be removed from the models, as long as the models in $\mathcal{M}$ all "behave" in the same manner for the elements within Γ.

Definition 5.24 (Signature Category $\mathrm{Sig}^{\wp^{\mathrm{i}}_{*}\mathscr{I}}$) We define the signature category $\mathrm{Sig}^{\wp^{\mathrm{i}}_{*}\mathscr{I}}$ of $\wp^{\mathrm{i}}_{*}\mathscr{I}$ as follows: The objects of $\mathrm{Sig}^{\wp^{\mathrm{i}}_{*}\mathscr{I}}$ are the morphisms $\sigma : \Gamma \to \Sigma$ in $\mathrm{Sig}^{\mathscr{I}}$.

A morphism $(\gamma, \rho) : (\sigma : \Gamma \to \Sigma) \to (\sigma' : \Gamma' \to \Sigma')$ of $\mathrm{Sig}^{\wp^{\mathrm{i}}_{*}\mathscr{I}}$ consists of morphisms $\gamma : \Gamma \to \Gamma'$ and $\rho : \Sigma \to \Sigma'$ in $\mathrm{Sig}^{\mathscr{I}}$ such that $\sigma' \circ \gamma = \rho \circ \sigma$.

We then define that sentences of $\wp^{\mathrm{i}}_{*}\mathscr{I}$ are inherited from the underlying category, namely from the signature of the target of the underlying signature morphism:

Definition 5.25 (Sentences Functor for $\mathit{Sen}^{\wp^{\mathrm{i}}_{*}\mathscr{I}}$) Define the sentence functor $\mathit{Sen}^{\wp^{\mathrm{i}}_{*}\mathscr{I}} :$ $\mathrm{Sig}^{\wp^{\mathrm{i}}_{*}\mathscr{I}} \to \mathrm{Set}$ of $\wp^{\mathrm{i}}_{*}\mathscr{I}$ by $\mathit{Sen}^{\wp^{\mathrm{i}}_{*}\mathscr{I}}(\sigma : \Gamma \to \Sigma) = \mathit{Sen}^{\mathscr{I}}(\Sigma)$ and $\mathit{Sen}^{\wp^{\mathrm{i}}_{*}\mathscr{I}}(\gamma, \rho) = \mathit{Sen}^{\mathscr{I}}(\rho)$.

Considering models, we now define the model category and the functor $\mathrm{Mod}^{\wp^{\mathrm{i}}_{*}\mathscr{I}}$ of $\wp^{\mathrm{i}}_{*}\mathscr{I}$ as:

Definition 5.26 (Model Category and Functor of $\wp^{\mathrm{i}}_{*}\mathscr{I}$) We define the contra-variant model functor $\mathrm{Mod}^{\wp^{\mathrm{i}}_{*}\mathscr{I}} : (\mathrm{Sig}^{\wp^{\mathrm{i}}_{*}\mathscr{I}})op \to \mathrm{Cat}$ of $\wp^{\mathrm{i}}_{*}\mathscr{I}$ as follows:

Let $(\sigma : \Gamma \to \Sigma)$ be a signature in $|\mathrm{Sig}^{\wp^{\mathrm{i}}_{*}\mathscr{I}}|$. For mapping σ we have: The objects of $\mathrm{Mod}^{\wp^{\mathrm{i}}_{*}\mathscr{I}}(\sigma)$ are the pairs $(M_0, \mathcal{M})$ with $M_0 \in |\mathrm{Mod}^{\mathscr{I}}(\Sigma)|$ and $\mathcal{M}$ a sub-*set* of $|\mathrm{Mod}^{\mathscr{I}}(\Sigma)|$ such that $M_0 \in \mathcal{M}$ and the model reduct $\mathrm{Mod}^{\mathscr{I}}(\sigma)(M) = \mathrm{Mod}^{\mathscr{I}}(\sigma)(M_0)$ for each $M \in \mathcal{M}$. A morphism of $\mathrm{Mod}^{\wp^{\mathrm{i}}_{*}\mathscr{I}}(\sigma)$ from $(M_0, \mathcal{M})$ to $(M_0', \mathcal{M}')$ is a morphism $\chi_0 : M_0 \to M_0'$ in $\mathrm{Mod}^{\mathscr{I}}(\sigma)$ where $\mathcal{M}$ and $\mathcal{M}'$ satisfy the condition that for every $M \in \mathcal{M}$ there is a $M' \in \mathcal{M}'$ and a morphism $\chi : M \to M'$.

Then, let $(\gamma, \rho) : (\sigma : \Gamma \to \Sigma) \to (\tau : \Delta \to \mathrm{T})$ be a morphism in $\mathrm{Sig}^{\wp^{\mathrm{i}}_{*}\mathscr{I}}$. For mapping (γ, ρ) we have: $\mathrm{Mod}^{\wp^{\mathrm{i}}_{*}\mathscr{I}}(\gamma, \rho)(N_0, \mathcal{N}) = (\mathrm{Mod}^{\mathscr{I}}(\rho)(N_0), \{\mathrm{Mod}^{\mathscr{I}}(\rho)(N) \mid N \in \mathcal{N}\})$ and $\mathrm{Mod}^{\wp^{\mathrm{i}}_{*}\mathscr{I}}(\gamma, \rho)(\psi_0) = \mathrm{Mod}^{\mathscr{I}}(\rho)(\psi_0)$.

Finally, we consider the satisfaction relation for $\wp^{\mathsf{i}}_{*}\mathscr{I}$:

Definition 5.27 (Satisfaction relation for $\wp^{\mathsf{i}}_{*}\mathscr{I}$) For each $(\sigma : \Gamma \to \Sigma) \in |\mathrm{Sig}^{\wp^{\mathsf{i}}_{*}\mathscr{I}}|$, we define the satisfaction relation $- \models^{\wp^{\mathsf{i}}_{*}\mathscr{I}}_{\sigma:\Gamma\to\Sigma} - \subseteq |\mathrm{Mod}^{\wp^{\mathsf{i}}_{*}\mathscr{I}}(\sigma : \Gamma \to \Sigma)| \times \mathit{Sen}^{\wp^{\mathsf{i}}_{*}\mathscr{I}}(\sigma : \Gamma \to \Sigma)$ of $\wp^{\mathsf{i}}_{*}\mathscr{I}$ by $(M_0, \mathcal{M}) \models^{\wp^{\mathsf{i}}_{*}\mathscr{I}}_{\sigma:\Gamma\to\Sigma} \varphi \iff \forall M \in \mathcal{M} \,.\, M \models^{\mathscr{I}}_{\Sigma} \varphi$.

Here, we see that satisfaction simply lifts and holds if all models (including M_0 as we have $M_0 \in \mathcal{M}$) satisfy the given sentence φ. From this, we can show the satisfaction condition and hence, that $\wp^{\mathsf{i}}_{*}\mathscr{I}$ forms an institution.

Theorem 5.28 ($\wp^{\mathsf{i}}_{*}\mathscr{I}$ is an institution) $\mathrm{Sig}^{\wp^{\mathsf{i}}_{*}\mathscr{I}}, \mathit{Sen}^{\wp^{\mathsf{i}}_{*}\mathscr{I}}, \mathrm{Mod}^{\wp^{\mathsf{i}}_{*}\mathscr{I}}, \models^{\wp^{\mathsf{i}}_{*}\mathscr{I}}$) forms an institution. □

Proof. Given the definitions above, it remains to show the satisfaction condition, that is:

$$\mathrm{Mod}^{\wp^{\mathsf{i}}_{*}\mathscr{I}}(\gamma,\rho)(M_0,\mathcal{M}) \models^{\wp^{\mathsf{i}}_{*}\mathscr{I}}_{\sigma:\Gamma\to\Sigma} \varphi \quad\iff\quad (M_0,\mathcal{M}) \models^{\wp^{\mathsf{i}}_{*}\mathscr{I}}_{\sigma':\Gamma'\to\Sigma'} \mathit{Sen}^{\wp^{\mathsf{i}}_{*}\mathscr{I}}(\gamma,\rho)(\varphi)\ .$$

for each model $(M_0, \mathcal{M}) \in |\mathrm{Mod}^{\wp^{\mathsf{i}}_{*}\mathscr{I}}(\sigma' : \Gamma' \to \Sigma')|$, signature morphism $(\gamma, \rho) : (\sigma : \Gamma \to \Sigma) \to (\sigma' : \Gamma' \to \Sigma') \in \mathrm{Sig}^{\wp^{\mathsf{i}}_{*}\mathscr{I}}$ and sentence $\varphi \in \mathit{Sen}^{\mathscr{I}}(\sigma : \Gamma \to \Sigma)$.

Let $(\gamma, \rho) : (\sigma : \Gamma \to \Sigma) \to (\sigma' : \Gamma' \to \Sigma')$ be a signature morphism in $\mathrm{Sig}^{\wp^{\mathsf{i}}_{*}\mathscr{I}}$, φ be a sentence in $\mathit{Sen}^{\mathscr{I}}(\sigma : \Gamma \to \Sigma)$ and finally let $(M_0, \mathcal{M})$ be a model $\in |\mathrm{Mod}^{\wp^{\mathsf{i}}_{*}\mathscr{I}}(\sigma' : \Gamma' \to \Sigma')|$. Then we have:

$$\begin{array}{lcr}
\mathrm{Mod}^{\wp^{\mathsf{i}}_{*}\mathscr{I}}(\gamma,\rho)(M_0,\mathcal{M}) \models^{\wp^{\mathsf{i}}_{*}\mathscr{I}}_{\sigma} \varphi & \iff & \text{(defn. of model reduct)} \\
(\mathrm{Mod}^{\mathscr{I}}(\rho)(M_0), & & \\
\{\mathrm{Mod}^{\mathscr{I}}(\rho)(M) \mid M \in \mathcal{M}\}) \models^{\wp^{\mathsf{i}}_{*}\mathscr{I}}_{\sigma} \varphi & \iff & \text{(defn. of sat. rel.)} \\
\forall M \in \{\mathrm{Mod}^{\mathscr{I}}(\rho)(M) \mid M \in \mathcal{M}\} \,.\, M \models^{\mathscr{I}}_{\Sigma} \varphi & \iff & \text{(defn. set comprehension)} \\
\forall M \in \mathcal{M} \,.\, \mathrm{Mod}^{\mathscr{I}}(\rho)(M) \models^{\mathscr{I}}_{\Sigma} \varphi & \iff & \text{(sat. cond. for } \mathscr{I}\text{)} \\
\forall M \in \mathcal{M} \,.\, M \models^{\mathscr{I}}_{\Sigma'} \mathit{Sen}^{\mathscr{I}}(\rho)(\varphi) & \iff & \text{(defn. of sat. rel.)} \\
(M_0,\mathcal{M}) \models^{\wp^{\mathsf{i}}_{*}\mathscr{I}}_{\sigma':\Gamma'\to\Sigma'} \mathit{Sen}^{\wp^{\mathsf{i}}_{*}\mathscr{I}}(\gamma,\rho)(\varphi)\ . & &
\end{array}$$

□

5.3.2 Using $\wp^{\mathsf{i}}_{*}\mathscr{I}$ for UML Class Diagrams with Rigidity Constraints

We now apply this construction to the UML class diagram institution ClDiag to represent rigidity. We note that in the following, we assume all signature elements of the class diagram to be consistently and completely annotated with the stereotypes «rigid» and «dynamic». For consistency rules we mean, for example, that: Boolean is «rigid» and if a class c is «dynamic», then List[c] is «dynamic» as well, etc. Also, classes c that are added by the signature closure are «rigid» unless forced to be «dynamic» by consistency rules.

The first step of our approach is to form a class net Γ consisting only of those elements that are stereotyped with «rigid». Along with this, we additionally form the full class net Σ. As the signatures of ClDiag are set-based, there is a simple inclusion morphism from the rigid class net Γ to the full class net Σ. We take this inclusion as the signature σ in $\wp^{\mathsf{i}}_{*}$ClDiag. Thus, the meaning of all elements stereotyped with «rigid» is fixed by their meanings in the distinguished state. In fact, for representing UML class diagrams with rigidity constraints it is enough to work with those signatures in $\wp^{\mathsf{i}}_{*}$ClDiag that are inclusions in the category of signatures Cl of ClDiag. We denote this institution by $\wp^{\hookrightarrow}_{*}$ClDiag.

From this point onward, we will consider how we can map from $\wp^{\hookrightarrow}_{*}$ClDiag into ModalCasl.

Remark 5.29 At this point we note that there is a trivial simple institution comorphism from $\wp^{\mathsf{i}}_{*}\mathscr{I}$ to $\mathscr{I}$ (which also applies to $\wp^{\hookrightarrow}_{*}$ClDiag). Namely, we define the functor $\Phi^{\wp^{\mathsf{i}}_{*}\mathscr{I},\mathscr{I}} : \mathrm{Sig}^{\wp^{\mathsf{i}}_{*}\mathscr{I}} \to \mathrm{Pres}^{\mathscr{I}}$ by $\Phi^{\wp^{\mathsf{i}}_{*}\mathscr{I},\mathscr{I}}(\sigma : \Gamma \to \Sigma) = (\Sigma, \emptyset)$ and $\Phi^{\wp^{\mathsf{i}}_{*}\mathscr{I},\mathscr{I}}(\gamma, \rho) = \rho$. We then define the natural transformation $\alpha^{\wp^{\mathsf{i}}_{*}\mathscr{I},\mathscr{I}} : \mathit{Sen}^{\wp^{\mathsf{i}}_{*}\mathscr{I}} \to \mathit{Sen}^{\mathscr{I}} \circ \Phi^{\wp^{\mathsf{i}}_{*}\mathscr{I},\mathscr{I}}$ by $\alpha^{\wp^{\mathsf{i}}_{*}\mathscr{I},\mathscr{I}}_{\sigma:\Gamma\to\Sigma}(\varphi) = \varphi$ for $\varphi \in \mathit{Sen}^{\mathscr{I}}(\Sigma)$. Finally, we define the natural transformation $\beta^{\wp^{\mathsf{i}}_{*}\mathscr{I},\mathscr{I}}$: $\mathrm{Mod}^{\mathscr{I}} \circ (\Phi^{\wp^{\mathsf{i}}_{*}\mathscr{I},\mathscr{I}})^{op} \to \mathrm{Mod}^{\wp^{\mathsf{i}}_{*}\mathscr{I}}$ by $\beta^{\wp^{\mathsf{i}}_{*}\mathscr{I},\mathscr{I}}_{\sigma:\Gamma\to\Sigma}(M) = (M, \{M\})$ and $\beta^{\wp^{\mathsf{i}}_{*}\mathscr{I},\mathscr{I}}_{\sigma:\Gamma\to\Sigma}(\chi) = \chi$. Then, by definition we obtain:

Theorem 5.30 $\mu^{\wp^{\mathsf{i}}_{*}\mathscr{I},\mathscr{I}} = (\Phi^{\wp^{\mathsf{i}}_{*}\mathscr{I},\mathscr{I}}, \alpha^{\wp^{\mathsf{i}}_{*}\mathscr{I},\mathscr{I}}, \beta^{\wp^{\mathsf{i}}_{*}\mathscr{I},\mathscr{I}})$ forms a simple institution comorphism from $\wp^{\mathsf{i}}_{*}\mathscr{I}$ to $\mathscr{I}$. □

5.4 From Class Diagrams with Rigidity Constraints to Modal CASL

In the following, we give a sketch of a simple institution comorphism from UML class diagrams with rigidity constraints to ModalCasl: Let $(\Gamma \subseteq \Sigma)$ be a signature[2] in $|\mathrm{Sig}^{\wp^{\hookrightarrow}_{*}\mathrm{ClDiag}}|$ with corresponding class net $\Sigma = ((C, \leq_C), K, P, M, A)$. We map $(\Gamma \subseteq \Sigma)$ to the following ModalCasl presentation:

$$\Phi(\Gamma \subseteq \Sigma) = (((S, \mathit{TF}, \mathit{PF}, P, \leq), F_{\mathit{Rigid}}, P_{\mathit{Rigid}}, \mathit{modalities}, \mathit{modalitySorts}), \mathit{Ax}).$$

Throughout this section, we illustrate the mapping using examples from the class diagram for Bjørner's DSL as given in Figure 5.1. The full result of the mapping is given in Appendix B.

We begin by considering the components of a class diagram that are connected via generalisation. Let us denote the connected components of $\leq_C$ by $cc(\leq_C)$ and the connected component of a class name[3] $c \in C$ by $[c]_{\leq_C}$, i.e., $cc(\leq_C) = \{[c]_{\leq_C} \mid c \in C\}$. We then define the sorts of our ModalCasl signatures as

$$S = C \cup cc(\leq_C).$$

[2]Here we notice the use of $\subseteq$ as for our construction we only consider signature morphisms that are inclusions.

[3]Here we assume $[c]_{\leq_C} = \{t\}$, if t is the top sort of the component of c.

Here, the possibly additional top sorts $cc(\leq_C)$ mediate between the CASL semantics (which allows for models M with $s_M \not\subseteq t_M$ for sorts $s \leq t$) and the UML class diagram semantics (which requires models I with $s^I \subseteq t^I$ for $s \leq_C t$).

The sub-sort relation $\leq$ is given by the union of the following relations:

$$\leq \; = \; \leq_C \cup \{(c, [c]_{\leq_C}) \mid c \in C\} \cup \{([c]_{\leq_C}, [c]_{\leq_C}) \mid c \in C\}$$

We note that as part of our translation, we often include and (possibly) instantiate specifications from the CASL Basic Datatypes [BM04] for any built-in types from the class diagram, e.g. for the class Boolean and class involving type formers such as Pair$[c_1, c_2]$. Such an instantiation, for example, for the sort *PairConnectorConnector* is specified by instantiating the CASL PAIR specification and renaming the sort name:

PAIR[**sort** *Connector*][**sort** *Connector*] **with**
    Pair[**sort** *Connector*][**sort** *Connector*] $\mapsto$ *PairConnectorConnector*

Here, we assume that the semantics of each predefined type and type former is an element of the model class of the respective monomorphic CASL datatype.

From the elements of the comorphism we have given so far, the following is a result of applying these translations to our running example for Bjørner's DSL:

%% Classes:
sorts *Net*, *Station*, *Unit*, ..., *UID*
%% Hierarchy:
sorts *Point*, *Linear* < *Unit*; ...; *Route* < *ListPairUnitPath*

We can see that there are sorts for each of the classes within the class diagram, and that the class hierarchy has been captured in MODALCASL by subsorting.

Next, the total function symbols *TF* simply comprise of the instance specification declarations:

$$TF = \{k : c \mid k : c \in K\}.$$

Similarly, the partial function symbols *PF* comprise of the property declarations:

$$PF = \{p : c \times c_1 \cdots \times c_n \rightarrow ? \; c' \mid c.p(x_1 : c_1, \ldots, x_n : c_n) : c' \in P\}.$$

For these functions, the classification into "rigid" and "flexible" can be obtained directly from the signature $(\Gamma \subseteq \Sigma) \in |\mathrm{Sig}^{\wp^{\hookrightarrow}_{*}\mathrm{ClDiag}}|$. That is, we gain F_{Rigid} by taking the instance specifications and property declarations given in Γ. Considering Bjørner's DSL, there are only property declarations and hence we obtain:

%% Properties:
rigid op *id* : *Net* $\rightarrow ?$ *UID*
flexible ops *isClosedAt* : *Unit* $\rightarrow ?$ *Boolean*;
...

Here, we can see that *id* is rigid and *isClosedAt* is flexible, as prescribed by the stereotype in the class diagram.

Next, the predicate symbols P comprise the composition declarations M and the association declarations A. Again, P_{Rigid} can be obtained by considering those elements in Γ. As well as these, a predicate *isAlive* is added for each sort in S. This predicate is used to model "flexible" sort interpretations in MODALCASL. It arises from the interpretation of classes as objects in the class diagram. The predicate *isAlive* is rigid if the corresponding class is rigid. Overall, we obtain:

$$\begin{aligned} P = \quad & \{r : c \times c' \mid c{\bullet}r : c' \in M\} \cup && \text{(for compositions)} \\ & \{a : c_1 \times \cdots \times c_n \mid a(r_1 : c_1, \ldots, r_n : c_n) \in A\} \cup && \text{(for associations)} \\ & \{\mathit{isAlive} : c \mid c \in C\} && \text{(isAlive)} \end{aligned}$$

Considering the running example, we obtain the following MODALCASL:

%% Compositions:
rigid preds *__has__* : *Station* × *Unit*; *__has__* : *Station* × *Track*;
...
%% Associations:
rigid preds *__has__* : *Unit* × *Connector*; *__has__* : *Linear* × *Connector*; ...
...
%% Is Alive preds:
rigid preds *isAlive* : *Net*; *isAlive* : *Station*; *isAlive* : *Unit*; ...; *isAlive* : *UID*;

For our institution comorphism, the *modalities* set *M*for our MODALCASL is just $\{\epsilon\}$, and the *modalitySorts* set *SM* is simply empty. That is the models have exactly one accessibility relation defined between worlds. Naturally, representing class diagrams in MODALCASL imposes no constraints on this relation.

Finally, we need to add some axioms to our MODALCASL for the constraints from our class diagram institution and also to capture the multiplicities constraints from the class diagram institution. The first group of axioms stipulates that arguments of operations and predicates need to be "alive". That is, for operations and predicates capturing the class diagram elements of instance specifications, property declarations, composition declarations and association declarations. Whilst the second group of axioms corresponds to the condition on Σ-instance nets that each instance has at most one owner. Overall we gain the following:

$$\begin{aligned} Ax \quad = \quad & \{\mathit{isAlive}(k : c) \mid k : c \in K\} \\ \cup \quad & \{\forall x : c, x_1 : c_1, \ldots, x_n : c_n.\mathit{def}\ p(x, x_1, \ldots, x_n) \implies \\ & \quad \mathit{isAlive}(x) \wedge \mathit{isAlive}(x_1) \wedge \ldots \mathit{isAlive}(x_n) \mid \\ & \qquad c.p(x_1 : c_1, \ldots, x_n : c_n) : c' \in P\} \\ \cup \quad & \{\forall x : c, x' : c'.r(x, x') \implies \mathit{isAlive}(x) \wedge \mathit{isAlive}(x') \mid c{\bullet}r : c' \in M\} \\ \cup \quad & \{\forall x_1 : c_1, \ldots, x_n : c_n.a(x_1, \ldots, x_n) \implies \\ & \quad \mathit{isAlive}(x_1) \wedge \ldots \mathit{isAlive}(x_n) \mid a(r_1 : c_1, \ldots, r_n : c_n) \in A\} \\ \cup \quad & \{\forall x_1 : [c_1]_{\leq_C}, x_2 : [c_2]_{\leq_C}, x : [c'_1]_{\leq_C} . r(x_1, x) \wedge r(x_2, x) \Rightarrow x_1 = x_2 \mid \\ & \quad c_1{\bullet}r_1 : c'_1, c_2{\bullet}r_2 : c'_2 \in M,\ [c_1]_{\leq_C} = [c_2]_{\leq_C},\ [c'_1]_{\leq_C} = [c'_2]_{\leq_C}\} \end{aligned}$$

Then, as we have seen, the sentences of $\wp_*^{\hookrightarrow}$ClDiag are the sentences of ClDiag, that is, the multiplicity constraints imposed in the class diagram. These can be systematically encoded in first order logic using "poor man's counting", e.g. [EFT94], by providing the necessary number of variables. We note that we do not use a standard specification of numbers as this would increase the axiomatic base for automatic verification (see Chapter 6). For example, the constraint $\#(c\bullet r : c') \geq n$ is translated to the following axiom:

$$\forall x : c \, \exists y_1, \ldots, y_n : c' \, . \, \mathit{pwDifferent}(y_1, \ldots, y_n) \wedge \; r(x, y_1) \wedge \cdots \wedge r(x, y_n)$$

were, $\mathit{pwDifferent}(y_1, \ldots, y_n)$ encodes $y_i \neq y_j$ for $i \neq j$, $1 \leq i, j \leq n$.

This axiom states that for every value x in c we find at least n different values in c' of which x is related via r. A typical example of the mapping for a composition constraint from Bjørner's DSL is:

$$\#(\mathit{Station}\bullet\mathit{station} : \mathit{Unit}) \geq 1$$

where the resulting ModalCasl axiom would be:

$$\forall s : \mathit{Station}.\exists u1 : \mathit{Unit}.\mathit{station}(s, u1).$$

Similarly, an example of a constraint on an association would be:

$$2 \leq \#\mathsf{has}(\mathsf{connector} : \mathsf{Connector}, \mathsf{unit} : \mathsf{Unit})[\mathsf{unit}]$$

and the resulting ModalCasl axiom would be:

$$\forall u : \mathit{Unit}. \exists c1, c2 : \mathit{Connector}. \, c1 \neq c2 \wedge \mathit{has}(c1, u) \wedge \mathit{has}(c2, u).$$

Finally, with regards to models, we define how to turn a ModalCasl model into a corresponding instance net. Let $\mathcal{M}$ be a ModalCasl model in $|\mathrm{Mod}^{\mathrm{ModalCasl}}(\Phi(\Gamma \subseteq \Sigma))|$, let W be the sets of worlds in $\mathcal{M}$, $i \in W$ the initial world, and M_w the Casl models associated with each world $w \in W$.

In the following, we use the abbreviations $\lceil x \rceil = (inj_{\mathsf{c},[\mathsf{c}]})_{M_w}(x)$ for $x \in c_{M_w}$; and $\lfloor x \rfloor = (pr_{[\mathsf{c}],c})_{M_w}(x)$ for $x \in [c]_{M_w}$.

We first define for each w, how to turn its associated Casl model M_w into a Σ-*instance net* $\beta(M_w) = (C^{M_w}, K^{M_w}, P^{M_w}, M^{M_w}, A^{M_w})$:

- If c does not involve a type former: $C^{M_w}(c) = \{\lceil x \rceil \mid x \in c_{M_w}, \mathit{isAlive}(x)\}$.

  If c involves a type former, we take the representation of the corresponding type in the instance net.

- $K^{M_w}(k : c) = \lceil (k_{\langle\rangle,c})_{M_w} \rceil$.

- $P^{M_w}(c.p(x_1 \;:\; c_1, \ldots, x_n \;:\; c_n) \;:\; c')(y, y_1, \ldots, y_n) \;=\; \lceil (p_{\langle c, c_1, \ldots, c_n \rangle, c'})_{M_w}(\lfloor y \rfloor, \lfloor y_1 \rfloor, \ldots, \lfloor y_n \rfloor) \rceil$ for all $y \in C^{M_w}(c)$, $y_1 \in C^{M_w}(c_1), \ldots, y_n \in C^{M_w}(c_n)$.

- $M^{M_w}(c{\bullet}r : c')(x) = \{\lceil y \rceil \mid (r_{c,c'})_{M_w}(\lfloor x \rfloor, y)\}$ for all $x \in C^{M_w}(c)$.
- $A^{M_w}(a(r_1 : c_1, \ldots, r_n : c_n)) = \{(\lceil x_1 \rceil, \ldots, \lceil x_n \rceil) \mid (x_1, \ldots, x_n) \in (a_{\langle c_1, \ldots c_n \rangle})_{M_w}\}$. for all $c \in C^{M_w}(c)$.

From the above construction we get that $\beta(\mathcal{M}) = (\beta(M_i), \{\beta(M_w) \mid w \in W\})$ is a model in $|\mathrm{Mod}^{\wp_*^{\hookrightarrow}\mathrm{ClDiag}}(\Gamma \subseteq \Sigma)|$.

We can now give a proof that the mapping we have defined forms a comorphism.

Theorem 5.31 (A comorphism from $\wp_*^{\hookrightarrow}$ClDiag to ModalCasl) The described mapping forms a comorphism.

Proof. Given the definitions above, it remains to show the satisfaction condition, that is:

$$M \models^{\mathscr{I}}_{\mathbf{Sig}(\Phi(\Sigma))} \alpha(\varphi) \iff \beta_\Sigma(M) \models^{\mathscr{I}}_{\Sigma} \varphi \,.$$

The proof follows from a case distinction over φ. Here we sketch the case of $\ell \leq \#c{\bullet}r$.

Case: $\varphi \equiv \ell \leq \#c{\bullet}r : c'$ is given by:

$$\begin{aligned}
&M \models^{\mathscr{I}}_{\mathbf{Sig}(\Phi(\Gamma\subseteq\Sigma))} \alpha(\ell \leq \#c{\bullet}r : c')\\
&\iff \text{(defn. } \alpha)\\
&M \models^{\mathscr{I}}_{\mathbf{Sig}(\Phi(\Gamma\subseteq\Sigma))} \forall x : c\ \exists y_1, \ldots, y_n : c' \,.\, \mathit{pwDifferent}(y_1, \ldots, y_n)\\
&\qquad\qquad\qquad \wedge r(x, y_1) \wedge \cdots \wedge r(x, y_n)\\
&\iff \text{(defn. sat. rel.)}\\
&\forall w \in W . M_W \models^{\mathscr{I}}_{\mathbf{Sig}(\Phi(\Gamma\subseteq\Sigma))} \forall x : c\ \exists y_1, \ldots, y_n : c' \,.\, \mathit{pwDifferent}(y_1, \ldots, y_n)\\
&\qquad\qquad\qquad \wedge r(x, y_1) \wedge \cdots \wedge r(x, y_n)
\end{aligned}$$

This axiom ensures that the underlying carrier sets of c' in all models have at least l distinct elements. It also ensures that these elements are in relation r with the carrier set elements of c. If we now consider the right hand side, we obtain the same restriction for the underlying carrier sets of the instance net:

$$\begin{array}{lcr}
\beta_\Sigma(M) \models^{\mathscr{I}}_{\sigma:\Gamma\to\Sigma} \ell \leq \#c{\bullet}r : c' & \iff & \text{(defn. } \beta)\\
(\beta(M_i), \{\beta(M_w) \mid w \in W\} \models^{\mathscr{I}}_{\sigma:\Gamma\to\Sigma} \ell \leq \#c{\bullet}r : c' & \iff & \text{(defn. sat. rel.)}\\
\forall m \in \{\beta(M_w) \mid w \in W\} . m \models^{\mathscr{CLD}}_{\Sigma} \ell \leq \#c{\bullet}r : c' & \iff & \text{(defn. set comprehension)}\\
\forall w \in W \beta(M_w) \models^{\mathscr{CLD}}_{\Sigma} \ell \leq \#c{\bullet}r : c' & &
\end{array}$$

All other cases for φ follow in a similar manner. □

5.5 Crafting a Formal DSL

To conclude this chapter, we now demonstrate how to create a formal DSL with the techniques developed in the thesis so far. This process, outlined in Figure 5.2, fits with the methodology we are presenting in this thesis.

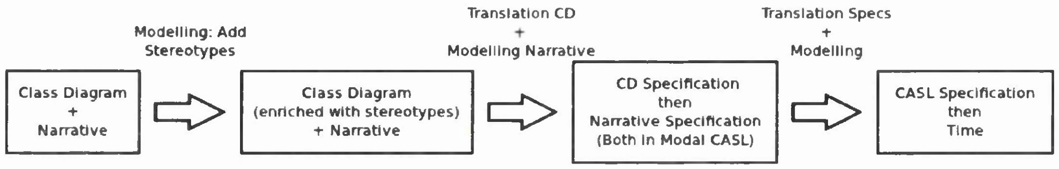

Modelling: Add Stereotypes
Translation CD + Modelling Narrative
Translation Specs + Modelling
Class Diagram + Narrative
Class Diagram (enriched with stereotypes) + Narrative
CD Specification then Narrative Specification (Both in Modal CASL)
CASL Specification then Time

Figure 5.2: Process for capturing a DSL.

Step 1. Start by creating a UML class diagram giving concepts and relationships between concepts within the desired domain. Add an accompanying narrative to describe dynamical aspects as illustrated in Chapter 3. These documents are commonly used in industry, and hence are often already available, see, e.g., [Rai10]. Next, in a modelling step, add stereotypes (as discussed in Section 5.3). These stereotypes are used to describe elements which remain static or are dynamic within the system.

Result: Class diagram and Narrative.

Step 2. Using the comorphism defined in Section 5.4, translate the class diagram into ModalCasl. Next, extend the resulting ModalCasl specification by a modelling of any narrative elements not captured within the class diagram.

For example, considering the narrative of what it means for a *Route* to be "open" from Bjørner's DSL we can produce the following ModalCasl specification:

flexible op $isOpen : Route \rightarrow ? \; Boolean$
$\forall \; r : Route; \; upp : UnitPathPair; \; u : Unit; \; us : UnitState; \; p : Path$
$\bullet \; isOpen(r) = tt$
$\quad \Leftrightarrow r \; has \; upp \wedge upp \; getUnit \; u \wedge upp \; getPath \; p$
$\qquad \wedge \; u \; stateAt \; us \wedge us \; has \; p$

Also, at this point any verification conditions required can be modelled and added to the specification.

Result: ModalCasl specification capturing the class diagram and Narrative.

Step 3. Finally, the last step of the process is to apply the existing ModalCasl to Casl comorphism [Mos02, Mos04a] to gain a Casl Specification of the DSL. This will allow connection to multiple theorem provers for later use in verification. If appropriate, one can also specify in Casl how worlds evolve, e.g., by adding a loose specification of discrete time.

Result: Casl specification of the formal DSL.

To illustrate the overall result of this process, Appendix C.1 gives the full translation of Bjørner's DSL into Casl. In this specification, we have both instantiated various specification capturing types, e.g. Lists and Pairs, and also given a loose specification of time as the modality gained from ModalCasl. Finally, Appendix C.2 also gives an example model of the track plan from Figure 3.1 formulated using this Casl DSL. We note that in these specifications, several of the operations and predicates have had their profiles changed to use mix fix notation for ease of readability.

Chapter 6

Supporting Verification of DSLs

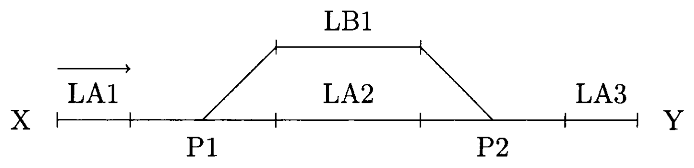
Contents

In this chapter, we explore automatic verification for domain specific languages. We show that via careful design and extension of a domain specific language, it is possible to automatically check properties over models formulated in the DSL. Concretely, we illustrate the approach using Bjørner's domain specific language for the railway domain captured in CASL. As we have seen, such a domain specific language is a loose specification, the logical closure of which we regard as implicitly encoded "domain knowledge". We show that we can systematically exploit this "domain knowledge" to allow for successful automatic verification.

6.1 Railway Dynamics

As presented thus far, Bjørner's DSL has been translated to CASL with the correct features for modelling the layout and static structure of a railway track plan. We now consider an extension to Bjørner's DSL. This extension allows the formulation of elements related to dynamic railway control. In particular, we consider the elements of a control table and release table. Together with a track plan, these form what is commonly referred to as a *s*cheme plan. Figure 6.1 gives the *s*cheme plan for a simple pass through station as considered by Moller et al. [MNR$^+$13].

Such a scheme plan is used as a design document for implementation of the main controller for railways, namely an interlocking [KR01]. An interlocking system gathers



Route	Clear	Normal	Reverse
RA1	LA1, P1, LA2	P1	
RA2	P2, LA3	P2	
RB1	LA1, P1, LB1		P1
RB2	P2, LA3		P2

Route	Point(Cleared By)
RA1	P1(P1)
RA2	P2(P2)
RB1	P1(LA3)
RB2	P2(LA3)

Figure 6.1: A Scheme Plan for a Simple Pass Through Station. Top: Track Plan, Bottom Left: Control Table, Bottom Right: Release Table.

train locations, and sends out commands to control signal aspects and point positions. The control table aspect of a scheme plan determines how the interlocking system sets signals and points for routes. Each route has a corresponding signal that allows entry into the route. For each route, there is one row in the control table describing the conditions under which the corresponding signal can show proceed for that route. For example, the first row in the control table for the pass through station in Figure 6.1 states that route $RA1$ is open (or can be set) if units $LA1$, $P1$ and $LA2$ are not occupied, and point $P1$ is in normal position. The interlocking also allocates so called locks on points to particular route requests. These locks ensure that the point remain locked in position. Such locks are then released according to the information contained in the release table. For example, the first row of the release table in Figure 6.1 states that for route $RA1$ the point $P1$ can be released by the point itself becoming clear. Releasing of these locks allows the corresponding points to be configured and used within another route. The checking of the logic of these tables with respect the topology of the track plan is vital in avoiding train collisions. Later in Section 6.1.3 we will see that this forms the basis of our verification problem.

Finally, the last element in the dynamical aspects of railways is that of train movements. Above, we have described how access to certain routes is granted, and areas of tracks can be released. This follows conventional railway signalling [KR01]. However, the newer ETCS [ERT02] standard builds on this conventional signalling with the notion of a movement authority. A movement authority can be thought of as an area of a railway for which a given train has been granted access to travel along. For example, a train may be granted access to move along units LA1 and P1 in Figure 6.1. At any given time, a movement authority (or sequence of regions) can be assigned to a train. Before a train is allowed to move through a railway, it must gain a movement authority for the parts of the railway it would like to use. The standard operation of interlockings for assigning movement authorities to trains is given by the following rules:

- Extension: Initially, no train has a movement authority. A request can be made for a train to travel along a particular route. If the route is available (as dictated by the control table) then the trains movement authority can be extended to include that route. The movement authority for a train can contain multiple routes. For example, a train travelling through our example station may be given a movement authority to use routes RA1 and RA2.

- *R*elease: As a train travels through its movement authority, it releases areas from its authority. A train can release areas of a route depending on the release table for that route. For example, a train travelling on route RA1 through our example station can release unit LA1 and point P1 from its movement authority once it has exited point P1.

Such an approach towards movement authorities allows the example run illustrated in Figure 6.2. This run illustrates that train A releases the use of point P1 before it exits the full route RA1. This allows for train B to use route RB1 whilst the end of route RA1 is still in use by train A.

Time	LA1	P1	LA2	LB1	P2	LA3
0	-	-	-	-	-	-
1	$\Longmapsto_a$	-	-	-	-	-
2	-	$\Longmapsto_a$	-	-	-	-
3		-	$\Longmapsto_a$	-	-	-
4	$\Longmapsto_b$	-	$\Longmapsto_a$	-	-	-
5	-	$\Longmapsto_b$	$\Longmapsto_a$	-	-	-
6	-	-	$\Longmapsto_a$	$\Longmapsto_b$	-	-
7	-	-	$\Longmapsto_a$	-	$\Longmapsto_b$	-
8	-	-	$\Longmapsto_a$	-	-	$\Longmapsto_b$
9	-	-	$\Longmapsto_a$	-	-	-
10	-	-	-	-	$\Longmapsto_a$	-
11	-	-	-	-	-	$\Longmapsto_a$
12	-	-	-	-	-	-

Figure 6.2: A time/position diagram for an example run of the station scheme plan.

6.1.1 Modelling Assumptions

Before discussing our modelling approach for movement authorities in detail, we list the assumptions that we have made about the various railway systems involved in our modelling. We assume:

Correct System Operation: We assume that all track side components such as signals and point detection units are fully functional and correctly working. This is simply a separation of concerns, as we do not aim to explore any aspects of fault tolerance within our modelling.

Scheme Plan Restrictions: We impose a number of boundaries for our modelling. That is, we consider complex railway scheme plans, but impose some restrictions on the features of such scheme plans, namely:

- No crossover tracks. We restrict ourselves to linear track units and points as these are the most common railway track units. Also, other approaches have shown that it is possible to model crossovers using a pair of points [MNR$^+$12a] and hence, no interesting modelling challenges remain.
- Optimal Release Tables. We assume that a release table exhibits the property that all points are released "optimally" from a capacity point of view, that is, they are released as soon as the point is cleared by a train if travelling towards the point, or released at the end of the route if travelling away from a point. This assumption allows us to define, as we shall see later, the smallest regions for verification. The assumption also ensures that any verification performed is correct given any release table that releases a lock on a point with any track occurring later than the point. That is, our verification considers the most critical possible scenario.
- Discrete time. All our modelling is based on discrete time only. This is justified as interlockings work in a cycling control manner, where each control loop iteration takes a few seconds depending on the concrete interlocking.
- Two aspect signalling only. In all our models, we consider signals with only two aspects, namely "Stop" and "Proceed". These are indicated by a red and green light respectively. Considering two aspect signalling rather than say four aspect signalling (see [KR01]) is enough to allow us to consider safety of railway systems, but does not allow us to reason accurately, for example, about train speeds and system capacity.
- Bi-directional Routes are of Equivalent Length. Finally, the current granularity of regions presented in Section 6.1.2 requires that opposing routes that consist of only linear tracks are of the same length. As, in general, most routes contain a point [KR01] we exclude a very small, and often simple, track plans by this restriction.

Human Behaviour: Finally, we make some assumptions on human behaviour. That is, we assume a train driver does not drive a train outside the movement authority it has been granted, and that signallers request a maximum of one route per time step. This again is separation of concerns, and in fact this second assumption is ensured by most interlockings to stop race conditions on routes.

With these assumptions in place, we introduce the notion of a region of a railway. These regions will allow us to correctly model movement authorities.

6.1.2 Introducing Regions for Movement

To allow us to correctly model the extension and release of movement authorities, we introduce the ability to assign certain regions of track to a train. We define the notion of a region to be a collection of units of a track plan and then define a movement authority to be a sequence of such regions. These regions allow areas of a track to be reserved and released in the typical fashion for train control. A movement authority then represents all the regions of a track layout that a train has been granted access to. Such regions are categorised by both the topological routes of a track plan, and the release point from the release table. For example, considering the pass through station in Figure 6.1, we can split the topological routes at the units outlined by the release table to gain the possible regions of the scheme plan. The splitting of the pass through station into regions is illustrated in Figure 6.3.

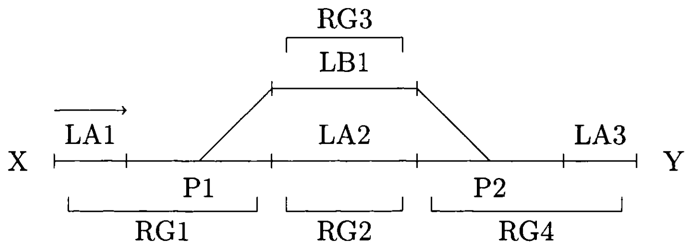


Figure 6.3: Regions of the simple station.

The regions are defined using the release points within the release table to break up a route at the given release unit. Mathematically, we can define the set of regions as the union of the regions of all routes:

Definition 6.1 (Regions of a Route) To compute the regions of a given route R we define the function $Regions : List[Unit] \times List[Unit] \rightarrow Set[Unit]$ as:

$$\begin{array}{lcl} Regions([u_1, u_2, \ldots, u_n], []) & = & \{[u_1, \ldots, u_n]\} \\ Regions([u_1, u_2, \ldots, u_n], [r_1, r_2, \ldots, r_n]) & = & \{[u_1, \ldots, u_k]\} \cup \\ & & Regions([u_{k+1}, \ldots, u_n], [r_2, \ldots, r_n]) \end{array}$$

where $u_k = r_1$. Now, for the given route r, we define the regions of that route to be $Regions(units(r), releaseTable(r))$ where $units(r)$ is simply the units of the route and $releaseTable(r)$ is the topologically ordered list of release units occurring in the release table for r.

For example, considering Figure 6.3 and the route $RA1$, we have: $Regions([LA1, P1, LA2], [P1]) = \{LA1, P1\} \cup Regions([LA2], []) = \{LA1, P1\} \cup \{LA2\}$. These regions allow for modelling of the interlocking behaving taking in to account the release tables. As an example, we can see that region $RG1$ contains all the units of routes $RA1$ and $RB1$ up to and including the release point for $P1$, as given by the release table. Then after this release point we gain regions $RG2$ and $RG3$ representing the remaining units from the routes respectively. Such an approach allows, the train movements given in Figure 6.2 to be captured as the following series of movement authority assignments:

Time	Train A Movement Authority	Train B Movement Authority
0	$\{\}$	$\{\}$
1	$\{RG1, RG2\}$	$\{\}$
2	$\{RG1, RG2\}$	$\{\}$
3	$\{RG2\}$	$\{\}$
4	$\{RG2\}$	$\{RG1, RG3\}$
5	$\{RG2\}$	$\{RG1, RG3\}$
6	$\{RG2\}$	$\{RG3\}$
7	$\{RG2\}$	$\{RG4\}$
8	$\{RG2\}$	$\{RG4\}$
9	$\{RG2\}$	$\{\}$
10	$\{RG4\}$	$\{\}$
11	$\{RG4\}$	$\{\}$
12	$\{\}$	$\{\}$

With this approach in mind, we can now consider what sort of safety properties one might like to prove over the design of a given scheme.

6.1.3 Discussion on Safety Properties

When modelling and verifying railway control components, there are many variations of safety property that could be considered. For example, on the concrete level of an interlocking, one may want to check the concrete property that "Signal x only shows green when route y is free to use" [KMS08, JR10]. Alternatively, when checking the design of a scheme plan, one could follow the approach by Moller et al. [MNR+13] and check that the control table ensures:

- collision-freedom: excluding two trains occupying the same track unit;
- run-through-freedom: stating that whenever a train enters a point, the point is set to cater for this; e.g., when a train travels from P1 LA1 to track LA2, point P1 is set so that it connects itself to LA2 (and not to LB1);
- no-derailment: stating that whenever a train occupies a point, the point doesn't move under it.

Our aim differs slightly, as all these approaches consider classical signalling, whereas we consider the assignment of movement authorities which is outlined by the newer ERTMS standard [ERT02]. We aim to automatically verify train stations modelled using the above approach with respect to the safety principle that "overlapping movement authorities are not assigned at the same time". Such a property is at a higher level than the concrete properties mentioned above. However, as movement authorities are extended depending on the rules of the control table, we do in fact cover the property of collision freedom under the assumption that trains are well behaved. That is, if trains stay within their given movement authority, and movement authorities are proven to never overlap, then we know that two trains cannot occupy the same track unit.

6.2 Modelling Dynamics and Safety in CASL

Taking Bjørner's DSL in CASL as a starting point, we now show how we have extended the CASL models to include a modelling of regions, movement authorities and safety.

6.2.1 Modelling Movement

Firstly, to model regions, we instantiate the specification of *List* with the sort *Units* from Bjørner's DSL (see Appendix D). This list specification follows the CASL standard library [Mos04b], however it contains a smaller number of axioms. This smaller axiomatic base is still enough to prove the required properties in Section 6.6 but improves the performance of various automatic theorem provers. We then make a subsort of this called *Region*. This allows us, as presented above, to model regions as a collection of units. Similarly, we capture movement authorities by instantiating the specification of *List* with these regions and making a subsort *MA*:

```
...
     LIST[sort Unit]
then
     LIST[sort Region]
then
     sort   Region < List[Unit]
     sort   MA < List[Region]
...
```

Next, we model the assignment of a movement authority at a given time as a predicate *assigned* : *MA* × *Time*. We then add an axiom that states that only an empty movement authority can be assigned initially, that is at time 0:

pred *assigned* : *MA* × *Time*

$\bullet\ \forall\ m : MA \bullet assigned(m, 0) \Rightarrow m = []$ %(no_ma_0)%

We note, that for simplicity, regions and movement authorities are both modelled using lists. However, lists are a stronger data type than is necessary for modelling regions, as regions do not reflect all list operations.

Similarly, to model the fact that a train can be granted a movement authority at any time, we add an axiom that states the empty movement authority is always available for a train to request an extension to:

$\bullet\ \forall\ t : Time \bullet assigned([]\ as\ MA,\ t)$ %([]_assigned_at_ all_times)%

We then add an axiom that states when the predicate *assigned* can be true for non empty lists. To simplify the definition, we make use of predicates *canExtend* : *MA* × *Time* and *canReduce* : *MA* × *Time* whose definitions we discuss next. The axiom given below allows for movement authorities at some time *suc*(*t*) to remain assigned as they were at time *t*, become extended from time *t* and also to reduce from time *t*. It also states that only one of these possibilities can occur for a given movement authority at a given time. This definition matches the expected behaviour of movement authorities given in Section 6.1.

$$\begin{array}{l}
\bullet\ \forall\ ma1 : MA;\ t : Time \\
\quad \bullet\ \neg\ ma1 = [] \\
\qquad \Rightarrow assigned(ma1, suc(t)) \\
\qquad\quad \Rightarrow (assigned(ma1, t) \wedge \neg\ canExtend(ma1, t) \wedge \neg\ canReduce(ma1, t)) \\
\qquad\qquad \vee (\neg\ assigned(ma1, t) \wedge canExtend(ma1, t) \wedge \neg\ canReduce(ma1, t)) \\
\qquad\qquad \vee (\neg\ assigned(ma1, t) \wedge \neg\ canExtend(ma1, t) \wedge canReduce(ma1, t))
\end{array}$$
%(assigned_defn)%

The predicate *canExtend* : $MA \times Time$ is true for a given movement authority $ma1$, at a given time t, when the following conditions are met: (1) there exists a movement authority $ma2$ and a route r such that the movement authority $ma2$ is assigned at t, and can be extended to $ma1$ by route r. Here, the topological information of valid extensions from the track plan are encoded using the predicate *ext* : $MA \times Route \times MA$. We control the behaviour of this predicate by adding the following axiom:

$$\bullet\ ext(ma1, r, ma2) \Rightarrow ma2 = ma1 ++ regions(r)$$ %(ext_defn)%

stating that *ext* behaves like list concatenation for the elements it is defined for. (2) The route used for the extension is open at the given time – where the predicate *__isOpenAt__* : $Unit \times Time$ is used (as we shall see later in Section 6.2.2) to encode the clear table conditions of a route; (3) If the movement authority which is being extended is non empty[1], it becomes not assigned. This is encoded in CASL as:

$$\begin{array}{l}
\forall\ ma1 : MA;\ t : Time \\
\bullet\ canExtend(ma1, t) \\
\quad \Leftrightarrow \exists\ ma2 : MA;\ r : Route \\
\qquad \bullet\ assigned(ma2, t) \wedge ext(ma2, r, ma1) \\
\qquad\quad \wedge\ r\ isOpenAt\ t \\
\qquad\quad \wedge\ (assigned(ma2, suc(t)) \Rightarrow ma2 = [])
\end{array}$$
%(extends_defn)%

In a similar manner, the predicate *canReduce* : $MA \times Time$ is true, at a given time t and for a given movement authority $ma1$, if there exists a region rg such that the region concatenated at the head of $ma1$ was assigned at t and becomes unassigned at $suc(t)$.

$$\begin{array}{l}
\forall\ ma1 : MA;\ t : Time \\
\bullet\ canReduce(ma1, t) \\
\quad \Leftrightarrow \exists\ rg : Region \\
\qquad \bullet\ assigned((rg :: ma1)\ as\ MA, t) \wedge \neg\ assigned((rg :: ma1)\ as\ MA, suc(t))
\end{array}$$
%(reduces_defn)%

[1]This check is required as we want the empty movement authority to always be assigned.

Finally, we would like to ensure that, in line with our assumptions, only one movement authority can be extended at any given time, this is captured using the following axiom:

- $\forall\ t : \mathit{Time}$
  - $\forall\ m1,\ m2 : \mathit{MA}$
    - $(\mathit{assigned}(m1,\ \mathit{suc}(t)) \Rightarrow \mathit{canExtend}(m1,\ t)) \wedge$
      $(\mathit{assigned}(m2,\ \mathit{suc}(t)) \Rightarrow \mathit{canExtend}(m2,\ t))$
      $\Rightarrow m1 = m2$ %(one_MA_changes)%

6.2.2 Modelling Control Tables and Route Availability

The above modelling of movement authorities requires the definition of what it means for a route to be open. As we have seen, this information is given by the control table for a scheme plan. With respect to collisions, the clear column of a clear table states that certain track units must not be occupied by a train for a route to be granted. To model this, we introduce the predicate *clear* : *Route* × *Unit*. We then use this predicate to encode that a particular unit occurs within the clear column for the given route. As an example, the control table given in Figure 6.1 would be encoded as:

- $\mathit{clear}(RA1,\ LA1)$
- $\mathit{clear}(RA1,\ P1)$
- $\mathit{clear}(RA1,\ LA2)$
- $\mathit{clear}(RB1,\ LA1)$
- $\mathit{clear}(RB1,\ P1)$
- $\mathit{clear}(RB1,\ LB1)$
- $\mathit{clear}(RA2,\ P2)$
- $\mathit{clear}(RA2,\ LA3)$
- $\mathit{clear}(RB2,\ P2)$
- $\mathit{clear}(RB2,\ LA3)$

Here we note, that due to the fact that *clear* may be specified loosely for a given scheme plan, we obtain models where *clear* holds for more tracks than is stated. However, this does not disturb our verification proofs, as later we check properties concerning all models, and hence if *clear* is not specified tightly enough, it will appear at this point.

In a similar manner, we can use predicates to encode the normal and reverse columns of the control table. However we note that we are interested in collision freedom only. As the normal and reverse columns of a control table ensure that derailments do not occur, we exclude these definitions. Using these predicates we can give the following definition of what it means for a route to be open:

$$\forall\ r : \mathit{Route};\ t : \mathit{Time} \bullet r\ \mathit{isOpenAt}\ t \Leftrightarrow \forall\ u : \mathit{Unit} \bullet \mathit{clear}(r,\ u) \Rightarrow u\ \mathit{isOpenAt}\ t$$

This axiom states that a route r is open at a given time t, if for all units u for which clear(r, u) holds, the unit u is open at t.

Finally, we have yet to consider what is means for a unit to be occupied. Within the operation of an interlocking a unit is in use when a train is detected on a particular track. In our modelling, we abstract on the concrete position of a train and instead say that if a region is assigned to some train, then at least one of the units within that region will report that it is occupied by a train. This is captured by the following axiom:

$$\begin{aligned}
&\bullet \forall\, t : Time;\ r : Route;\ rg : Region;\ ma : MA\\
&\quad \bullet\ assigned(ma,\ t) \wedge rg\ eps\ regions(r) \wedge rg\ eps\ ma\\
&\quad \Rightarrow \exists\, u : Unit;\ upp : UnitPathPair\\
&\qquad \bullet\ \neg\ u\ isOpenAt\ t \wedge u\ eps\ rg \wedge getUnit(upp) = u \wedge upp\ eps\ r
\end{aligned}$$

%(occupied)%

This axiom makes use of the operation *regions* : *Route* $\rightarrow$ *MA* that, for a given route gives the regions it has been split into via our modelling. It also uses the operations *eps* for list elementhood and *getUnit* for returning the unit component of a UnitPathPair.

Although this axiom is fairly loose, i.e. we do not even impose the restriction that trains should move in the correct direction, we will show in Section 6.6 later that this is enough to prove the safety property we discuss next.

6.2.3 Modelling our Safety Property

In Section 6.1.3 we discuss the safety property that we would like to verify, namely that "overlapping movement authorities are not assigned at the same time". To model such a property, we introduce the auxiliary predicate *share* : *MA* × *MA* that encodes what it means for two movement authorities to overlap:

$$\begin{aligned}
&\textbf{pred}\ \ share : MA \times MA\\
&\qquad \forall\, m1,\ m2 : MA\\
&\qquad\quad \bullet\ share(m1,\ m2) \Leftrightarrow \exists\, rg : Region \bullet rg\ eps\ m1 \wedge rg\ eps\ m2
\end{aligned}$$

%(share_defn)%

Finally, this allows us to model safety using the following formula:

$$\begin{aligned}
&\forall\, t : Time,\ m1,\ m2 : MA\\
&\qquad \bullet\ share(ma1,\ ma2)\\
&\qquad \Rightarrow ma1 = ma2 \vee \neg\ (assigned(ma1,\ t) \wedge assigned(ma2,\ t))
\end{aligned}$$

%(safety)%

It states that if two movement authorities share a region, then either they are the same, or that they are never both assigned at the same time. This concludes our modelling for movement authorities and safety, next we consider why this modelling is appropriate and argue its correctness.

6.3 Validating our Modelling Through Instantiation

With the axioms we have presented in place, we validate our modelling of movement authorities via a presentation of an example concrete model and proof using the scheme plan given in Figure 6.1. An excerpt of the encoding of the dynamical aspects of this scheme plan in CASL is given below:

free type *Route* ::= *RA1* | *RB1* | *RA2* | *RB2*
• *RA1* = *unitPathPair*(*LA1*, *path*(*ca0*, *ca1*)) :: (*unitPathPair*(*P1*, *path*(*ca1*, *ca2*))
:: (*unitPathPair*(*LA2*, *path*(*ca2*, *ca3*)) :: []))
• *RB1* = *unitPathPair*(*LA1*, *path*(*ca0*, *ca1*)) :: (*unitPathPair*(*P1*, *path*(*ca1*, *cb0*))
:: (*unitPathPair*(*b1*, *path*(*cb0*, *cb1*)) :: []))
...
free type *Region* ::= *RG1* | *RG2* | *RG3* | *RG4*
• *RG1* = *LA1* :: (*P1* :: [])
• *RG2* = *LA2* :: []
...
• *regions*(*RA1*) = *RG1* :: (*RG2* :: [])
• *regions*(*RA2*) = *RG4* :: []
...
free type *MA* ::= *MA1* | *MA2* | *MA3* | *MA4* | *MA5* | *MA6* | *MA7* |
MA8 | *MA9* | []
• *MA1* = *RG1* :: (*RG2* :: [])
• *MA2* = *RG1* :: (*RG3* :: [])
...

To validate our model, we consider the transition system given in Figure 6.4. The transition system shows the possible assignment of movement authorities for the scheme plan given in Figure 6.1 under the assumption of a single train. That is, a system where we only consider assignments of a single movement authority. The transition system shows transitions possible in a real world setting. We now use CASL to model a series of proof goals representing the states and transitions of this system. Through proving that our modelling approach captures this transition system, we know that any property that is proven over our model holds for the real world system.

To validate that our modelling allows for the assignments given in the transition system, we specify the following proof goals describing the movement authorities that are assignable over the first four time steps of the system. As all states can be reached within four steps, this ensures all states in Figure 6.4 are captured by our model.

then %implies
∀ *m* : *MA*
• $\neg$ *m* = [] $\Rightarrow$ $\neg$ *assigned*(*m*, 0) %(Time_0)%
• *assigned*(*m*, *suc*(0)) $\Rightarrow$ *m* = *MA1* $\vee$ *m* = *MA2* $\vee$ *m* = [] %(Time_1)%
• *assigned*(*m*, *suc*(*suc*(0))) $\Rightarrow$ *m* = *MA1* $\vee$ *m* = *MA2* $\vee$

$m = \mathit{MA4} \vee m = \mathit{MA5} \vee m = \mathit{MA6} \vee$
$m = \mathit{MA7} \vee m = []$ %(Time_2)%

- $\mathit{assigned}(m, \mathit{suc}(\mathit{suc}(\mathit{suc}(0)))) \Rightarrow m = \mathit{MA1} \vee m = \mathit{MA2} \vee$
  $m = \mathit{MA4} \vee m = \mathit{MA5} \vee m = \mathit{MA6} \vee$
  $m = \mathit{MA7} \vee m = \mathit{MA8} \vee m = \mathit{MA9} \vee m = []$ %(Time_3)%
- $\mathit{assigned}(m, \mathit{suc}(\mathit{suc}(\mathit{suc}(\mathit{suc}(0))))) \Rightarrow m = \mathit{MA1} \vee$
  $m = \mathit{MA2} \vee m = \mathit{MA3} \vee m = \mathit{MA4} \vee$
  $m = \mathit{MA5} \vee m = \mathit{MA6} \vee m = \mathit{MA7} \vee$
  $m = \mathit{MA8} \vee m = \mathit{MA9} \vee m = []$ %(Time_4)%

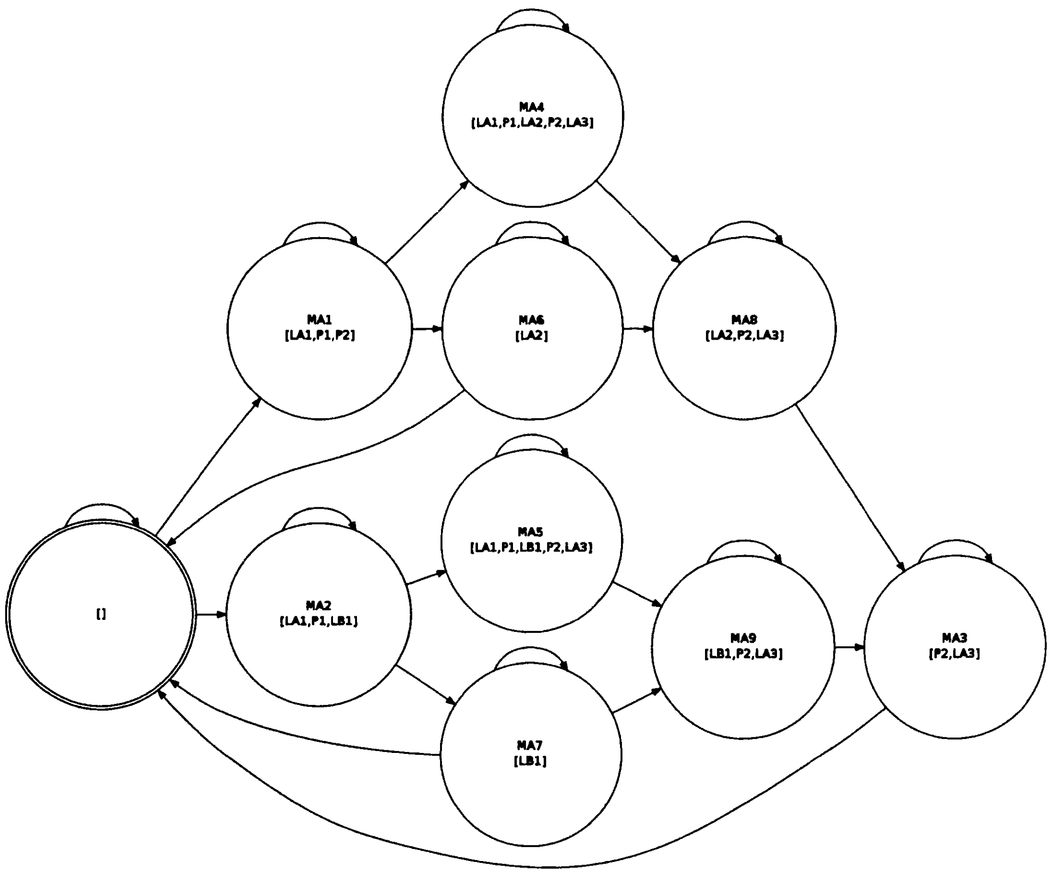


Figure 6.4: Transition system for movement authority assignment under the assumption of a single train.

We also encode the possible movements in and out of each state within the transition system given in Figure 6.4. This ensures that all the transitions that can be performed by the real world system are captured by model.

then %implies

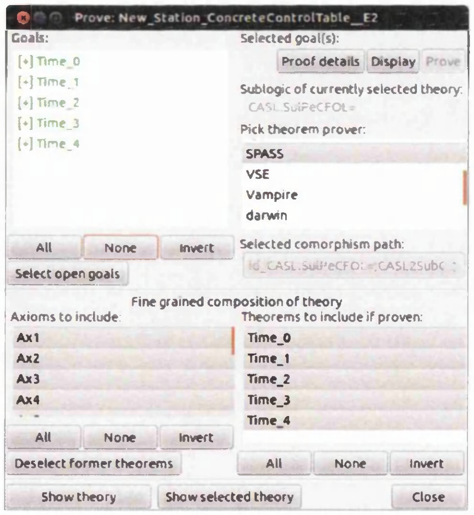
$\forall\ t : Time$

- $assigned([], t) \Rightarrow assigned([], suc(t)) \vee assigned(MA1, suc(t))$
  $\vee\ assigned(MA2, suc(t))$ %(State_Empty)%
- $assigned(MA1, t) \Rightarrow assigned(MA1, suc(t)) \vee assigned(MA4, suc(t))$
  $\vee\ assigned(MA6, suc(t))$ %(State_MA1_Out)%
- $assigned(MA1, suc(t)) \Rightarrow assigned(MA1, t)$
  $\vee\ assigned([], t)$ %(State_MA1_In)%
- $assigned(MA2, t) \Rightarrow assigned(MA2, suc(t)) \vee assigned(MA5, suc(t))$
  $\vee\ assigned(MA7, suc(t))$ %(State_MA2_Out)%
- $assigned(MA2, suc(t)) \Rightarrow assigned(MA2, t)$
  $\vee\ assigned([], t)$ %(State_MA2_In)%

...

- $assigned(MA9, suc(t))$
  $\Rightarrow assigned(MA9, t) \vee assigned(MA7, t)$
  $\vee\ assigned(MA5, t)$ %(State_MA9_In)%

For example, the axiom labelled %(State_Empty)% describes that from the state where no movement authority is assigned, we can get to two possible states, namely where movement authorities MA1 or MA2 are assigned. Using HETS, we can discharge each of these proof goals automatically. Figure 6.5 shows the HETS proof window with each goal discharged. It highlights in the top left of each window, the series of proof goals with a green "+" symbol next them. This shows that HETS has successfully discharged the proof.



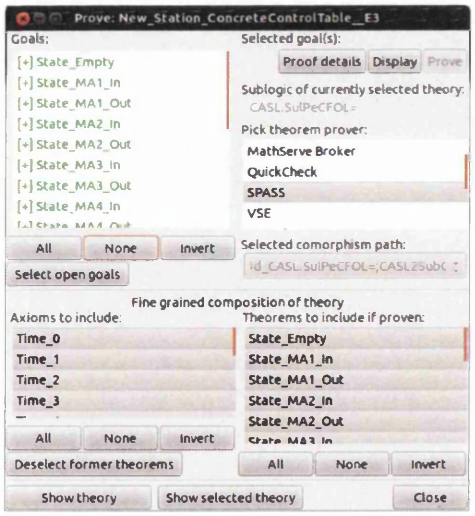


Figure 6.5: Proof windows showing discharged proof goals.

The validation we have provided illustrates that our modelling of movement authorities matches the real world expectations of movement authorities. With this level of confidence within our modelling, we move on to consider verification of real world scheme plans.

6.3.1 Unsuccessful Verification Over Bjørner's DSL

Overall, the capturing of Bjørner's DSL along with our extension of movement authorities in CASL is intuitive and relatively straightforward for railway engineers to follow. However, with respect to verification, SPASS, Vampire and eProver were all unable to directly prove our safety principle (within six hours per prover). Even with the addition of the following property specific axiom for induction over time, all the provers still fail:

$$
\begin{array}{l}
\bullet\ (\forall\ ma1,\ ma2 : MA \bullet share(ma1,\ ma2) \\
\quad \Rightarrow ma1 = ma2 \vee \neg\ (assigned(ma1,\ 0) \wedge assigned(ma2,\ 0))) \qquad \text{%% Base Case} \\
\quad \wedge \\
\quad \forall\ t : Time \bullet (\forall\ ma1,\ ma2 : MA \bullet share(ma1,\ ma2) \\
\qquad \Rightarrow ma1 = ma2 \vee \neg\ (assigned(ma1,\ t) \wedge assigned(ma2,\ t))) \\
\quad \Rightarrow (\forall\ ma1,\ ma2 : MA \bullet share(ma1,\ ma2) \\
\qquad\quad \Rightarrow ma1 = ma2 \vee \neg\ (assigned(ma1,\ suc(t)) \wedge assigned(ma2,\ suc(t)))) \\
\hfill \text{%% Step Case} \\
\Rightarrow \forall\ t : Time \bullet (\forall\ ma1,\ ma2 : MA \bullet share(ma1,\ ma2) \\
\qquad \Rightarrow ma1 = ma2 \vee \neg\ (assigned(ma1,\ t) \wedge assigned(ma2,\ t)))
\end{array}
$$

This is not surprising, as Bjørner intended to provide a strong language for modelling, and we have aimed to model movement authorities intuitively. However, the concept of a movement authority and our safety principle lend themselves, on the general level of the railway domain, to natural abstractions that we now show can be exploited for verification.

6.4 Domain Specific Property Supporting Lemmas

Within the railway domain, it is understood that control tables are vital in ensuring safety. In our presentation we can see that movement authorities are extended depending on the rules of the control table. That is, for a movement authority to be extended by the regions of a route, that route must be open. Thus, we consider simplifying our proof goal for safety to a goal that considers the control table. Through some domain analysis, we can reduce the reasoning on the level of movement authorities, to a reasoning on the level of topological routes and the control table. That is, under the condition that the predicate *ext* is encoded faithfully for the given model (i.e. that for all movement authorities $m1, m2$ and routes r we have $ext(m1, r, m2)$ only if $m2$ is the extension of $m1$ with the regions of r), then the scheme plan exhibits the following property:

for all routes, if a region of the route being assigned at a particular time implies that the route is not open at that time, then two overlapping movement authorities cannot be assigned at the same time.

This is captured formally by the following lemma:

Lemma 6.2 (Property Reduction to Routes) Given a faithfully modelled scheme plan SP then

$$\begin{aligned} &\forall\ t : \mathit{Time};\ \mathit{m1},\ \mathit{m2} : \mathit{MA} \bullet \mathit{share}(\mathit{m1},\ \mathit{m2}) \\ &\qquad \Rightarrow \mathit{m1} = \mathit{m2} \vee \neg\ (\mathit{assigned}(\mathit{m1},\ t) \wedge \mathit{assigned}(\mathit{m2},\ t)) \qquad \%(*)\% \end{aligned}$$

if and only if,

$$\begin{aligned} &\forall\ t : \mathit{Time};\ r : \mathit{Route};\ \mathit{rg} : \mathit{Region};\ \mathit{ma} : \mathit{MA} \\ &\qquad \bullet\ \mathit{assigned}(\mathit{ma},\ t) \wedge \mathit{rg}\ \mathit{eps}\ \mathit{ma} \wedge \mathit{rg}\ \mathit{eps}\ \mathit{regions}(r) \\ &\qquad\qquad \Rightarrow \neg\ r\ \mathit{isOpenAt}\ t \qquad \%(**)\% \end{aligned}$$

Proof. Following from the axioms presented in Section 6.2 we have:

$\Leftarrow$: Let us assume (**). The proof is given by induction on time t.

In the base case (t=0), (*) is true. This is the case as only the empty movement authority is assigned at time 0 (given by axiom %(no_ma_0)%). Let $m1$, $m2$ be two movement authorities such that $share(m1, m2)$ holds. We consider two cases:

(1) if $m1 = m2$ then the implication holds trivially;

(2) if $m1 \neq m2$ then only one of these movement authorities can be the empty movement authority. Let, without loss of generality, $m1 \neq []$. Then $assigned(m1, 0)$ is false (by axiom %(no_ma_0)%), hence the implication holds.

For the step case we have to show that:

$$\begin{aligned} &\forall\ t : \mathit{Time} \bullet \\ &\qquad (\forall\ \mathit{m1},\ \mathit{m2} : \mathit{MA} \bullet \mathit{share}(\mathit{m1},\ \mathit{m2}) \\ &\qquad\qquad \Rightarrow \mathit{m1} = \mathit{m2} \vee \neg\ (\mathit{assigned}(\mathit{m1},\ t) \wedge \mathit{assigned}(\mathit{m2},\ t)))\ \%(\mathrm{ih})\% \\ &\qquad \Rightarrow \\ &\qquad (\forall\ \mathit{m1},\ \mathit{m2} : \mathit{MA} \bullet \mathit{share}(\mathit{m1},\ \mathit{m2}) \\ &\qquad\qquad \Rightarrow \mathit{m1} = \mathit{m2} \vee \neg\ (\mathit{assigned}(\mathit{m1},\ \mathit{suc}(t)) \wedge \mathit{assigned}(\mathit{m2},\ \mathit{suc}(t)))) \end{aligned}$$

Let us assume (ih) for all movement authorities $m1, m2$. Let $m1'$, $m2'$ be two movement authorities such that $share(m1', m')$ holds. We consider two cases:

(1) if $m1' = m2'$ then the implication holds trivially;

(2) if $m1' \neq m2'$ then we need to show $\neg$ (*assigned*(*m1'*, *t*) $\wedge$ *assigned*(*m2'*, *t*)). By case distinction over the definition of the predicate *assigned* (axiom %(assigned_defn)%), we show that (*assigned*(*m1'*, *t*) $\wedge$ *assigned*(*m2'*, *t*)) is not possible and hence the implication holds.

- Case 1: (*assigned*(*m1'*, *suc*(*t*)) $\wedge$ *assigned*(*m2'*, *suc*(*t*))) $\Rightarrow$ *assigned*(*m1'*, *t*) $\wedge$ *assigned*(*m2'*, *t*) does not hold, as it contradicts the assumed induction hypothesis (ih).

- Case 2: $(assigned(m1', suc(t)) \wedge assigned(m2', suc(t))) \Rightarrow canExtend(m1', t) \wedge canExtend(m2', t)$ does not hold, as it contradicts with axiom %(one_ma_changes)% stating that only one movement authority extends per time step.

- Case 3: $(assigned(m1', suc(t)) \wedge assigned(m2', suc(t))) \Rightarrow assigned(m1', t) \wedge canExtend(m2', t)$. Given that $share(m1', m2')$ holds, by the definition of the predicate *share* (axiom %(share_defn)%) we know that there exists a region rg such that $rg \in m1'$ and $rg \in m2'$. Let us consider what it means for $canExtend(m2', t)$ to hold, i.e. instantiating axiom %(extend_defn)% with $m2'$ gives:

  $$\begin{array}{l} canExtend(m2', t) \Leftrightarrow \\ \qquad \exists\ ma : MA;\ r : Route \bullet \\ \qquad\qquad assigned(ma, t) \wedge ext(ma, r, m2') \wedge \\ \qquad\qquad r\ isOpenAt\ t \wedge \\ \qquad\qquad (\neg\ ma = [] \Rightarrow \neg\ assigned(ma, suc(t))) \end{array}$$

  Hence, we have two cases to consider for the shared region rg:

  (1) if $rg \in ma$, then by (**), all routes r' such that $rg \in regions(r')$ are not open. Hence given axiom %(ext_defn)%, which tells us that ext acts like list concatenation, we know that $\neg\ r\ isOpenAt\ t$. Thus we have a contradiction that $canExtend(m2', t)$ must hold, but also $\neg\ r\ isOpenAt\ t$ must hold;

  (2) if $rg \notin ma$ then by axiom %(ext_defn)% we know that $share(m1', ma)$ holds. We also know, by the definition of canExtend (axiom %(extends_defn)%) that $assigned(ma, t)$ holds and similarly by our case assumption that $assigned(m1', t)$ holds. Now if $m1' \neq ma$ then we gain a contradiction to our induction hypothesis (ih). Whereas, if $m1' = ma$, then by the definition of *canExtend* (axiom %(extends_defn)%), we know that $\neg assigned(ma, suc(t))$ and hence that $\neg assigned(m1', suc(t))$. This again contradicts our case assumption that $assigned(m1', t)$ holds.

- Case 4: $(assigned(m1', suc(t)) \wedge assigned(m2', suc(t))) \Rightarrow canExtend(m1', t) \wedge assigned(m2', t)$: Analogous to Case 3.

- Case 5-9: The remaining cases involve the reduction of one movement authority that is already assigned. Hence, each case follows from the definition of *canReduce* %(reduces_defn)% and our induction hypothesis (ih).

$\Rightarrow$: Let us assume (*). The proof is by contradiction.

We assume (**) does not hold, that is, we know there exists a route r such that:

$$\begin{array}{l} \exists\ t : Time;\ r : Route;\ rg : Region;\ ma : MA \\ \qquad \bullet\ assigned(ma, t) \wedge rg\ eps\ ma \wedge rg\ eps\ regions(r) \\ \qquad\qquad \Rightarrow r\ isOpenAt\ t \end{array}$$

Given this, this route can be used for an extension to another movement authority even though it is assigned. Assume at some time t', such that $suc(t') = t$ there exist movement authorities $ma1, ma2$ such that $ext(ma1, r, ma2)$ and $rg \in ma2$. Then the predicate $canExtend$ (axiom %(extends_defn)%) holds for $ma2$ at time t. Thus, by the definition of assigned (axiom %(assigned_defn)%) $ma2$ can become assigned at time $suc(t')$, i.e. time t. Now we have the situation that $assigned(ma, t)$ and $assigned(ma2, t)$ hold, but so does $share(ma, ma2)$ (axiom %(share_defn)%) over the region rg. Hence we have a contradiction to (**).

□

The result of this lemma is that we can effectively reduce the verification problem over movement authorities to a simpler problem over route openness.

6.4.1 Automatic Proof of DSL Lemmas

The proof we have presented above has also been encoded in CASL and automatically discharged using HETS. The specification for this encoding is given in Appendix E. To do this, the overall proof goal was split into the nine cases given above. Each of these cases were then discharged as an implied axiom of our extended DSL using SPASS. Overall, the proof of these nine cases takes a total of 42 seconds. Finally, we encoded an implication showing that these nine cases do in fact lead to Lemma 6.2. Again, this proof can be automatically discharged with SPASS and took 87 seconds. Giving a total proof time of 129 seconds.

At this point, we note that as this lemma is completely independent of any concrete scheme plan formulated using our extended DSL. Hence, once it has been proven as a consequence of the DSL, it can be used to aid with verification of any scheme plan formulated using the extended DSL. However, we note that it is the responsibility of the computer scientist extending the DSL to ensure these proofs still hold if any changes are made to the underlying DSL.

6.5 A Note on Faithful Modelling

In the previous sections, we have presented a modelling of movement authorities that extends Bjørner's original DSL. We have also mentioned that elements of the extension should be encoded faithfully with respect to the original scheme plan. For example, the operation *regions* : *Route* $\rightarrow$ *MA* should be encoded such that concatenating the resulting list of regions (or MA) for a route gives exactly the list of units occurring in a route. In this section, we illustrate that it is possible to prove that the elements in the extension of the modelling for a concrete scheme plan match the original elements of the scheme plan. This ensures that confidence is maintained in the fact that anything proven over the extended scheme plan holds for the scheme plan formulated in the original DSL. We illustrate such an approach using the scheme plan in Figure 6.1 and the operation *regions*.

Consider route $RA1$ from the single station scheme plan (Figure 6.1), with our modelling using regions, this route gets split into regions $RG1$ and $RG2$. Thus, we would encode $\mathit{regions}(RA1) = RG1 :: (RG2 :: [])$ where $RG1$ and $RG2$ have the definitions $RG1 = LA1 :: (P1 :: [])$ and $RG2 = LA2 :: []$ respectively. Now, we can add the following proof goal to our model:

then %implies
$\forall\ u : \mathit{Unit}$
$\bullet\ \exists\ upp : \mathit{UnitPathPair}$
$\bullet\ \mathit{getUnit}(upp) = u \wedge upp\ \mathit{eps}\ RA1$
$\Leftrightarrow \exists\ rg : \mathit{Region} \bullet rg\ \mathit{eps}\ \mathit{regions}(RA1) \wedge u\ \mathit{eps}\ rg$

ensuring that all the units of route $RA1$ appear in the regions of $RA1$ and vice versa. Such a proof goal is then discharged using the provers in HETS. Similar proof goals can be added over all predicates and operations involved in the extension. This ensures that the extensions to the scheme plan are encoded correctly with respect to the original scheme plan. Later in Chapter 7 we see how the models we have presented can be generated via model transformations where such checks again ensure the correctness of the implemented model transformation.

6.6 Verification of Real World Scheme Plans

In this section, we apply our extended DSL by modelling a variety of real world stations and junctions, and verifying that they meet our defined safety property. To begin, we discuss the specification structure we have in place.

6.6.1 Specification Structure

In this thesis so far, we have outlined a series of extensions to Bjørner's original DSL. Before continuing to give verification results, we briefly comment on the specification structure our approach has led to.

Considering Figure 6.6 we can see, that our specifications begin with the Datatypes (DATATYPES) and DSL (DSL) gained from our translation of the UML class diagram. We then extend these specifications with DSLEXTENSION for modelling the narrative aspects of the original informal DSL. Concretely for Bjørner's DSL this includes our modelling of movement authorities. Next, we can see that the property supporting lemmas (DSLEMMAS) are added to aid with verification. These are added as implied axioms over the extended DSL. This illustrates that these lemmas are independent of any scheme plan formulated in the DSL, however that a change in the DSL would require these lemmas to be re-proven. Next, we see the specification CONCRETESCHEMEPLAN that encodes a particular track plan and its associated movement authorities and control tables. Finally, we can see the proof goals for proving safety are added (SAFETY). These are again added as implied axioms, and are then proven relative to the given concrete

```
spec DSLForVerification =
      Datatypes                          %% From the UML class diagram.
then
      DSL                                %% From the UML class diagram.
then
      DSLExtension                       %% From the narrative.
then %implies
      DSLemmas                           %% Extension with proof support.
then
      ConcreteSchemePlan                 %% Specific scheme plan.
then %implies
      Safety                             %% Safety properties to be proven.
end
```

Figure 6.6: Specification Structure.

scheme plan. Given this verification setup, we now discuss the encoding and verification of several scheme plans.

6.6.2 Verification Results

The track plan in Figure 3.1 (TP-A) and each track plan in Figure 6.7 (TP-B to TP-D) have been modelled in Bjørner's extended DSL including our modelling of movement authorities. Following the specification structure in Figure 6.6, we have added a then %implies block for the proof of safety. This block is structured in two parts, the first contains lemmas to be proven, then the second our overall proof goal. The aim of the lemmas from the first block is similar to the case distinction lemmas in Section 6.4. That is, they help the automatic prover in proving our final goal. For verification using Bjørner's DSL, these lemmas are in the form of our final proof goal instantiated for each route. Such lemmas can be easily automatically generated by static analysis. That is, taking route RA1 as an example, we have:

$$\begin{aligned} &\forall\, t : \mathit{Time};\ rg : \mathit{Region};\ ma : \mathit{MA} \\ &\quad \bullet\ \mathit{assigned}(ma,\ t) \wedge rg\ \mathit{eps}\ ma \wedge rg\ \mathit{eps}\ \mathit{regions}(\mathit{RA1}) \\ &\qquad\qquad \Rightarrow \neg\ \mathit{RA1}\ \mathit{isOpenAt}\ t \end{aligned}$$

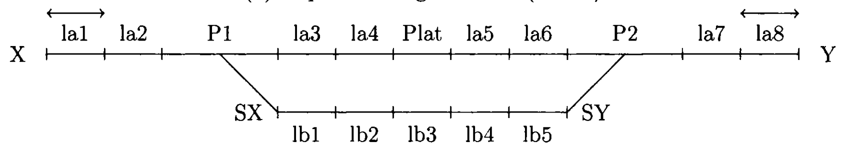
Once these lemmas have been automatically proven, they can be used to help deduce our overall proof goal:

then %**implies**

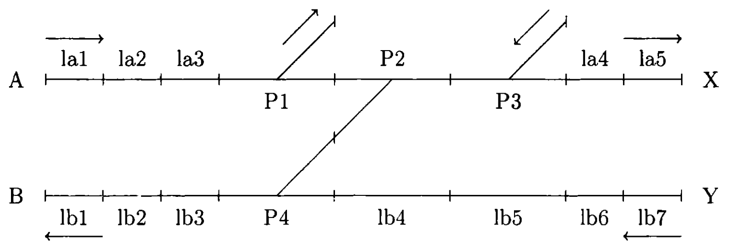
$$\begin{aligned} &\forall\, t : \mathit{Time};\ r : \mathit{Route};\ rg : \mathit{Region};\ ma : \mathit{MA} \\ &\quad \bullet\ \mathit{assigned}(ma,\ t) \wedge rg\ \mathit{eps}\ ma \wedge rg\ \mathit{eps}\ \mathit{regions}(r) \\ &\qquad\qquad \Rightarrow \neg\ r\ \mathit{isOpenAt}\ t \end{aligned}$$

end

(a) A pass through station (TP-B)



(b) A double junction (TP-C)



(c) A terminal station (TP-D)

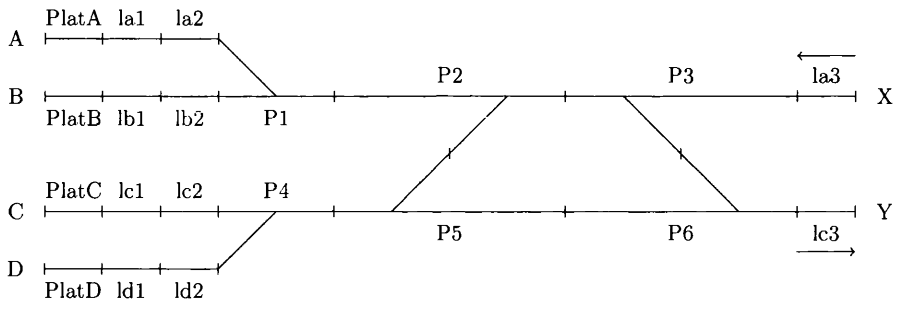


Figure 6.7: Verified track plans.

The verification times presented in Figure 6.8 show that verification is now possible over our extended version on Bjørner's DSL. To this end, we have successfully automatically verified four track plans of real world complexity (see Figure 6.7). The proofs have been performed on a 3GHz quad core machine with 8GB of RAM running Ubuntu 12.04.

Track Plan	Routes Lemma Proofs (s)	Safety Proof (s)	Avg. Memory (MB)
TP-A	236.10	20.46	489.64
TP-B	54.54	10.04	176.88
TP-C	23.88	6.57	90.36
TP-D	194.41	22.34	188.07

Figure 6.8: Verification times for the given track plans.

Overall, all proofs are completed relatively quickly, with the longest proof time being for track plan TP-A. Interestingly, this is due to the number of units contained within this track plan. It is also interesting to note that the track plans that look more complicated, i.e. TP-C and TP-D are relatively quick to verify. This is due to the fact that these track plans get split into many smaller regions compared with the fewer larger regions of track plan TP-A. Each of these small regions is represented by a list containing few elements within our modelling. Hence, the automatic theorem prover does not need to search to such a depth to find a proof. This point is also illustrated by the increased average memory usage for TP-A.

Overall, these results shows that the DSL as introduced by Bjørner and extended by us is rich enough to allow natural abstractions and establish lemmas which give a measurable effect for verification.

Chapter 7

Creating Graphical Tool Support

Contents

This chapter describes the OnTrack tool[1] 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 the model transformations required to generate CASL models. This discussion serves as an illustration on how the tool can be extended for other formalisms. OnTrack also contains the ability to output CSP||B models [JTT$^+$13], however we refrain from a discussion of these models here. Finally, the tool allows the user to apply abstractions that are formed purely over the DSL. These abstractions are independent of the formal modelling language used. We briefly discuss these abstractions and how they can be applied to aid with verification of models formulated in various different specification languages.

7.1 Motivation for Graphical Tooling

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. Up until now, we have presented several CASL specifications for varying aspects of the railway domain. We

[1]OnTrack is available for download from http://www.csp-b.org.

have also shown how to support these models with domain specific lemmas allowing for successful automatic verification. However, although these models use terminology from the railway domain, the modelling approaches presented are not in a form that is common knowledge for the everyday engineer working within the railway domain. In this chapter, we introduce the OnTrack toolset that, following the aims we have set out for this thesis, achieves the goal of encapsulating formal methods for the railway domain. Overall, the OnTrack toolset is 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 meets the third aim of this thesis, namely to make formal methods accessible to domain engineers.

In OnTrack, we also emphasise the use of a DSL and decoupling this DSL from the verification method. One of the novelties of this is that we 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. Thus meaning 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 more widely applicable.

7.2 The OnTrack Toolset 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 [Gro09] and multiple associated Epsilon [KRPP13] model transformations.

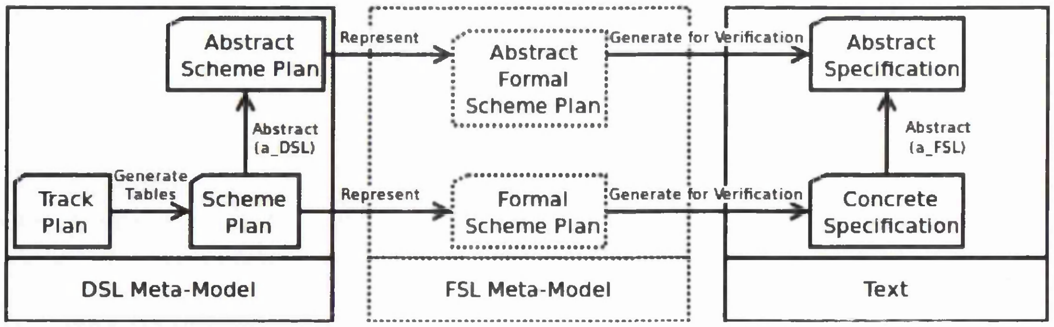


Figure 7.1: The OnTrack workflow.

Figure 7.1 shows the architecture that we employ in OnTrack. Firstly, considering the bottom horizontal workflow, a user initially draws a *Track Plan* using the graphical front end. Then the first transformation, *Generate Tables* leads to a *Scheme Plan*, which is a track plan and its associated control tables. Generation of control tables has been previously studied [MY09] and here we implement a technique that produces control tables based on track topology and signal positions. That is, we form all possible routes between pairs of signals. Control tables, as we have seen, contain information about when routes can be granted, see [KR01] for details. Track plans and scheme plans are models formulated relative to Bjørner's DSL meta model, see Figure 2.3. A scheme plan is then the basis for subsequent workflows that support its verification. Scheme plans can then be translated to formal specifications. This can be achieved in two possible ways, indicated by the optional dashed box in Figure 7.1:

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 *Generate for Verification* 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 enrich the models appropriately for verification, for example by including the DSL lemmas discussed in Section 6.4. Here, the advantage of this approach is that the *Generate for Verification* transformations can be defined generally for the formal specification language meta model.

2. Direct generation of a formal specification.

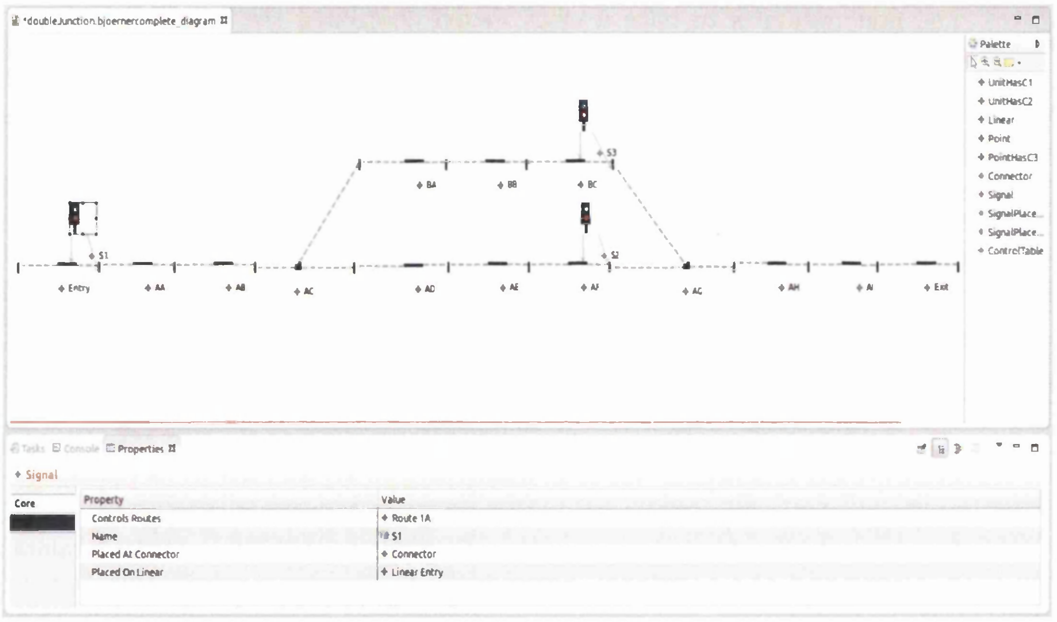
   The second approach is to directly generate a formal specification. Thus only the *Generate for Verification* model to text transformations need to be implemented. Once again, these transformations turn a *Scheme Plan* into a *Formal Specification Text* ready for verification. Again, these *Generate for Verification* transformations can still enrich the models appropriately for verification.

The horizontal workflow, described above, provides a validated transformation (via manual review of the transformations) yielding a formal specification that faithfully represents a scheme plan. Within OnTrack, the first approach has been taken for the generation of CSP||B [JTT$^+$13]. Whilst here, we highlight the second approach that we have taken for the generation of CASL.

The top level of the workflow shows the ability of OnTrack to include abstractions. We are interested in abstractions to ease verification. In [JBR13] we identified domain specific abstractions over scheme plans and similarly Moller et al. [MNR$^+$13] have identified abstractions representing topological insights from the domain. Interestingly, these abstractions are all formulated over railway scheme plans, and as such are independent of the formal specification language being used. In OnTrack we provide the ability to define such topological abstractions with respect to the DSL, thus they are decoupled from the formal specification language. We discuss an example of such an abstraction later in Section 7.5.

7.3 OnTrack Implementation for Bjørner's DSL

OnTrack implements the workflow from Section 7.2 in a typical EMF/GMF/Epsilon architecture [Gro09, SBMP08, KRPP13]. That is, as discussed in Section 2.6.2 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 2.3 of Bjørner's DSL can easily (with some slight modifications e.g. see Section 7.6) be captured within an ECORE meta model that underlies our toolset. Practically, this ECORE meta model is a UML class diagram represented using XML.



Signal

Core

Property	Value
Controls Routes	Route 1A
Name	S1
Placed At Connector	Connector
Placed On Linear	Linear Entry

Figure 7.2: A screenshot of "OnTrack" modelling a station.

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 7.2 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 Bjørner 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 CASL specifications.

7.3.1 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 Bjørner's DSL. Figure 7.3 shows several of the figures we have used for given classes.

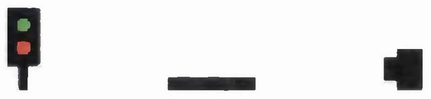
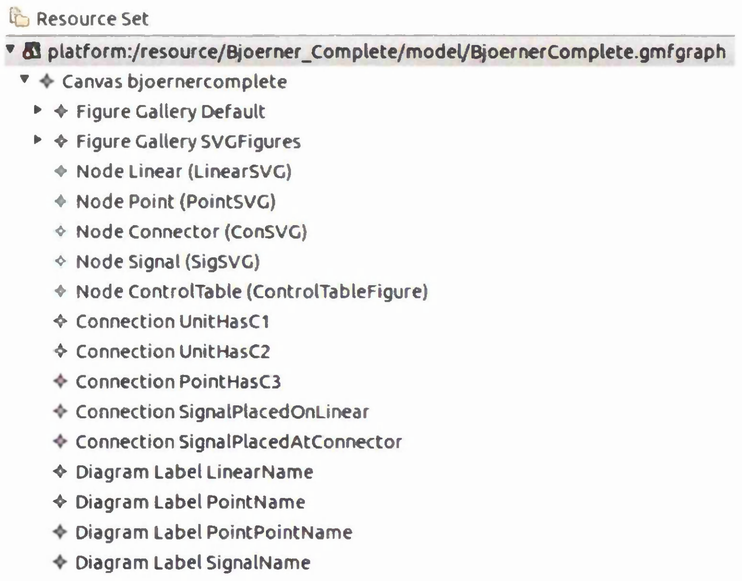


Figure 7.3: The OnTrack SVG Figures for signals, linear units and points.

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 7.4.

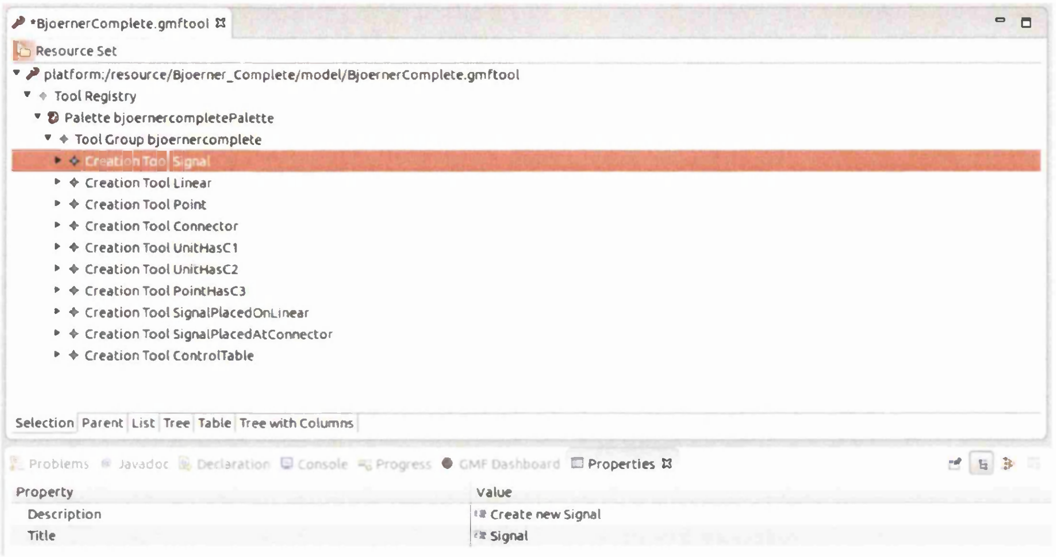


Resource Set
- platform:/resource/Bjoerner_Complete/model/BjoernerComplete.gmfgraph
  - Canvas bjoernercomplete
    - Figure Gallery Default
    - Figure Gallery SVGFigures
    - Node Linear (LinearSVG)
    - Node Point (PointSVG)
    - Node Connector (ConSVG)
    - Node Signal (SigSVG)
    - Node ControlTable (ControlTableFigure)
    - Connection UnitHasC1
    - Connection UnitHasC2
    - Connection PointHasC3
    - Connection SignalPlacedOnLinear
    - Connection SignalPlacedAtConnector
    - Diagram Label LinearName
    - Diagram Label PointName
    - Diagram Label PointPointName
    - Diagram Label SignalName

Figure 7.4: The graphical model definition for OnTrack.

From the graphical model definition in Figure 7.4, 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 Bjørner'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 Bjørner'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 Bjørner'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.



```
*BjoernerComplete.gmftool
Resource Set
▾ platform:/resource/Bjoerner_Complete/model/BjoernerComplete.gmftool
  ▾ Tool Registry
    ▾ Palette bjoernercompletePalette
      ▾ Tool Group bjoernercomplete
        ▸ Creation Tool Signal
        ▸ Creation Tool Linear
        ▸ Creation Tool Point
        ▸ Creation Tool Connector
        ▸ Creation Tool UnitHasC1
        ▸ Creation Tool UnitHasC2
        ▸ Creation Tool PointHasC3
        ▸ Creation Tool SignalPlacedOnLinear
        ▸ Creation Tool SignalPlacedAtConnector
        ▸ Creation Tool ControlTable
Selection | Parent | List | Tree | Table | Tree with Columns
```

Property	Value
Description	Create new Signal
Title	Signal

Figure 7.5: The tooling model definition for OnTrack.

7.3.2 OnTrack Tooling Model

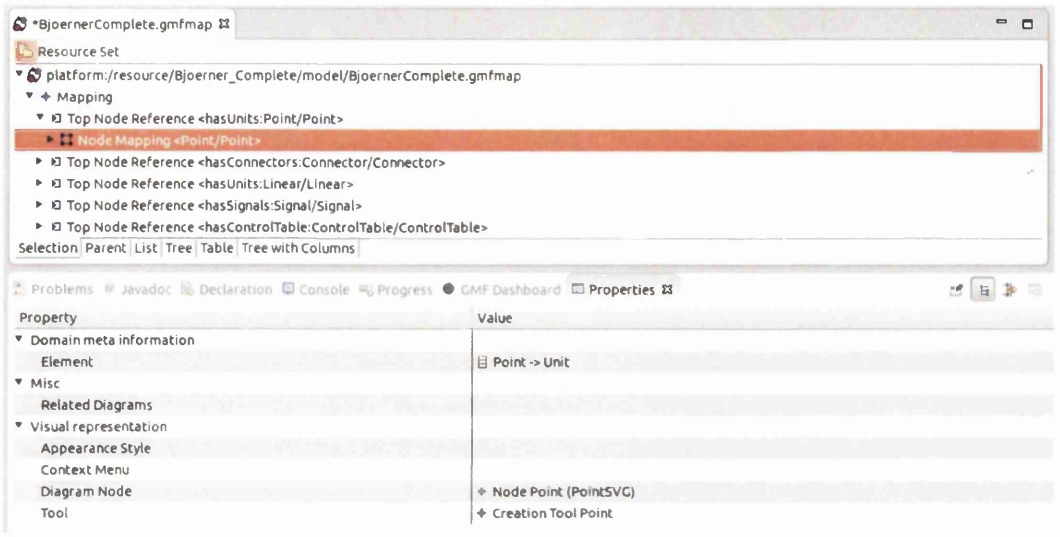
The tooling model defines the elements that can be selected from the palette of the GMF editor. The tooling model for OnTrack is shown in Figure 7.5. The elements appearing in this tooling model can be seen in the palette on the right hand side of the editor in Figure 7.2.

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 7.5) 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.

7.3.3 OnTrack Mapping Model

Finally, the mapping model for OnTrack, given in Figure 7.6, links the graphical model elements to their creation tools defined by the tooling model.

In particular, Figure 7.6 shows the specific mapping details for the *Point* element of Bjørner's DSL. It shows that the underlying model *Element* for a *Point* (which is a



```
▾ platform:/resource/Bjoerner_Complete/model/BjoernerComplete.gmfmap
  ▾ Mapping
    ▾ Top Node Reference <hasUnits:Point/Point>
      ▸ Node Mapping <Point/Point>
    ▸ Top Node Reference <hasConnectors:Connector/Connector>
    ▸ Top Node Reference <hasUnits:Linear/Linear>
    ▸ Top Node Reference <hasSignals:Signal/Signal>
    ▸ Top Node Reference <hasControlTable:ControlTable/ControlTable>
```

Property	Value
▾ Domain meta information	
Element	Point -> Unit
▾ Misc	
Related Diagrams	
▾ Visual representation	
Appearance Style	
Context Menu	
Diagram Node	Node Point (PointSVG)
Tool	Creation Tool Point

Figure 7.6: The mapping model definition for OnTrack.

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 7.2 was generated from this model. The generated OnTrack editor has been extended with a series of model transformations defined using the Epsilon framework [KRPP13]. These model transformations implement the transformations that are shown in the OnTrack workflow in Figure 7.1. The first transformation *Generate Tables*, automatically computes a control table for a track plan and is implemented as a model to model transformation using the Epsilon Wizard Language (EWL). We omit details of this transformation and focus instead on the more interesting transformations for the *Generate for Verification* and *Abstraction* transformations.

7.4 Generation of Formal CASL Specifications

Here we describe the second approach of generating formal specification, namely the direct implementation of the *Generate for Verification* transformations. We illustrate this approach for the concrete case of CASL specifications. For a discussion of the *Represent* transformations, we refer the reader to our work on OnTrack's transformations allowing output of CSP||B specifications [JTT$^+$13].

The *Generate for Verification* transformation translates meta model instances of Bjørner's DSL into formal specification text. This transformation is implemented using the Epsilon Generation Language (EGL) [KRPP13] for generating text. We have designed the generation such that it mirrors the specification structure in Section 6.6. 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. Whilst *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, Bjørner's DSL specification and similarly our extension of this with 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 7.7.

```
spec Pair [sort S] [sort T] =
    sort Pair[S,T]
    ops first: Pair[S,T] -> S;
        second: Pair[S,T] -> T;
        pair: S * T -> Pair[S,T]
...

spec ExactBjoernerStaticSignature =
...
     sorts Net, Station, Unit, Connector ...
     sorts Linear, Switch < Unit ...

     preds  __hasLine__: Net * Line;
            __hasStation__: Net * Station;
...
```

Figure 7.7: Example of static text generation for datatypes and Bjørner's DSL.

Later in the same EGL template, we can then specify the output of the concrete track plan. 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 7.8 for the dynamic generation of the free type Unit. The result of applying this EGL fragment, for example, to the concrete track plan in Figure 3.1 is the following CASL specification fragment:

```
free type Unit ::= la1 | la2 | la3 | la4 | P1 | ... | lb12 | lb13.
```

Considering Figure 7.8, 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,

```
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 7.8: Dynamic text generation for the concrete elements of the free type Unit.

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 free type.

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 7.8. 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. The result is a full CASL specification ready for verification of the current editor model instance.

7.5 General Domain Abstractions Over DSLs

The OnTrack workflow outlined in Section 7.2 allows for abstraction mechanisms for verification to be defined over the DSL. As these abstractions can be formulated purely over the DSL, it means that they can be applied independently to the chosen formal language being generated. Although, we note that it is vital that the abstractions are shown to be valid for the formal approach being used. That is, considering Figure 7.1, any abstraction a_{DSL} formulated over the DSL induces a corresponding abstraction a_{FSL} over the formal models being produced. This a_{FSL} should then be shown correct for the formal approach being applied. To illustrate this point, we have implemented a particular a_{DSL} abstraction (See Figure 7.9) based on the *simplifying scheme plan* abstraction by Moller et al. [MNR+13].

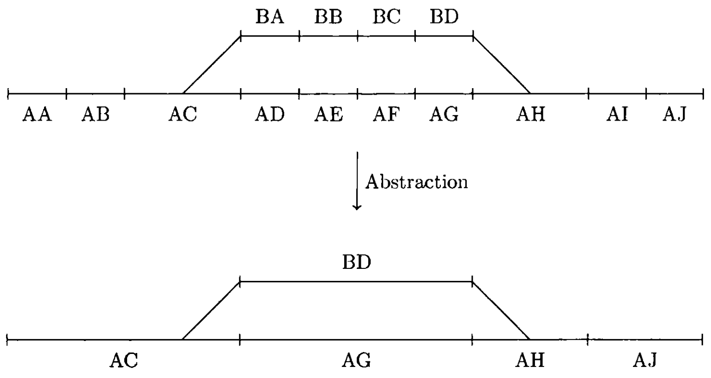


Figure 7.9: An example abstraction by Moller et al. implemented in OnTrack.

The abstraction technique defined by Moller et al. [MNR+13] involves "collapsing" various sequences of units into single units. For example, considering the track plans in Figure 7.9, as presented by Moller et al. [MNR+13]. We can see that track sections AA, AB and AC are collapsed into the single track section AC. This abstraction has

```
1.  rule abs
2.    transform rd: Input!RailDiagram to
3.    rd2 : Target!RailDiagram {
4.        rd.computeAbstractions();
5.
6.        for(ut:Unit in rd.hasUnits){
7.            if(not (toDelete.contains(ut))){
8.                if(consToBeMapped.contains(ut.hasC1))
9.                {
10.            ut.hasC1 = ut.hasC1.getMapping();
11.               }
12.               if(consToBeMapped.contains(ut.hasC2))
13.               {
14.                   ut.hasC2 = ut.hasC2.getMapping();
15.               }
16.               rd2.hasUnits.add(ut);
17.           }
18.       }
19.
20.       //Omitted code: similar computation with
21.       connectors and signals//
22.
23.       rd2.computeTables();
24.   }
```

Figure 7.10: ETL rule for abstract model transformation.

been shown correct, and to improve the feasibility of verification thanks to a reduction in the number of elements to be considered [MNR+13].

The given abstraction has been implemented at the DSL level within OnTrack. It is implemented using the Epsilon Transformation Language (ETL) [KRPP13] that is designed for model to model transformations. Figure 7.10 gives an excerpt of a main part of the transformation. The presented algorithm uses the following list structures:

- toDelete: storing units to be removed from the track plan, and
- consToBeMapped: storing which connectors from the track plan require renaming, ensuring the new track plan is properly connected.

The abs rule performs as follows: lines 2 and 3 state that the rule translates the given rail diagram `rd` to another `rd2`. The fourth line calls the operation `computeAbstraction()` on `rd` to compute which units can be collapsed and to populate the lists with appropriate values. For example, considering Figure 7.9, `toDelete` $= [AA, AB, BA, BB, BC, AD, AE, AF, AI]$. Next, the algorithm will consider every unit `ut` within the rail diagram `rd` (line 6). If `ut` is not in the list `toDelete` (line 7), then the algorithm will perform analysis on the connectors of `ut`. If connector one of `ut` is within the set of connectors requiring renaming (line 8), then the first connector of `ut` is renamed using a call to the (omitted) operation `getMapping()` (line 10). Lines 12 to 15 of the algorithm perform these steps for connector two of `ut`. After this computation, the modified unit `ut` is added as an element to the new rail diagram `rd2` (line 16). The algorithm continues in a similar manner, computing which connectors and signals should be added to `rd2`. Finally, an operation `computeTables` is called to compute a new control table for the new rail diagram `rd2`. The result of this translation is that units `AA`, `AB`, `BA`, `BB`, `BC`, `AD`, `AE`, `AF`,and `AI` are removed from the track plan to produce the abstract track plan in Figure 7.9.

Overall, implementing abstraction in the presented manner results in an automatic generation process for computing abstractions that is formulated over scheme plans. Thus, it is independent of the specification language under consideration.

7.6 A Summary of OnTrack

The OnTrack toolset achieves the aim of automating the production of formal specifications from a graphical model. Thus providing a toolset that is usable by engineers from the railway domain to produce formal CASL specifications ready for verification. This meets the third objective of our methodology as it makes formal specification available to domain engineers in a format that is accessible to them. In this chapter, we have illustrated how OnTrack can be extended to output formal specifications in further languages through the implementation of a *Generate for Verification* model transformation. We have also shown that the toolset allows for abstractions to be defined over the DSL in order to produce optimised railway models, from which transformations to formal specifications can be defined. Importantly, these abstractions are decoupled from the formal specifications. We note that work is currently underway to show that the implemented abstraction technique discussed in Section 7.5 is also valid for the CASL modelling approach discussed throughout this thesis. In building the tool, we have shown that the encoding of the DSL into a meta model is relatively straightforward, however we do note that there needs to be a close relationship between the graphical artefacts and the meta model. The overall result from this chapter is a graphical tooling environment that incorporates a "push button" verification process for critical systems within the railway domain.

7.6.1 Evaluation of the Implementation Process

To conclude this chapter, we give a reflection on the OnTrack implementation process.

In building the tool, the encoding of the DSL into an ECORE meta model is straightforward. However, we point out that the original DSL from industry may need extensions to allow certain graphical elements to be described. That is, there needs to be a close relationship between the graphical artefacts of the tool and the DSL meta model. For example, the original UML class diagram for Bjørner's DSL (Figure 2.3) gives a "has" relationship between *Unit* and *Connector* with the constraint that each *Unit* "has" at least 2 *Connectors*. Considering the graphical representation of this relation, the constraint imposes that there must be two elements in the relation. Hence, there must be two graphical lines connecting to a unit to represent the two elements of the relation. But to gain these two graphical elements requires the underlying meta model to have two separate relations between *Unit* and *Connector*. These relations are illustrated in the graphical model definition in Figure 7.4 where we can see *UnitHasC1* and also *UnitHasC2*. We note that these changes are only required for relations that need a graphical representation.

Finally, we note that once a graphical editor has been created, the implementation of model transformations for text generation is relatively straightforward. This is thanks to the language constructs that are provided by the Epsilon framework for exploring the given model. Overall, the EMF/GMF/Epsilon combination provides a powerful and easy to use tooling framework that supports the design of graphical editors with model transformation capabilities. In our experience, there are no severe limitations of these frameworks and thus we would suggest that they would be useful in a universal setting.

Chapter 8

Invensys Rail Data Model

Contents

In this chapter, we consider the application of our methodology to an industrial example in the form of the Invensys Rail Data Model (IRDM) [Rail0]. We begin by introducing the main aspects of the model. We then apply the steps of our methodology to allow for automatic verification of models formulated within the data model. We illustrate that: (1) the IRDM can be captured within the class diagram institution we have presented; (2) our pointed powerset construction and comorphism to MODALCASL can then be applied and the resulting specification translated to CASL; (3) this specification can then be extended to include a modelling of dynamical aspects from the railway domain; (4) the extended DSL can be supported with domain specific lemmas to allow for verification; and, (5) due to these lemmas, we show that verification for models formulated using this data model is possible by verifying several scheme plans. Overall, this chapter illustrates that the techniques we have presented within the thesis are applicable, and also scale to industrial sized models.

8.1 Concepts from the IRDM

We begin by outlining the main features of the Invensys Rail Data Model (IRDM). The model is currently still under development, therefore we comment on the most recent version provided to us [Rail0]. We also note, that the example modelling and verification we provide is based on this version, and as more details are finalised, the models may have to be adapted accordingly.

Overall, the model is constructed using several layers. The layers specify different levels and aspects of the railway domain. The six main layers of the model are aimed at capturing:

- Topology – such as the basic connectivity of track segments.
- Dimensions – providing dimensions for the positioning of track segments and various structures.
- Geography – for physical attributes of track segments, including gradients and curvatures etc.
- Civil Structures – covering details of buildings, bridges, stations etc.
- Track Equipment – for elements placed around the track, such as signals, points, wires etc.
- Signalling – capturing signalling constructs and principles including routes and speed restrictions.

Each layer is specified individually, although layers can extend classes specified in other layers. For example a simple track segment from the topology layer may be extended to contain details of its length in the dimensioned layer. The model is constructed using a combination of UML class diagrams and natural language descriptions. Figure 8.1 introduces an excerpt of the IRDM class diagram containing the main concepts that we are interested in for modelling track plans and verifying safety. We note that this diagram has already been labelled with the stereotypes required for our construction.

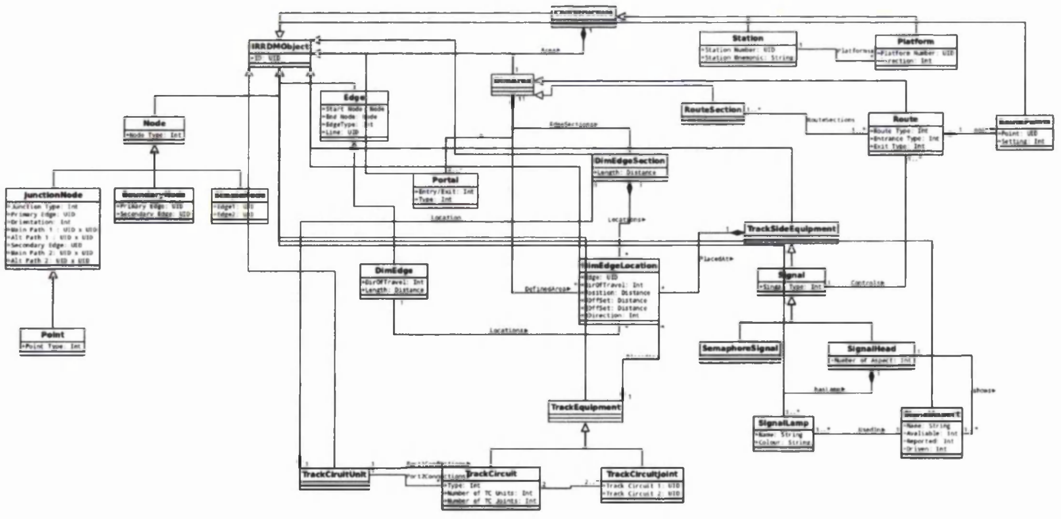


Figure 8.1: A UML Class Diagram for part of the Invensys Rail Data Model.

At the top of the Invensys Rail Data Model is a class called IRRDMObject. This class has one simply property, which is a unique identifier. All classes inherit from this class and hence inherit the unique identifier property. Overall, the railway domain is modelled as a graph like structure. A typical railway is represented as a collection of Nodes and Edges. Nodes have various types and represent areas of interest on the railway. For example, a BoundaryNode is used to represent the end of a region of railway, and a JunctionNode is used to represent a junction on the railway. Along each edge, other areas of interest can be defined, for example, CivilStructures such as Stations and Platforms. For an edge, dimensions can be defined and associated to certain elements. The class DimEdge, which is a specialised type of edge, is used as a basic measuring reference point for positioning elements along an edge. For example, consider Figure 8.2. A DimEdge can be associated to a series of DimEdgeLocations. DimEdgeLocations allow the modelling of where elements such as TrackSideEquipment are placed on the railway. Dimensioned edges can be segmented into DimEdgeSections which allow the position of train detection equipment such as TrackCircuits to be described. In a similar manner, a DimArea is used to define interesting areas within the railway. A good example of such an area is a route along which a train can travel.

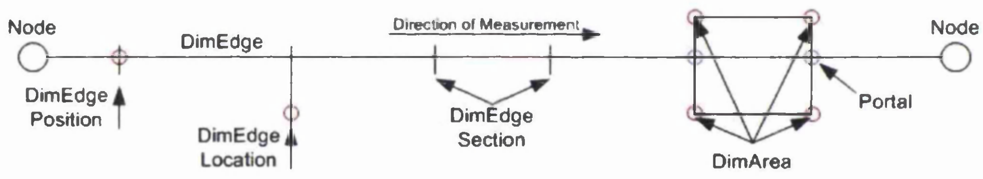


Figure 8.2: An excerpt from the Invensys Rail Data Model showing dimensional aspects of the model.

The remainder of the concepts within the UML class diagram represent common railway equipment such as: Signals that can indicate certain SignalAspects to drivers of trains; Routes that are predefined paths through the railway; Points that allow the tracks of the railway to "split"; and finally, TrackCircuits and TrackCircuitUnits, which can be used to detect where trains are along a given edge. We note that types such as *UID* and *Int* are considered built-in from this point onward.

8.1.1 Modelling a Station Using IRDM

In a similar manner to Bjørner's DSL, the Invensys Rail data model can be used to describe the layout of track plans. For example, consider the track plan in Figure 8.3. The figure shows a pass through station, similar to the one presented in Chapter 6. Firstly, one can see the graph like structure of the model, with nodes being represented by small circles and edges connecting lines between nodes. For example, at either end of the track plan we can see boundary nodes *BN1* and *BN2*, whilst we can also see two junction nodes where the track splits. Connecting these nodes, we can see a

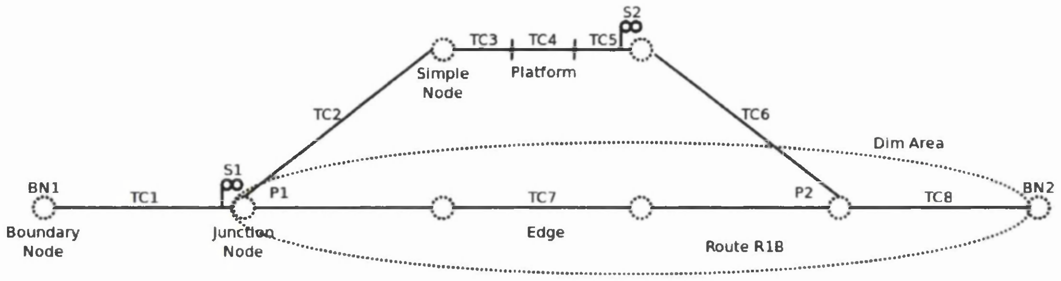


Figure 8.3: A track plan (TP-B) modelled using the IRDM.

series of simple edges. We note that each of these edges can be specialised further into dimensioned edges, as required for any elements being placed along the edge. In the center of the track plan we can see a platform, which would consist of information including the dim edge section describing the dimensions of the section of the edge it occupies. Within the track plan we can also see particular items of track equipment and track side equipment. For example within each of the junction nodes there are points *P1* and *P2* and also we can see some signals *S1* and *S2*. Each of these signals have two aspects (or coloured lamps) that it can show, namely *Proceed*, represented by a green lamp, and *Stop* represented by a red lamp[1]. Finally, we can see a dim area marked on the diagram using dotted lines running between signal *S1* and boundary node *BN2*. This area is associated to a particular route and consists of a series of track circuits. For example, we can see the marked dim area corresponds to route *R1B* and includes track circuits, *TC2*, *TC7* and *TC8*. Such track circuits are types of track equipment used to detect the presence of a train and report back to the interlocking system. Hence, there is only one track circuit covering both branches of each of the points.

8.2 Capturing IRDM within our Institution

The first step of our methodology is to formalise the Invensys Rail UML class diagrams in Figure 8.1 using our class diagram institution. Then, using the comorphism defined in Section 5.4 we can translate this class diagram into a ModalCasl specification. Here we give some examples showing how our UML class diagram institution captures the IRDM class diagram, before discussing the ModalCasl specification obtained from applying our comorphism.

8.2.1 The IRDM Class Net

Firstly, we give the class net for the IRDM UML class diagram given in Figure 8.1. For the class net, we have $CN = ((Classes, Generalisations), Instances, Properties, Compositions, Associations)$ where:

[1]Here we note that often signalling systems contain more aspects, typically four, but for the purpose of illustration of techniques we only consider two aspect signalling.

Classes	:=	$\{\mathsf{IRRDMObject}, \mathsf{Node}, \mathsf{Edge}, \mathsf{JunctionNode}, \mathsf{BoundaryNode}, \mathsf{SimpleNode}, \mathsf{Point}, \mathsf{CivilStructure}, \mathsf{Station}, \mathsf{Platform}, \mathsf{DimArea}, \ldots, \mathsf{SignalHead}, \mathsf{SignalLamp}, \mathsf{SignalAspect}\}$.
Generalisations	:=	Node $\leq$ IRRDMObject, Edge $\leq$ IRRDMObject, …, JunctionNode $\leq$ Node, BoundaryNode $\leq$ Node, SimpleNode $\leq$ Node, Point $\leq$ JunctionNode, …, Station $\leq$ CivilStructure, Platform $\leq$ CivilStructure.
Properties	:=	{IRRDMObject.ID : UID, Node.NodeType : Int, JunctionNode.JunctionType : Int, …, JunctionNode.AltPath2 : UID $\times$ UID, BoundaryNode.PrimaryEdge: UID, …, SignalAspect.Reported : Int, SignalAspect.Driven : Int}.
Compositions	:=	{CivilStructure $\blacklozenge$Area : DimArea, DimArea $\blacklozenge$definedArea : DimEdgeLocation, …, Route $\blacklozenge$point : RoutePoints, SignalHead $\blacklozenge$hasLamp : SignalLamp}.
Associations	:=	{locations{dimedge : DimEdge, dimedgelocation : DimEdgeLocation}, has{ trackcircuit : TrackCircuit, trackcircuitjoint : TrackCircuitJoint}, …, shows{ signalhead : SignalHead, signalaspect : SignalAspect}}.

We note that where role names for compositions and associations are not defined within the UML class diagram, we simply use the name of the class in lower case. It is also a simple process to check that the above forms a valid class net according to the definition in Section 5.5.

8.2.2 Multiplicity Constraints within IRDM

Next, for an example of the sentences from the IRDM class diagram, we formalise some of the composition and association multiplicities from Figure 8.1:

$$\mathsf{CivilStucture}\blacklozenge\mathsf{Area} : \mathsf{DimArea}\,!$$
$$\mathsf{DimArea}\blacklozenge\mathsf{DefinedArea} : \mathsf{DimEdgeLocation}\,!$$
$$\mathsf{TrackEquipment}\blacklozenge\mathsf{PlacedAt} : \mathsf{DimEdgeLocation}\,!$$
$$\ldots$$
$$1 \leq \#\mathsf{shows}\{\mathsf{signalhead} : \mathsf{SignalHead}, \mathsf{signalAspect} : \mathsf{SignalAspect}\}[\mathsf{shows}]$$
$$1 \leq \#\mathsf{locations}\{\mathsf{dimedge} : \mathsf{DimEdge}, \mathsf{dimedgelocation} : \mathsf{DimEdgeLocation}\}[\mathsf{locations}]$$
$$\ldots$$
$$\#\mathsf{portconnections}\{\mathsf{trackcircuit} : \mathsf{TrackCircuit}, \\ \mathsf{trackcircuitunit} : \mathsf{TrackCircuitUnit}\}[\mathsf{portconnections}] = 1$$

We note that we have once again used formulae with = instead of two formulae with $\leq$ where the left hand side and the right hand side are switched.

8.2.3 An Example Instance Net

In order to validate that our above presentation, namely the IRDM class net and corresponding multiplicity constraints, allow us to capture a scheme plan, we give an example instance net. We consider the following (partial) formalisation of the track plan in Figure 8.3 into an instance net:

$$
\begin{array}{lcl}
\ldots & \ldots & \ldots \\
C^{\mathcal{I}}(\mathsf{Node}) & = & \{\mathtt{BN1}, \mathtt{BN2}, \mathtt{JN1}, \mathtt{JN2}, \mathtt{SN1}, \mathtt{SN2}\} \\
C^{\mathcal{I}}(\mathsf{BoundaryNode}) & = & \{\mathtt{BN1}, \mathtt{BN2}\} \\
C^{\mathcal{I}}(\mathsf{JunctionNode}) & = & \{\mathtt{JN1}, \mathtt{JN2}\} \\
C^{\mathcal{I}}(\mathsf{Platform}) & = & \{\mathtt{Platform1}\} \\
\ldots & \ldots & \ldots \\
 & & \\
P^{\mathcal{I}}(\mathsf{IRRDMObject.ID : UID}) & : & C^{\mathcal{I}}(\mathsf{IRRDMObject}) \rightarrow ?\ C^{\mathcal{I}}(\mathsf{UID}) \\
 & & \text{where} \\
 & & \mathtt{BN1} \mapsto \mathtt{"BN1"} \\
 & & \mathtt{BN2} \mapsto \mathtt{"BN2"} \quad \text{etc.} \\
P^{\mathcal{I}}(\mathsf{Edge.startNode}) & : & \mathsf{Node}) : C^{\mathcal{I}}(\mathsf{Edge}) \rightarrow ?\ C^{\mathcal{I}}(\mathsf{Node}) \\
 & & \text{where} \\
 & & \mathsf{E1} \mapsto \mathsf{BN1} \quad \text{etc.} \\
\ldots & \ldots & \ldots \\
 & & \\
M^{\mathcal{I}}(\mathsf{CivilStructure{\bullet}Area : DimArea}) & : & C^{\mathcal{I}}(\mathsf{CivilStructure}) \rightarrow \wp(C^{\mathcal{I}}(\mathsf{DimArea})) \\
 & & \text{where} \\
 & & \mathsf{Platform1} \mapsto \mathtt{Area5} \quad \text{etc.} \\
\ldots & \ldots & \ldots \\
 & & \\
A^{\mathcal{I}}(\mathsf{Controls}(\ldots)) & = & \{(\mathtt{S1}, \mathtt{R1}), (\mathtt{S2}, \mathtt{R2}), \ldots\} \quad \text{etc.} \\
\ldots & \ldots & \ldots
\end{array}
$$

One can easily check that such an instance net conforms to the multiplicity constraints given by the class diagram. For example, it is clear that all *Platforms* are required to have at least one corresponding *DimArea*. This is due to the fact that *Platform* is a subclass of *CivilStructure* and we have the constraint $\mathsf{CivilStucture{\bullet}Area : DimArea}\,!$. From our above instance net, we can also see that *Platform1* is associated to *Area5*, and thus for this platform, the constraint is met.

This concludes the application of the first step of our methodology, namely the formalisation of the IRDM class diagram within the institution given in Chapter 5.

```
spec IRDM =

        %% Classes
        sorts IRRDMObject, Node, Edge, JunctionNode,
              BoundaryNode, SimpleNode, Point, UID,
              CivilStructure, DimEdge, DimArea, DimEdgeLocation,
              TrackCircuit, TrackCircuitUnit, SignalHead, SignalAspect ...
        %% Hierarchy
        sorts Node, Edge, JunctionNode, BoundaryNode,
              SimpleNode, Point < IRRDMObject
              ...
        %% Is Alive Preds
        rigid preds
              isAlive : IRRDMObject;
              isAlive : Node;
              isAlive : Edge;
              ...
        %% Static Associations and Compositions
        rigid preds
              __area__ : CivilStructure × DimArea;
              __definedarea__ : DimArea × DimEdgeLocation;
              __locations__ : DimEdge × DimEdgeLocation;
              __portconnections__ : TrackCircuit × TrackCircuitUnit
              ...
        %% Dynamic Associations and Compositions
        flexible pred
              __shows__ : SignalHead × SignalAspect
        %% Properties
        rigid ops
              id : IRRDMObject →? UID;
              startnode : Edge →? Node;
              endnode : Edge →? Node;
              primaryedge : Node →? Edge;
              secondaryedge : Node →? Edge
              ...
        %% Axioms
        ∀ c : CivilStructure • ∃ d : DimArea • c area d
          ...
end
```

Figure 8.4: Translation of IRDM class diagram to MODALCASL.

8.3 Translation to ModalCASL

Using our formalisation of IRDM within our class diagram institution, we can now apply our general pointed power set construction given in Section 5.3. This allows us to gain a model which includes the capture of the stereotypes from the IRDM. Here, the only element marked as dynamic is the "shows" association between Signals and SignalHeads. By manually applying the comorphism outlined in Section 5.4 we gain the ModalCasl specification given in Figure 8.4.

From this translation, we can see that classes (all of which were rigid) have been translated to sorts along with a rigid "isAlive" predicate for each class. For example, we have sorts *IRRDMObject*, *BoundaryNode*, etc. Built-in types, for example, UID, etc. have also been translated to sorts. Next the class hierarchy has been translated into subsorting definitions. For example, we have $Node < IRRDMObject$ etc. Rigid compositions have been translated into predicates with axioms for the multiplicity constraints. For example, *__area__* and the corresponding axiom for this predicate stating that there is at least one DimArea for each CivilStructure. Similarly, rigid associations have been translated into predicates with axioms capturing the multiplicity constraints. Properties have then been translated into operations. For example, the property of Edges to have a start Node has been translated to $startnode : Edge \rightarrow Node$. Finally, the dynamic association "shows" for SignalHeads has been translated into the flexible predicate $\_\_shows\_\_ : SignalHead \times SignalAspect$.

8.3.1 Translation to CASL

To conclude the application of the second step of our methodology, we can now translate the ModalCasl specification gained above to Casl. This translation can be performed automatically using the Hets toolset and the comorphism discussed in Section 4.6.4. To provide a feeling of the resulting Casl, we present a modelling of the track plan in Figure 8.3. An excerpt of this model is given in Figure 8.5. We note that once again, this model includes the renaming of the modality introduced by the comorphism to the sort Time, and the instantiation of several types.

```
spec IRDMSignature =
     Time
then sorts Node, Edge, DimArea, CivilStructure, ... < IRRDMObject
     sort  UID
     sorts BoundaryNode, SimpleNode, JunctionNode < Node
     sort  Signal < TrackSideEquipment
     sort  Route < DimArea
     sort  TrackCircuit < TrackEquipment
     ...
     free type UID ::= bn1 | bn2 | jn1 | jn2 | sn1 | ...
```

```
    free type Node ::= BN1 | BN2 | JN1 | JN2 | SN1 | SN2 | ...
    free type BoundaryNode ::= BN1 | BN2
    free type Route ::= R1A | R1B | R2
    free type Signal ::= S1 | S2
    ...
    op    id : IRRDMObject → UID
    op    primaryEdge : BoundaryNode → UID
    op    secondaryEdge : BoundaryNode → UID
    op    edge1 : SimpleNode → UID
    ...
    pred  controls : Signal × Route
    ...
    • id(BN1) = bn1
    • id(BN2) = bn2
    • id(E1) = e1
    ...
    • secondaryEdge(BN1) = e1
    • primaryEdge(JN1) = e1
    • secondaryEdge(JN1) = e2
    • edge1(SN1) = e2
    ...
    • controls(S1, R1A)
    • controls(S1, R1B)
    • controls(S2, R2)
    ...
end
```

Figure 8.5: Example CASL model of the track plan in Figure 8.3.

Considering the specification in Figure 8.5, we can see how several of the track plan elements are modelled. For example, we can see how every element of the model has a *UID* associated to it through the operation *id* : *IRRDMObject* → *UID* (note that every sort is a subsort of *IRRDMObject*). Similarly, we can see how the topology of the track plan is captured through axioms controlling the use of the operations *primaryEdge*, *secondaryEdge* etc. Finally, we can see that certain signals control certain routes via the predicate modelling the *controls* association. For example, we have that signal *S1* *controls* *RA1*.

8.4 Modelling Industrial Standard Dynamics

The IRDM has the aim of capturing all the concepts within the railway domain, however it does not cover full details of how these concepts can change over time. In several places, there are references to the dynamics of railways within the narrative describing classes. For example, for the route class, the following can be found in the narrative: "The Route class defines a signalling route in the data and is derived from the foundation DimArea class. The route is treated as a signalling area." This clearly shows that Routes are used as an area for signalling, and hence are a data element that is used by the dynamic signalling systems. Therefore, to model the dynamic aspects of railways that we are interested in for verification, we use the standard signalling approach for movement authorities from the ERTMS standard [ERT02]. This signalling approach is the same as the approach taken in Chapter 6. However, here we show how such a signalling approach can be defined on top of the data elements of the IRDM.

We begin by considering the notion of a route. As we can see from Figure 8.3, in the Invensys Rail Data Model a route covers a dimensioned area that includes a series of track circuits. For example, the route *R1B* includes track circuits *TC2*, *TC7* and *TC8*. Such track circuits are the railway components that are used to detect whether or not a train is occupying that section of track. Hence, it is standard within the railway domain that concrete tracks appearing within a physical railway have a corresponding track circuit. When considering a typical control table, the clear column captures exactly which track circuits need to report that they are clear of trains for a route to be declared as available. Figure 8.6 gives the control table in terms of track circuits for the routes of the track plan in Figure 8.3.

Route	Clear	Normal	Reverse
R1A	TC2, TC3, TC4, TC5		P1
R1B	TC2, TC7, TC6, TC8	P1, P2	
R2	TC6, TC8		P2

Figure 8.6: A control table for the track plan in Figure 8.3.

In a similar manner to the clear predicate in Chapter 6, we introduce a predicate *isClearAt* that captures whether a track circuit is clear at a given Time:

$$\textbf{pred} \quad \_\_isClearAt\_\_ : TrackCircuit \times Time$$

This now allows us to express that Routes are formed of such detection units, i.e. are lists of track circuits:

$$\textbf{sort} \quad Route < List[TrackCircuit]$$

Here we note, that this definition fits with the multiplicity constraints imposed by the IRDM towards Routes being a sub type of DimAreas. That is, Routes must contain at least one TrackCircuit (via compositions through DimEdgeSection and TrackEquipment).

Considering the modelling of dynamics using movement authorities and regions in Chapter 6, we can see that this definition of the type Route is in fact, very close to Bjørner's definition of the type Route. Namely that both are a list type over elements that can be open (in Bjørner's terms) or clear (in Invensys terms). This is no coincidence, as the dynamic modelling we have provided for Bjørner's DSL follows a standard used throughout the railway domain. However, it is common practice for different railway companies to use differing terminologies for concepts within the railway domain. Hence, with a small adoption to the dynamic modelling of movement authorities, namely the replacement of unitPathPairs with TrackCircuits as route elements, we can re-use the domain specific theory and lemmas we have presented for supporting verification of scheme plans formulated over Bjørner's DSL. That is, we specify the following type system for regions and movement authorities:

sort $Region < List[TrackCircuit]$
sort $MA < List[Region]$

From this point onward, the general rules for the changing of movement authorities and control tables can be adapted to the new type system and hence re-used. For example, we can adapt the definition of a control table and route openness from our dynamic modelling for Bjørner's DSL to consider the terminology used by Invensys. That is, we can use TrackCircuits for the definition of clear instead of Units:

```
SPEC ControlTable =
       IRDMControl
then ∀ r : Route; t : Time
       • r isOpenAt t
         ⇔ ∀ tc : TrackCircuit • clear(r, tc) ⇒ tc isClearAt t
end
```

As a larger example of the adaption outlined above, in Figure 8.7 we have included an excerpt of the encoding of the dynamics for the track plan in Figure 8.3. This example illustrates the definition of the concrete routes of the track plan. It also gives a definition for the concrete regions formed over the track plan and the relation between routes and regions through the *regions* operation.

```
spec StationRoutes =
       IRDMSignature
then Control
then
       ...
       • R1A = TC2 :: (TC3 :: (TC4 :: (TC5 :: [])))
       • R1B = TC2 :: (TC7 :: (TC6 :: (TC8 :: [])))
       • R2 = TC6 :: (TC8 :: [])

       free type Region ::= RG1 | RG2 | RG3 | RG4 | RG5
```

- $RG1 = TC2 :: []$
- $RG2 = TC3 :: (TC4 :: (TC5 :: []))$
- $RG3 = TC7 :: []$
- $RG4 = TC6 :: (TC8 :: [])$
- $regions(R1A) = RG1 :: (RG2 :: [])$
- $regions(R1B) = RG1 :: (RG3 :: (RG4 :: []))$
- $regions(R2) = RG4 :: []$

...

end

Figure 8.7: Example CASL model for the dynamics of the track plan in Figure 8.3

This concludes the application of the third step of our methodology, namely the extension of the IRDM DSL gained from the UML class diagram with a modelling of the natural language dynamical aspects.

8.5 Supporting Verification

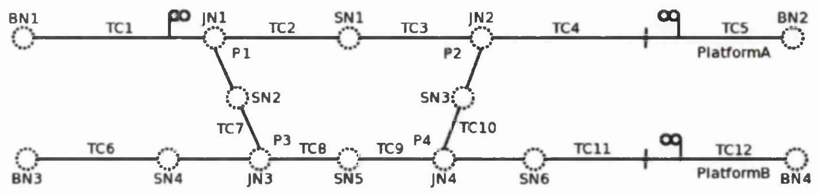
Finally, the last step of our methodology is to support the extended DSL with a series of lemmas to support verification. Here, the dynamics that have been added and the safety property we consider are standard for movement authorities. This means that we can re-use the theory developed in Chapter 6. That is, the domain specific lemmas presented in Chapter 6 can be re-used with this new type system for IRDM. Hence, they can also be applied to aid with verification of scheme plans modelled over the IRDM. This illustrates an important point within the railway domain. Namely that the domain specific lemmas we have introduced are built around insights within the railway domain on the operation of movement authorities. Hence, with small adaptations, these lemmas can be applied throughout different modelling approaches towards the domain. This in turn illustrates that the design of a language incorporating a solid domain understanding can lead to results that improve verification within that domain. Overall, the lemmas we have presented provide scalable verification over different starting DSLs and for the verification of movement authorities in general.

8.6 Formulating and Verifying IRDM Models

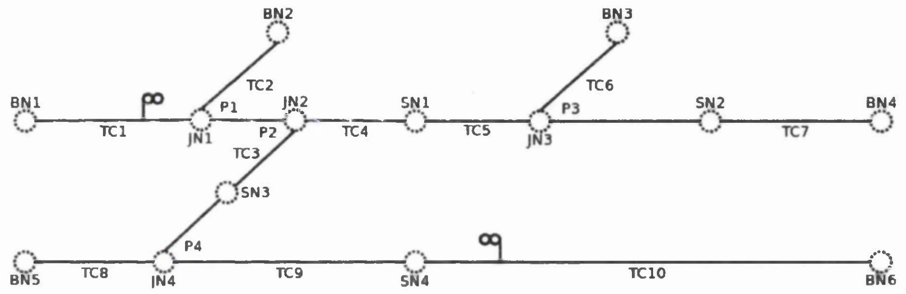
To illustrate that the modelling and verification of models formulated using the IRDM DSL presented above is possible, we once again consider the track plans from Figure 6.7. Each of these have been modelled, along with the track plan in Figure 8.3, using the IRDM. These track plans are illustrated in Figure 8.8. The automatic verification results are then given in Figure 8.9.

Here, we can see some interesting results. Firstly, the verification times are much smaller than those obtained from Bjørner's setting. Although we note that this is down

(a) A pass through station (TP-A) modelled in IRDM.



(b) A double junction (TP-C) modelled in IRDM



(c) A terminal station (TP-D) modelled in IRDM

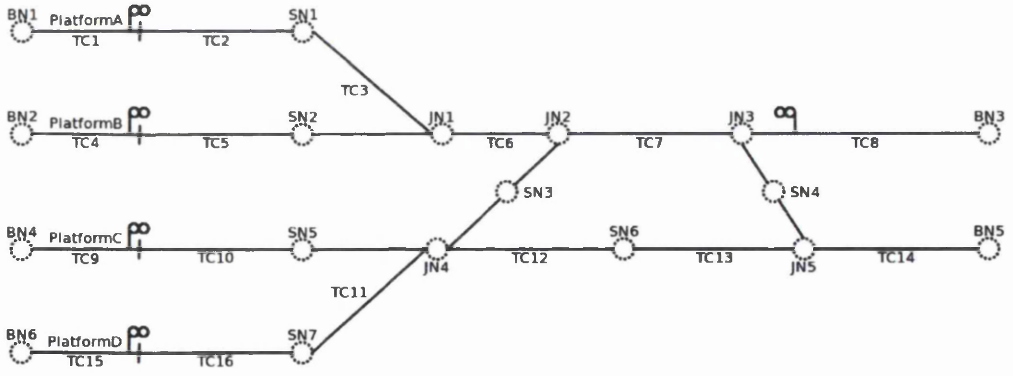


Figure 8.8: Various track plans modelled and verified using IRDM.

to the higher level of track abstraction we have used when modelling the scheme plans with the IRDM setting. For example, units $LA1$, $LA2$, $LA3$ and $LA4$ from modelling the station using Bjørner's approach (Figure 3.1) are considered as a single edge and hence track circuit in the IRDM based modelling (TP-A in Figure 8.8). This has the effect of reducing the length of lists representing routes, and hence makes exploration of these lists easier for automatic provers. Also, we note that the elements contained within the list structures in Bjørner's setting, namely pairs of units and paths, are much more complicated than the simple track circuit elements within the IRDM setting. Finally, the last point of interest with these results is the rise in time to verify TP-D. Whereas with Bjørner's setting the proof time for TP-A was the largest due to routes

Track Plan	Route Lemma Times (s)	Proof Time(s)	Avg. Memory (MB)
TP-A	9.79	1.58	36.63
TP-B	1.65	1.12	6.85
TP-C	4.76	1.24	22.49
TP-D	45.32	2.69	76.10

Figure 8.9: Verification times for the given track plans formulated using IRDM.

for this track plan containing many elements. Here we see that the main increase in time is caused by the increase in the number of routes being considered, i.e. TP-D has eight routes while TP-B only has four routes. Again, we believe this is due to the fact that each route in the above IRDM models contains a smaller number of elements than the corresponding route in the model formulated using Bjørner's DSL .

This concludes our application of our methodology to the industrial setting of the Invensys Rail Data Model. We have illustrated that our methodology is not only useful for faithfully capturing the IRDM, but also provides a scalable verification process for real world scheme plans.

Part III

Conclusions

Chapter 9

Conclusions and Future Work

Contents

The final chapter of this thesis presents a short summary of the work that has been completed and reviews the main contributions of the thesis. We then discuss directions for possible future work that complements and extends the work we have presented.

9.1 A Methodology for DSL Design

In this thesis, we have introduced a novel design methodology for creating domain specific languages for system specification and verification. We have supported our hypothesis that domain specific languages can aid with verification and shown it to be valid within the railway domain. The methodology we have proposed is based on the CASL specification language and, taking industrial documents as a starting point, results in a domain specific tooling environment for modelling and verification. Concretely, the methodology outlines two processes, which for readability are given again in Figures 9.1 and 9.2. The first process, illustrated in Figure 9.1, focuses on designing a domain specific language and associated tool support. The second process, illustrated in Figure 9.2, is a verification process that can be followed when applying the tools output from the design phase. To illustrate that our methodology is useful in practice, we have applied both these processes to two separate approaches within the railway domain. The first approach considers the academically based domain specification by Bjørner [Bjø00, Bjø03, BCJ+04, Bjø09]. The second approach considers the industrial data model provided by our industrial partners Invensys Rail [Rail0]. For each of these examples, we have provided concrete evidence of faithful modelling and successful verification of several real world railway scheme plans. Finally, we have presented and discussed the OnTrack toolset which is the result of applying our methodology to Bjørner's domain specification. To ensure that our methodology meets our three aims of providing (1) faithful modelling and (2) scalable verification in a manner that is (3) accessible to domain engineers, we have provided the following main contributions within this thesis:

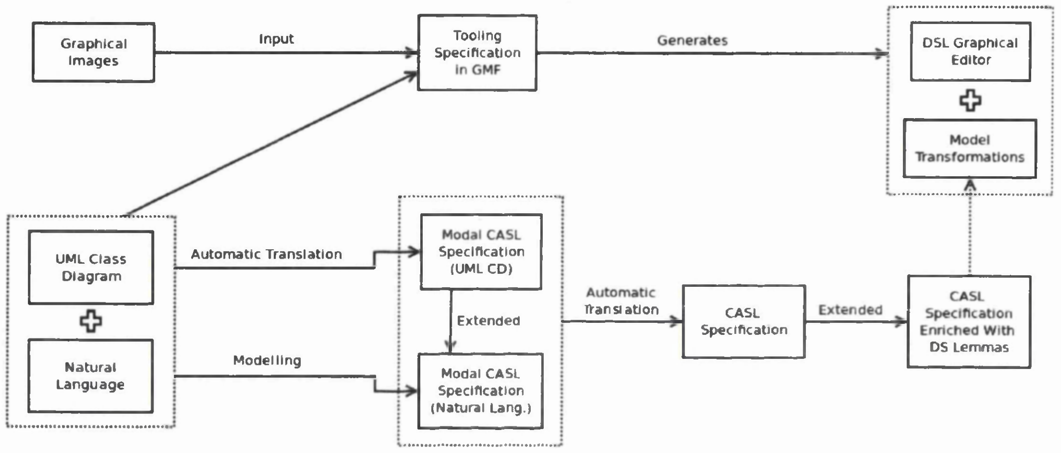


Figure 9.1: Our methodology for designing domain specific languages aimed at verification.

Faithful capture of UML class diagrams in MODALCASL*:* The first result of this thesis is to provide the theoretical framework allowing one to utilise industrial DSLs formulated as UML class diagrams for verification. To achieve this, we have presented an institution for capturing UML class diagrams. We have then shown how dynamical aspects of UML class diagrams can be captured via an institution independent construction. Overall, this allows the direct importation of informal specifications from industry into the design methodology we have proposed.

Domain specific lemmas for the railway domain allowing scalable verification: The second part of the methodology conjectures that the close incorporation of domain knowledge into the system modelling process can provide useful lemmas for verification. We have given examples of such lemmas for the railway domain, and shown how they can be incorporated into the DSL being designed. We have then illustrated that these lemmas allow for successful verification of models formulated in the DSL, concretely for several real world railway scheme plans. Interestingly, the domain specific lemmas we have presented have shown to be applicable to two different approaches towards formalising the railway domain. We conjecture that such lemmas are likely to exist in other domains where verification is required.

Development of accessible graphical tool support: Finally, we have presented the OnTrack toolset for generation of formal models from graphical scheme plans. OnTrack is the resulting DSL editor gained from applying our methodology and encapsulates the presented verification process for the railway domain. OnTrack provides railway engineers with a graphical form of specification with supported verification techniques for scheme plans within the railway domain.

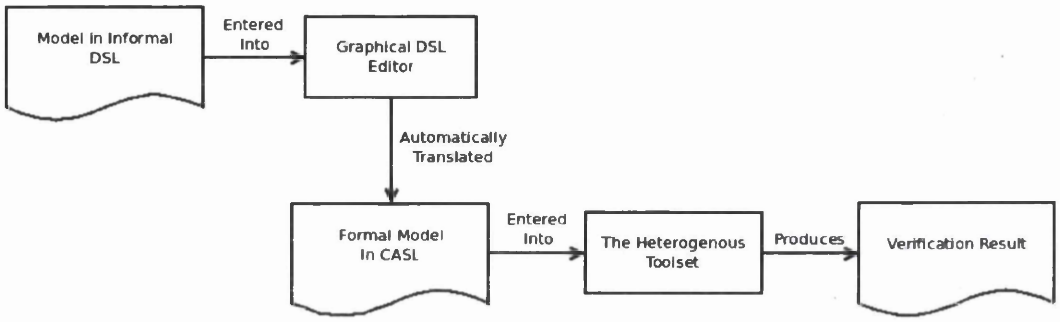


Figure 9.2: A verification process based on the designed tools.

Putting these point together, this thesis has successfully supported our original hypothesis that domain specific languages can aid with modelling and verification, in particularly within the railway domain. Along with supporting our hypothesis, we have also successfully illustrated: (1) a first use of algebraic specification for modelling railway systems; and (2) the use of automated theorem proving for railway verification on the design level. Bringing these points together results in a strong case for the use of domain specific languages in the setting of system specification and verification.

9.2 Future Research Directions

There are a number of interesting extensions that could be considered for the work we have presented. Below we outline some possible future research directions.

Backwards model transformations for failed proofs: The methodology we have presented does not explore how visual feedback of failed proof attempts could be displayed to the user. Currently, the only feedback provided to the user is in the form of a named proof goal that has failed. Even though, this provides the ability to show, for example, which route a particular proof fails for, finding problems within the scheme plan editor can still be tedious. Hence, a particularly interesting topic for future work would be to explore how to present failed proofs graphically on the scheme plan level. Such visualisations have been considered by Marchi et al. [dSWP11], and it would be interesting to investigate if such visualisations are possible within OnTrack. One possible approach to solve this, could be to implement a model transformation from the output of the proof, back to the graphical DSL meta model. This would require some elements for failed proof representation to be added to the DSL meta model. Then, a graphical definition would have to be developed for displaying elements of this type when they occur within a model. Finally, a backwards model transformation could be defined, instantiating these elements with details of the failed verification attempt.

Implementation of the presented comorphism in Hets: In Chapter 5 of this thesis, we have defined both an institution for UML class diagrams and also a comorphism into ModalCasl. Currently, the formalisation of a UML class diagram and also obtaining the resulting ModalCasl from applying the given comorphism is a process that must be completed by hand. Such a task is often tedious, and also error prone. The Heterogeneous Toolset (Hets) [MML07] makes extensive use of institutions and their relations to provide tool support for various logics. Hence, one future direction of research could be to implement the presented institutions and translations within Hets. This would require the implementation of a parser, static analyser and abstract syntax for the new UML class diagram institution within Hets. Finally, an implementation of the comorphism to ModalCasl would also be required.

Abstractions for CASL: In Section 7.5, we explored an abstraction technique by Moller et al. [MNR+13] for simplifying scheme plans before verification. As this technique is currently implemented within OnTrack at the DSL level, it is possible to apply the technique before generating a Casl specification representing the resulting abstract scheme plan. Currently, the correctness of this abstraction within the Casl modelling approach has not been considered. Hence, for it to be applied to verify, for example, abstract versions of the scheme plans in Figure 6.7, a formal proof for the abstraction should be given over the presented Casl modelling approach.

Technology transfer to Invensys Rail: Within the thesis we have successfully shown that two out of the three of the main steps of our methodology can be applied to the Invensys Rail Data Model [Rai10]. However, we have not presented or yet implemented a graphical tooling environment based around this data model. Discussions towards the development of this environment and technology transfer to Invensys Rail are currently ongoing. It is currently unclear if a new tool will be defined and developed, or if OnTrack will be extended with model transformations between Bjørner's DSL and the Invensys Rail data model. This second approach would allow the re-use of the current graphical front-end from OnTrack for generation of models formulated within the Invensys Rail data model.

Generalising our approach to other domains: Finally, the work we have presented in this thesis takes a first step towards the field of domain specific languages being designed for verification. Considering a larger setting, it would be interesting to consider further domains of application, for example within the field of financial systems or air traffic control. Such an application could lead to a generalisation or refinement of our methodology, improving its applicability to other domains. For example, a wider application of the methodology could lead to a general theory for the design and development of domain specific lemmas supporting verification that is independent of the domain.

Bibliography

[ABH$^+$10] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in event-b. *International journal on software tools for technology transfer*, 12(6):447–466, 2010.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering.* Cambridge University Press, 2010.

[adv] Advance FP7 project. `http://www.advance-ict.eu/`. Last accessed: 12/09/2013.

[AG07] Siemens AG. Siemens Railcom Manager. `http://w3.siemens.co.uk/mobility/uk/en/rail_solutions/rail_automation/control_information/railcom/pages/railcommanager.aspx`, 2007. Last accessed: 12/09/2013.

[Ant11] Marc Antoni, 2011. Practical formal validation method of interlocking systems, DCDS'11 Keynote Speaker.

[AvDR95] B.R.T Arnold, A. van Deursen, and M. Res. An algebraic specification of a language for describing financial products. In Martin Wirsing, editor, *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13. IEEE, April 1995.

[Bar97] J. Barnes. *High integrity Ada: the SPARK approach.* Addison-Wesley, 1997.

[BBG$^+$05] Sami Beydeda, Matthias Book, Volker Gruhn, et al. *Model-driven software development*, volume 15. Springer Heidelberg, 2005.

[BCJ$^+$04] D. Bjørner, P. Chiang, M. S. T. Jacobsen, J. K. Hansen, M. P. Madsen, and M. Penicka. Towards a formal model of CyberRail. In R. Jacquart, editor, *International Federation for Information Processing: Congress Topical Sessions*, volume 156, pages 657–664. Kluwer, 2004.

[BCS99] D. Bjørner, G. Chris, and P. Søren. Scheduling and Rescheduling of Trains. *LNCS*, 1999.

[Ber89] J. A. Bergstra. *Algebraic Specification.* ACM, 1989.

[BFRS00] D. Bonachea, K. Fisher, A. Rogers, and F. Smith. Hancock: a language for processing very large-scale data. *SIGPLAN Notice*, 35(1):163–176, 2000.

[BG00] J. Boulanger and M. Gallardo. Validation and verification of METEOR safety software. In j. Allen, R. J. Hill, C. A. Brebbia, G. Sciutto, and S. Sone, editors, *Computers in Railways VII*, volume 7, pages 189–200. WIT Press, 2000.

[BGP99] D. Bjørner, C. George, and S. Prehn. Scheduling and rescheduling of trains. *Industrial Strength Formal Methods in Practice*, 1:157–184, 1999.

[BH06] Jonathan P. Bowen and Michael G. Hinchey. Ten commandments of formal methods ...ten years later. *IEEE Computer*, 39(1):40–48, 2006.

[Bjø00] Dines Bjørner. Formal Software Techniques for Railway Systems. *CTS2000: 9th IFAC Symposium on Control in Transportation Systems*, pages 1–12, 2000.

[Bjø03] D. Bjørner. Dynamics of Railway Nets: On an Interface between Automatic Control and Software Engineering. *CTS2003: 10th IFAC Symposium on Control in Transportation Systems*, 2003.

[Bjø09] Dines Bjørner. *Domain Engineering Technology Management, Research and Engineering.* Japan Advanced Institute of Science and Technology, 2009.

[BM04] M. Bidoit and P.D. Mosses. CASL *User Manual – Introduction to Using the Common Algebraic Specification Language*, volume 2900 of *LNCS*. Springer, 2004.

[Bru97] D. Bruce. What makes a good domain-specific language? APOSTLE, and its approach to parallel discrete event simulation. In P. Ciancarini and C. Hankin, editors, *Proceedings First ACM Sigplan Workshop on Domain-Specific Languages.* ACM, 1997.

[CEW93] Ingo Classen, Hartmut Ehrig, and Dietmar Wolz. *Algebraic specification techniques and tools for software development: the ACT approach.* World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1993.

[CJKW07] Steve Cook, Gareth Jones, Stuart Kent, and Alan Cameron Wills. *Domain-specific development with visual studio dsl tools.* Addison-Wesley Professional, 2007.

[CK08] MarÃa Victoria Cengarle and Alexander Knapp. An institution for UML 2.0 static structures. Technical Report TUM-I0807, Technische Universitat Munchen, 2008.

[CKTW08] María Victoria Cengarle, Alexander Knapp, Andrzej Tarlecki, and Martin Wirsing. A Heterogeneous Approach to UML Semantics. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models*, LNCS 5065, pages 383–402. Springer, 2008.

[DF98] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing.* World Scientific, 1998.

[DHK96] Arie van Deursen, Jan Heering, and Paul Klint, editors. *Language Prototyping: An Algebraic Specification Approach: Vol. V.* World Scientific Publishing Co., Inc., 1996.

[dSWP11] Osmar Marchi dos Santos, Jim Woodcock, and Richard F. Paige. Using model transformation to generate graphical counter-examples for the formal analysis of xUML models. In *ICECCS*, pages 117–126. IEEE Computer Society, 2011.

[ecl] The Eclipse IDE homepage. `http://www.eclipse.org/`. Last accessed: 12/09/2013.

[EFH$^+$04] Sven Efftinge, Peter Friese, Arno Hase, Dennis Hübner, Clemens Kadura, Bernd Kolb, Jan Köhnlein, Dieter Moroff, Karsten Thoms, Markus Völter, et al. Xpand documentation. Technical report, Technical report, Eclipse Documentation. 2004-2010., 2004.

[EFT94] H.D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical logic.* Springer, 1994.

[EM85] Hartmut Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I.* Springer, Secaucus, NJ, USA, 1985.

[ERT02] ERTMS User Group. UNISIG: ERTMS/ETCS system requirements specification, 2002.

[FH98] Wan Fokkink and Paul Hollingshead. Verification of interlockings: from control tables to ladder logic diagrams. In J.F. Groote, S.P. Luttik, and J.J. van Wamel, editors, *FMICS'98.* CWI, 1998.

[FMGF11] A. Ferrari, G. Magnani, D. Grasso, and A. Fantechi. Model checking interlocking control tables. *FORMS/FORMAT 2010*, pages 107–115, 2011.

[FP10] M. Fowler and R. Parsons. *Domain Specific Languages.* Addison-Wesley, 1 edition, 2010.

[GB92] J. A. Goguen and R. M. Burstall. Institutions: abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.

[GH93] John V. Guttag and James J. Horning. *Larch: languages and tools for formal specification.* Springer, New York, NY, USA, 1993.

[Gra95] I. S. Graham. *The HTML SourceBook.* John Wiley and Sons, 3 edition, 1995.

[Gro09] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit.* Addison-Wesley Professional, 2009.

[GvVK95] J. F. Groote, S.F.M. van Vlijmen, and J.W.C. Koorn. The safety guaranteeing system at station hoorn-kersenboogerd. Technical report, Utrecht University, 1995.

[GWM$^+$93] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Applications of Algebraic Specification using OBJ.* Cambridge University Press, 1993.

[Hax12] A. E. Haxthausen. Automated generation of safety requirements from railway interlocking tables. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*, volume 7610 of *Lecture Notes in Computer Science*, pages 261–275. Springer Berlin Heidelberg, 2012.

[HC96] George Edward Hughes and Maxwell John Cresswell. *A new introduction to modal logic.* Burns & Oates, 1996.

[HCB00] H. Hussmann, M. Cerioli, and H. Baumeister. From uml to casl (static part). Technical Report DISI-TR-00-06, DISI-Universit di Genova, 2000.

[HHKR89] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF Reference Manual. *SIGPLAN Notice*, 24(11):43–75, 1989.

[HKRS09] G. Holland, T. Kahsai, M. Roggenbach, and B. H. Schlingloff. Towards formal testing of jet engine Rolls-Royce BR725. In L. Czaja and M. Szczuka, editors, *Proceedings 18th International Conference on Concurrency, Specification and Programming.* Springer, 2009.

[Hol04] John Holt. Uml for systems. *Engineering Watching the Wheels. The Institute of Electrical Engineers*, 2004.

[HP88] T. Hopkins and C. Phillips. *Numerical methods in practice: using the NAG library.* Addison Wesley, 1988.

[HP00] A. E. Haxthausen and J. Peleska. Formal development and verification of a distributed railway control system. *IEEE Trans. Software Eng.*, 26(8):687–701, 2000.

[HP07] Anne Haxthausen and Jan Peleska. A domain-oriented, model-based approach for construction and verification of railway control systems. In Cliff Jones, Zhiming Liu, and Jim Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems*, volume 4700 of *LNCS*, pages 320–348. Springer, 2007.

[HPvD09] F. Hermans, M. Pinzger, and A. van Deursen. Domain-Specific Languages in Practice: A User Study on the Success Factors. *Model Driven Engineering Languages and Systems*, 5795:423–437, 2009.

[IEC96] Extended Backus Naur Form, 1996. ISO/IEC Standard 14977.

[ILR13] Alexei Iliasov, Ilya Lopatkin, and Alexander Romanovsky. The SafeCap platform for modelling railway safety and capacity. Technical report, Computing Science, Newcastle University, 2013.

[Inv13] Invensys Rail. `www.invensys.com`, Last accessed: 12/09/2013.

[IR12a] Alexei Iliasov and Alexander Romanovsky. SafeCap domain language for reasoning about safety and capacity. Technical report, Computing Science, Newcastle University, 2012.

[IR12b] Alexei Iliasov and Alexander Romanovsky. The SafeCap toolset for improving railway capacity while ensuring its safety. Technical report, Computing Science, Newcastle University, 2012.

[Jac04] R. Jacquart, editor. *IFIP 18th World Computer Congress, Topical Sessions.* Kluwer, 2004.

[Jam10] P. James. SAT-based Model Checking and its applications to Train Control Software. Master's thesis, Swansea University, 2010.

[JBR13] Phillip James, Arnold Beckmann, and Markus Roggenbach. Using domain specific languages to support verification in the railway domain. In Armin Biere, Amir Nahir, and Tanja Vos, editors, *Hardware and Software: Verification and Testing*, volume 7857 of *Lecture Notes in Computer Science.* Springer, 2013.

[JKL$^+$13] Phillip James, Karim Kanso, Andy Lawrence, Faron Moller, Markus Roggenbach, Monika Seisenberger, and Anton Setzer. Verification of solid state interlocking programs. In *FM-RAIL-BOK 2013*, To Appear, 2013.

[JKMR13] Phillip James, Alexander Knapp, Till Mossakowski, and Markus Roggenbach. Designing domain specific languages – a craftsman's approach for the railway domain using casl. In *WADT 2012*, Lecture Notes in Computer Science. Springer, 2013.

[JMN+13] Phillip James, Faron Moller, Hoang Nga Nguyen, Markus Roggenbach, Steve Schneider, Helen Treharne, Matthew Trumble, and David Williams. Verification of Scheme Plans using CSP||B. In *FM-RAIL-BOK 2013*, To Appear, 2013.

[JR10] Phillip James and Markus Roggenbach. Automatically Verifying Railway Interlockings using SAT-based Model Checking. In Jens Bendispoto, Michael Leuschel, and Markus Roggenbach, editors, *AVoCS'10 – Proceedings of the Tenth International Workshop on Automated Verification of Critical Systems*, volume 35. Electronic Communications of the EASST, 2010.

[JR11] Phillip James and Markus Roggenbach. Designing domain specific languages for verification: First steps. In Georg Struth Peter Hofner, Annabelle McIver, editor, *ATE-2011 – Proceedings of the First Workshop on Automated Theory Engineering*, volume 760 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.

[JTT+13] P. James, M. Trumble, H. Treharne, M. Roggenbach, and S. Schneider. OnTrack: An open tooling environment for railway verification. In *Proceedings of NFM'13: Fifth NASA Formal Methods Symposium*, 2013.

[Kan13] Karim Kanso. *Agda as a Platform for the Development of Verified Railway Interlocking Systems*. PhD thesis, Deptartment of Computer Science, Swansea University, UK, August 2013.

[Ken02] Stuart Kent. Model driven engineering. In *Integrated Formal Methods*, pages 286–298. Springer, 2002.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Trans. Softw. Eng. Methodol.*, 2(2):176–201, 1993.

[KLR96] Steven Kelly, Kalle Lyytinen, and Matti Rossi. Metaedit+ a fully configurable multi-user and multi-tool case and came environment. In Panos Constantopoulos, John Mylopoulos, and Yannis Vassiliou, editors, *Advanced Information Systems Engineering*, volume 1080 of *Lecture Notes in Computer Science*, pages 1–21. Springer Berlin Heidelberg, 1996.

[KMS08] K. Kanso, F. Moller, and A. Setzer. Verification of safety properties in railway interlocking systems defined with ladder logic. In M. Calder and A. Miller, editors, *AVOCS08*. Glasgow University, 2008.

[Knu64] D. E. Knuth. Backus Normal Form vs. Backus Naur Form. *Communications ACM*, 7(12):735–736, 1964.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, 1988.

[KR01] D. Kerr and T. Rowbotham. *Introduction To Railway Signalling.* Institution of Railway Signal Engineers, 2001.

[KRPP13] D.S. Kolovos, L. Rose, R.F. Paige, and F.A. Polack. The Epsilon Book, 2013.

[Kru92] C. W. Krueger. Software reuse. *ACM Computing Survey*, 24(2):131–183, 1992.

[KST97] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The definition of extended ML: a gentle introduction. *Theoretical Computer Science*, 173(2):445–484, 1997.

[Kus06] Jochen Kuster. Definition and validation of model transformations. *Software and Systems Modeling*, 5(3):233–259, 2006.

[LCA04] K. Lano, D. Clark, and K. Androutsopoulos. UML to B: Formal Verification of Object-Oriented Models. In *IFM'04*, LNCS 2999, pages 187–206. Springer, 2004.

[LFFP11] M. Leuschel, J. Falampin, F. Fritz, and D. Plagge. Automated property verification for large scale B models with ProB. *Formal Asp. Comput.*, 23(6):683–709, 2011.

[LMB92] J. Levine, T. Mason, and D. Brown. *Lex and Yacc, 2nd edition.* O'Reilly, 1992.

[LV02] Juande Lara and Hans Vangheluwe. Atom3: A tool for multi-formalism and meta-modelling. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin Heidelberg, 2002.

[MA03] S. Meng and B. Aichernig. Towards a Coalgebraic Semantics of UML: Class Diagrams and Use Cases. Technical report, Technical Report 272, UNU/IIST, 2003.

[MAH06] Till Mossakowski, Serge Autexier, and Dieter Hutter. Development graphs: proof management for structured specifications. *The Journal of Logic and Algebraic Programming*, 67(1):114–145, 2006.

[MC10] Till Mossakowski and Girlea Codruta. An Extended Modal Logic Institution. Master's thesis, Bremen University, 2010.

[Mew09] Kirsten Mewes. Domain-specific modeling, validation, and verification of railway control systems. In Holger Giese, Michaela Huhn, Ulrich Nickel, and Bernhard Schätz, editors, *MBEES*, volume 2009-01 of *Informatik-Bericht*, pages 57–66. TU Braunschweig, Institut für Software Systems Engineering, 2009.

[Mew10] Kirsten Mewes. *Domain-specific Modelling of Railway Control Systems with Integrated Verification and Validation.* PhD thesis, University of Bremen, 2010.

[MFS$^+$07] Clavel Manuel, Duran Francisco, Eker Steven, Lincoln Patrick, Martí-Oliet Narciso, Meseguer Jose, and Talcott Carolyn. *All about Maude - A High-Performance Logical Framework*, volume XXII. Springer, 2007.

[MHS05] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Survey*, 37(4):316–344, 2005.

[ML98] Saunders Mac Lane. *Categories for the working mathematician.* Graduate texts in mathematics. Springer, 2nd edition, 1998.

[MML07] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set, HETS. In O. Grumberg and M. Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science.* Springer, 2007.

[MNR$^+$12a] Faron Moller, Hoang Nga Nguyen, Markus Roggenbach, Steve Schneider, and Helen Treharne. CSP||B Modelling for Railway Verification: The Double Junction Case Study. In *AVOCS'12*, 2012.

[MNR$^+$12b] Faron Moller, Hoang Nga Nguyen, Markus Roggenbach, Steve Schneider, and Helen Treharne. CSP||B modelling for railway verification: the Double Junction case study. Technical report, Department of Computing Technical Report CS-12-03, 2012.

[MNR$^+$12c] Faron Moller, Hoang Nga Nguyen, Markus Roggenbach, Steve Schneider, and Helen Treharne. Using ProB and CSP||B for railway modelling. In *Proceedings of the Posters and Tool demos Session, iFM 2012 and ABZ 2012*, 2012.

[MNR$^+$13] Faron Moller, Hoang Nga Nguyen, Markus Roggenbach, Steve Schneider, and Helen Treharne. Defining and Model Checking Abstractions of Complex Railway Models using CSP||B. In Armin Biere, Amir Nahir, and Tanja Vos, editors, *Hardware and Software: Verification and Testing*, volume 7857 of *Lecture Notes in Computer Science.* Springer, 2013.

[Mor93] Matthew J. Morley. Safety in railway signalling data: A behavioural analysis. In *6th International Workshop on HOLTPA*, pages 464–474. Springer, 1993.

[Mos97] Peter D. Mosses. CoFI: The common framework initiative for algebraic specification and development. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT'97*, LNCS 1214, pages 115–137. Springer, 1997.

[Mos02] Till Mossakowski. Relating CASL with other specification languages: The institution level. *Theoretical Computer Science*, 286(2):367–475, 2002.

[Mos04a] T. Mossakowski. ModalCASL — Specification with Multi-Modal Logics. Language Summary, 2004.

[Mos04b] P. D. Mosses, editor. *CASL Reference Manual*, volume 2960. Springer, 2004.

[MY09] Ahmad Mirabadi and Bemani Yazdi. Automatic generation and verification of railway interlocking control tables using FSM and NuSMV. *Signal*, 3, 2009.

[NPW02] T. Nipkow, L. C. Paulon, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic.* LNCS 2283. Springer, London, UK, 2002.

[Obj11] Object Managment Group. Unified Modeling Language (UML), v2.4.1, 2011.

[O'R12] Liam O'Reilly. *Structured Specification with Processes and Data — Theory, Tools and Applications.* PhD thesis, Swansea University, 2012.

[PALG08] Francisco Perez Andres, Juan Lara, and Esther Guerra. Domain specific languages with graphical and textual views. In Andy Schurr, Manfred Nagl, and Albert Zundorf, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 5088 of *Lecture Notes in Computer Science*, pages 82–97. Springer Berlin Heidelberg, 2008.

[PAMC06] Vicente Pelechano, Manoli Albert, Javier Muñoz, and Carlos Cetina. Building tools for model driven development comparing Microsoft DSL tools and Eclipse modeling plugins. In *Proceedings of the 11th Conference on Software Engineering and Database.* ACM, 2006.

[PGHD04] J. Peleska, D. Große, A. E. Haxthausen, and R. Drechsler. Automated verification for train control systems. In E. Schnieder and G. Tarnai, editors, *Proceedings of Formal Methods for Automation and Safety in Railway and Automotive Systems.* Technical University of Braunschweig, 2004.

[Pyl85] I.C. Pyle. *The Ada programming language.* Prentice Hall, 1985.

[RAI93] RAISE Language Group. *The RAISE specification language.* Prentice Hall, 1993.

[Rai10] Invensys Rail. Invensys Rail Data Model – Version 1A, 2010.

[RLHJ99] D. Raggett, A. Le Hors, and I. Jacobs. HTML 4.01 Specification. *W3C recommendation*, 24:1–389, 1999.

[RV01] Alexandre Riazanov and Andrei Voronkov. Vampire 1.1. In *Automated Reasoning*, pages 376–380. Springer, 2001.

[SAA02] Gwen Salaün, Michel Allemand, and Christian Attiogbé. Foundations for a combination of heterogeneous specification components. *Electronic Notes in Theoretical Computer Science*, 66(4):114 – 133, 2002.

[saf] The SafeCap project. `http://safecap.cs.ncl.ac.uk`. Last accessed: 12/09/2013.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[SBRG12] Denis Sabatier, Lilian Burdy, Antoine Requet, and Jérôme Guéry. Formal proofs for the nyct line 7 (flushing) modernization project. In *ABZ*, pages 369–372, 2012.

[Sch02] Stephan Schulz. E-a brainiac theorem prover. *AI Communications*, 15(2):111–126, 2002.

[Sim94] A. Simpson. A formal specification of an automatic train protection system. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *FME '94: Proceedings of the Second International Symposium of Formal Methods Europe on Industrial Benefit of Formal Methods*. Springer, 1994.

[SK03] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *Software, IEEE*, 20(5):42–45, 2003.

[SLTM91] Kari Smolander, Kalle Lyytinen, Veli-Pekka Tahvanainen, and Pentti Marttiin. Metaedit – a flexible graphical environment for methodology modelling. In Rudolf Andersen, Jr. Bubenko, JanisA., and Arne SÃ¸lvberg, editors, *Advanced Information Systems Engineering*, volume 498 of *Lecture Notes in Computer Science*, pages 168–193. Springer Berlin Heidelberg, 1991.

[Tah08] W. Taha. Domain-Specific Languages. In H. Fahmy, A. Wahba, M. El-Kharashi, A. El-Din, M. Sobh, and M.taher, editors, *Proceedings of International Conference on Com- puter Engineering and Systems (ICCES) 2008*. Ain Shams University, 2008.

[TL03] T.L. Thai and H.Q. Lam. *NET framework essentials*. O'Reilly Media, 3 edition, 2003.

[vdBvDH$^+$01] M. van den Brand, A. van Deursen, J. Heering, H. De Jong, M. De Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, et al. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. *LNCS*, 2027:365–370, 2001.

[vDK98] A. van Deursen and P. Klint. Little Languages: Little Maintenance? *Journal of Software Maintenance: Research and Practice*, 10(2):75–92, 1998.

[vDKV00] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Notice*, 35(6):26–36, 2000.

[WBH$^+$02] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobalt, and D. Topic. SPASS version 2.0. In A. Voronkov, editor, *Automated Deduction – CADE-18*, volume 2392 of *Lecture Notes in Computer Science*, pages 275–279. Springer, July 27–30 2002.

[Wex81] R.L. Wexelblat. *History of programming languages.* Academic Press, 1981.

[Wie96] Jan Wielemaker. Swi-prolog 2.7 - reference manual, 1996.

[Win02] K. Winter. Model checking railway interlocking systems. *Australian Computer Science Communications*, 24(1):303–310, 2002.

[Win12] Kirsten Winter. Optimising ordering strategies for symbolic model checking of railway interlockings. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*, volume 7610 of *Lecture Notes in Computer Science*, pages 246–260. Springer Berlin Heidelberg, 2012.

[WR03] K. Winter and N. J. Robinson. Modelling large railway interlockings and model checking small ones. In M. J. Oudshoorn, editor, *ACSC '03: Proceedings of the 26th Australasian computer science conference.* Australian Computer Society, 2003.

Appendix A

Bjørner's Narrative

Below we present Bjørner's natural language specification capturing the railway domain, as given in [Bjø03].

A.1 Structure Narrative

We introduce the phenomena of railway nets, lines, stations, tracks, (rail) units, and connectors.

1. A railway net consists of one or more lines and two or more stations.
2. A railway net consists of rail units.
3. A line is a linear sequence of one or more linear rail units.
4. The rail units of a line must be rail units of the railway net of the line.
5. A station is a set of one or more rail units.
6. The rail units of a station must be rail units of the railway net of the station.
7. No two distinct lines and/or stations of a railway net share rail units.
8. A station consists of one or more tracks.
9. A track is a linear sequence of one or more linear rail units.
10. No two distinct tracks share rail units.
11. The rail units of a track must be rail units of the station (of that track).
12. A rail unit is either a linear, or is a switch, or is a simple crossover, or is a switchable crossover, etc., rail unit.
13. A rail unit has one or more connectors.

14. A linear rail unit has two distinct connectors, a switch rail unit has three distinct connectors, crossover rail units have four distinct connectors (whether simple or switchable), etc.

15. For every connector there are at most two rail units which have that connector in common.

16. Every line of a railway net is connected to exactly two, distinct stations of that railway net.

17. A linear sequence of (linear) rail units is a non-cyclic sequence of linear units such that neighbouring units share connectors.

A.2 Dynamics Narrative

We introduce defined concepts such as paths through rail units, state of rail units, rail unit state spaces, routes through a railway network, open and closed routes, trains on the railway net, and train movement on the railway net.

1. A path, $p : P$, is a pair of connectors, (c, c'), which are distinct, and of some unit.

2. A state, σ : *Sigma*, of a unit is the set of all open paths of that unit (at the time observed).

3. A unit may, over its operational life, attain any of a (possibly small) number of different states ω, Ω.

4. A route is a sequence of pairs of units and paths – such that the path of a unit/path pair is a possible path of some state of the unit, and such that "neighbouring" connectors are identical.

5. An open route is a route such that all its paths are open. A train is modelled as a route.

6. Train movement is modelled as a discrete function (i.e., a map) from time to routes such that for any two adjacent times the two corresponding routes differ by at most one of the following:

   - a unit path pair has been deleted (removed) from one end of the route;
   - a unit path pair has been deleted (removed) from the other end of the route;
   - a unit path pair has been added (joined) from one end of the route;
   - a unit path pair has been added (joined) from the other end of the route;
   - a unit path pair has been added (joined) from one end of the route, and another unit path pair has been deleted (removed) from the other end of the route;

- a unit path pair has been added (joined) from the other of the route, and another unit path par has been deleted (removed) from the one end of the route;
- or there has been no changes with respect to the route (yet the train may have moved);

7. and such that the new route is a well-formed route.

We shall arbitrarily think of "one end" as the "left end", and "the other end", hence, as the "right end" – where 'left', in a model where elements of a list is indexed from 1 to its length, means the index 1 position, and 'right' means the last index position of the list.

Appendix B

Bjørner in ModalCasl

Below we present the MODALCASL for Bjørner's DSL.

library APPENDIXB

logic MODAL

spec MODALBJOERNER =

%% Classifiers
sorts *Net*, *Station*, *Unit*, *Connector*, *Line*, *Track*, *Linear*, *Point*, *Path*, *UnitState*, *AllStates*, *Route*, *UnitPathPair*, *ListLinear*, *ListUnitPathPair*, *SetUnitState*, *SetPath*, *PairConCon*, *PairUnitPath*, *Boolean*, *UID*

%% Hierarchy
sorts *Point*, *Linear* < *Unit*
sorts *Track*, *Line* < *ListLinear*
sort *Path* < *PairConCon*
sort *UnitState* < *SetPath*
sort *AllState* < *SetUnitState*
sort *Route* < *ListUnitPathPair*

%% Is Alive Preds
rigid preds
isAlive : *Net*;
isAlive : *Station*;
isAlive : *Unit*;
isAlive : *Connector*;
isAlive : *Line*;
isAlive : *Track*;
isAlive : *Linear*;

$isAlive : Point;$
$isAlive : Path;$
$isAlive : UnitState;$
$isAlive : AllStates;$
$isAlive : Route;$
$isAlive : UnitPathPair;$
$isAlive : ListLinear;$
$isAlive : ListUnitPathPair;$
$isAlive : SetUnitState;$
$isAlive : SetPath;$
$isAlive : PairConCon;$
$isAlive : PairUnitPath;$
$isAlive : Boolean$

%% Instance Specs
rigid op $tt : Boolean$
rigid op $ff : Boolean$

%% Associations and Compositions
rigid preds
$\_\_hasLine\_\_ : Net \times Line;$
$\_\_hasStation\_\_ : Net \times Station;$
$\_\_has\_\_ : Station \times Unit;$
$\_\_hasTrack\_\_ : Station \times Track;$
$\_\_hasConnector\_\_ : Unit \times Connector;$
$\_\_allStates\_\_ : Unit \times SetUnitState;$
$\_\_has\_\_ : Linear \times Connector;$
$\_\_has\_\_ : Point \times Connector$

flexible pred $\_\_stateAt\_\_ : Unit \times UnitState$

%% Properties
rigid ops
$id : Net \rightarrow UID;$
$isWellFormed : ListUnitPathPair \rightarrow Boolean;$
$isValidPath : PairConCon \rightarrow Boolean$

flexible ops
$isClosedAt : Unit \rightarrow Boolean;$
$isOpen : Route \rightarrow Boolean$

%% Composition and Multiplicity Axioms
• $\forall\ n : Net$
  • $\exists\ s1, s2 : Station$

$\bullet \neg\ s1 = s2 \wedge n\ hasStation\ s1 \wedge n\ hasStation\ s2$

$\bullet \forall\ s : Station \bullet \exists\ u : Unit \bullet s\ has\ u$

...

%% Alive Axioms

$\bullet\ isAlive(tt)$

$\bullet\ isAlive(ff)$

...

end

Appendix C

Bjørner in CASL

C.1 Bjørner's DSL in CASL

Below we give the signature of Bjørner's DSL after mapping it to CASL. We note that several types have been instantiated and some operations given a mixfix profile.

```
library BJOERNERALL

from BASIC/SIMPLEDATATYPES get BOOLEAN

spec PAIR[sort S][sort T] =
      sort   Pair[S,T]
      ops    first : Pair[S,T] → S;
             second : Pair[S,T] → T;
             pair : S × T → Pair[S,T]
      ∀ s : S; t : T
      • first(pair(s, t)) = s
      • second(pair(s, t)) = t
end

spec GENERATELIST[sort Elem] =
      sort   List[Elem]
      free type List[Elem] ::= [] | __::__(Elem; List[Elem])
      ops    __++__ : List[Elem] × List[Elem] → List[Elem];
             [] : List[Elem];
             __::__ : Elem × List[Elem] → List[Elem]
      ∀ x : Elem; L, K : List[Elem]
      • [] ++ K = K
      • (x :: L) ++ K = x :: (L ++ K)
end

spec LIST[sort Elem] =
      GENERATELIST[sort Elem]
then %def
      pred __eps__ : Elem × List[Elem]
      ∀ x, y : Elem; L, K : List[Elem]
      • ¬ x eps []
      • x eps x :: L
      • (x eps y :: L ⇔ x eps L) if ¬ x = y
end

spec TIME =
      sort   Time
      ops    0 : Time;
             suc : Time → Time
      pred  __<__ : Time × Time
      ∀ x, y, z : Time
      • 0 < suc(x)
      • suc(x) < suc(y) if x < y
      • ¬ suc(x) < 0
      • ¬ suc(x) < x
      • x < suc(x)
      • ¬ x < x
end

spec BJOERNERSIGNATURE =
      BOOLEAN
then TIME
then LIST[sort Linear]
then LIST[sort Path]
then LIST[sort UnitState]
then LIST[sort UnitPathPair]
then PAIR[sort Unit][sort Path]
      with Pair[Unit,Path] ↦ UnitPathPair, pair ↦ unitPathPair,
           first : Pair[Unit,Path] → Unit ↦ getUnit,
           second : Pair[Unit,Path] → Path ↦ getPath
then PAIR[sort Connector][sort Connector]
      with first : Pair[Connector,Connector] → Connector ↦ c1,
           second : Pair[Connector,Connector] → Connector ↦ c2
then sorts  Net, Station, Unit, Connector, UID
      sorts  Linear, Point < Unit
      sorts  Line, Track < List[Linear]
      sort   Path < Pair[Connector,Connector]
      sort   AllStates < List[UnitState]
      sort   Route < List[UnitPathPair]
      sort   UnitState < List[Path]
```

preds $\_\_hasLine\_\_ : Net \times Line;$
$\_\_hasStation\_\_ : Net \times Station;$
$\_\_hasUnit\_\_ : Station \times Unit;$
$\_\_hasTrack\_\_ : Station \times Track;$
$\_\_hasConnector\_\_ : Unit \times Connector;$
$\_\_hasUnit\_\_ : Track \times Unit;$
$\_\_allStates\_\_ : Unit \times List[UnitState];$
$\_\_has\_\_ : Linear \times Connector;$
$\_\_has\_\_ : Point \times Connector$
preds $isAlive : Net;$
$isAlive : Station;$
$isAlive : Unit;$
$isAlive : Connector;$
$isAlive : Line;$
$isAlive : Track;$
$isAlive : Linear;$
$isAlive : Point;$
$isAlive : Path;$
$isAlive : UnitState;$
$isAlive : AllStates;$
$isAlive : Route;$
$isAlive : UnitPathPair;$
$isAlive : List[Linear];$
$isAlive : List[UnitPathPair];$
$isAlive : List[UnitState];$
$isAlive : List[Path];$
$isAlive : Boolean$
pred $\_\_state\_\_At\_\_ : Unit \times UnitState \times Time$
ops $id : Net \to UID;$
$isWellFormed : List[UnitPathPair] \to Boolean;$
$isValidPath : Pair[Connector, Connector] \to Boolean$
ops $isClosedAt : Unit \times Time \to Boolean;$
$isOpen : Route \times Time \to Boolean$
op $path : Connector \times Connector \to ? Path$
$\forall c1, c2 : Connector$
$\bullet\ path(c1, c2) = pair(c1, c2)\ as\ Path$

%(sort_injection_for_pair)%

end

C.2 An Example Track Plan

Below we give an example track plan modelled in Bjørner's DSL.

spec STATIONSTATIC =

```
       BJOERNERSIGNATURE
then free type Net ::= stationNet
     free type Station ::= stationStation
     free type
     Connector
     ::= a1 | a2 | a3 | a4 | a5 | a6 | a7 | a8 | a9 | a10 | a11 | a12
       | a13 | a14 | a15 | b1 | b2 | b3 | b4 | b5 | b6 | b7 | b8 | b9
       | b10 | b11 | b12 | b13 | b14 | b15 | w | x | y | z | p1p3 | p2p4
     ops   lineIn, lineOut : Line
     free type
     Linear
     ::= la1 | la2 | la3 | la4 | la5 | la6 | la7 | la8 | la9 | la10
       | la11 | la12 | la13 | lb1 | lb2 | lb3 | lb4 | lb5 | lb6 | lb7 | lb8
       | lb9 | lb10 | lb11 | lb12 | lb13 | platform1 | platform2
     free type Point ::= p1 | p2 | p3 | p4
     ops   track1, track2, track3, track4, dummyIn, dummyOut
           : Track
     • track1 = la5 :: (la6 :: (la7 :: (la8 :: (la9 :: (la10 :: [])))))
     • track2 = la11 :: (la12 :: (platform1 :: (la13 :: [])))
     • track3 = lb6 :: (lb7 :: (lb8 :: (lb9 :: [])))
     • track4 = lb10 :: (lb11 :: (lb12 :: (platform2 :: (lb13 :: []))))
     • stationStation hasUnit la5
     • stationStation hasUnit la6
     • stationStation hasUnit la7
     • stationStation hasUnit la8
     • stationStation hasUnit la9
     • stationStation hasUnit la10
     • stationStation hasUnit la11
     • stationStation hasUnit la12
     • stationStation hasUnit platform1
     • stationStation hasUnit la13
     • stationStation hasUnit lb6
     • stationStation hasUnit lb7
     • stationStation hasUnit lb8
     • stationStation hasUnit lb9
     • stationStation hasUnit lb10
     • stationStation hasUnit lb11
     • stationStation hasUnit lb12
     • stationStation hasUnit platform2
     • stationStation hasUnit lb13
     • stationStation hasUnit p1
     • stationStation hasUnit p2
     • stationStation hasUnit p3
     • stationStation hasUnit p4
```

- *stationStation hasTrack track1*
- *stationStation hasTrack track2*
- *stationStation hasTrack track3*
- *stationStation hasTrack track4*
- *track1 hasUnit la5*
- *track1 hasUnit la6*
- *track1 hasUnit la7*
- *track1 hasUnit la8*
- *track1 hasUnit la9*
- *track1 hasUnit la10*
- *track2 hasUnit la11*
- *track2 hasUnit la12*
- *track2 hasUnit la13*
- *track3 hasUnit lb6*
- *track3 hasUnit lb7*
- *track3 hasUnit lb8*
- *track3 hasUnit lb9*
- *track4 hasUnit lb10*
- *track4 hasUnit lb11*
- *track4 hasUnit lb12*
- *track4 hasUnit lb13*
- *la1 hasConnector w*
- *la1 hasConnector a1*
- *la2 hasConnector a1*
- *la2 hasConnector a2*
- *la3 hasConnector a2*
- *la3 hasConnector a3*
- *la4 hasConnector a3*
- *la4 hasConnector a4*
- *p1 hasConnector a4*
- *p1 hasConnector a5*
- *p1 hasConnector p1p3*
- *la5 hasConnector a5*
- *la5 hasConnector a6*
- *la6 hasConnector a6*
- *la6 hasConnector a7*
- *la7 hasConnector a7*
- *la7 hasConnector a8*
- *la8 hasConnector a8*
- *la8 hasConnector a9*
- *la9 hasConnector a9*
- *la9 hasConnector a10*
- *la10 hasConnector a10*
- *la10 hasConnector a11*

- *p2 hasConnector a11*
- *p2 hasConnector a12*
- *p2 hasConnector p2p4*
- *la11 hasConnector a12*
- *la11 hasConnector a13*
- *la12 hasConnector a13*
- *la12 hasConnector a14*
- *platform1 hasConnector a14*
- *platform1 hasConnector a15*
- *la13 hasConnector a15*
- *la12 hasConnector x*
- *lb1 hasConnector y*
- *lb1 hasConnector b1*
- *lb2 hasConnector b1*
- *lb2 hasConnector b2*
- *lb3 hasConnector b2*
- *lb3 hasConnector b3*
- *lb4 hasConnector b3*
- *lb4 hasConnector b4*
- *lb5 hasConnector b4*
- *lb5 hasConnector b5*
- *p3 hasConnector b5*
- *p3 hasConnector b6*
- *p3 hasConnector p1p3*
- *lb6 hasConnector b6*
- *lb6 hasConnector b7*
- *lb7 hasConnector b7*
- *lb7 hasConnector b8*
- *lb8 hasConnector b8*
- *lb8 hasConnector b9*
- *lb9 hasConnector b9*
- *lb9 hasConnector b10*
- *p4 hasConnector b10*
- *p4 hasConnector b11*
- *p4 hasConnector p2p4*
- *lb10 hasConnector b11*
- *lb10 hasConnector b12*
- *lb11 hasConnector b12*
- *lb11 hasConnector b13*
- *lb12 hasConnector b13*
- *lb12 hasConnector b14*
- *platform2 hasConnector b14*
- *platform2 hasConnector b15*
- *lb13 hasConnector b15*

• *lb13 hasConnector z*
∀ *p* : *Path*
• *p* = *path*(*w*, *a1*) ∨ *p* = *path*(*a1*, *a2*) ∨ *p* = *path*(*a2*, *a3*)
∨ *p* = *path*(*a3*, *a4*) ∨ *p* = *path*(*a4*, *a5*) ∨ *p* = *path*(*a5*, *a6*)
∨ *p* = *path*(*a6*, *a7*) ∨ *p* = *path*(*a7*, *a8*) ∨ *p* = *path*(*a8*, *a9*)
∨ *p* = *path*(*a9*, *a10*) ∨ *p* = *path*(*a10*, *a11*)
∨ *p* = *path*(*a11*, *a12*) ∨ *p* = *path*(*a12*, *a13*)
∨ *p* = *path*(*a13*, *a14*) ∨ *p* = *path*(*a14*, *a15*) ∨ *p* = *path*(*a15*, *x*)
∨ *p* = *path*(*x*, *a15*) ∨ *p* = *path*(*a15*, *a14*) ∨ *p* = *path*(*a14*, *a13*)
∨ *p* = *path*(*a13*, *a12*) ∨ *p* = *path*(*a12*, *p2p4*)
∨ *p* = *path*(*p2p4*, *b10*) ∨ *p* = *path*(*a4*, *p1p3*)
∨ *p* = *path*(*p1p3*, *b6*) ∨ *p* = *path*(*b6*, *b7*) ∨ *p* = *path*(*b7*, *b8*)
∨ *p* = *path*(*b8*, *b9*) ∨ *p* = *path*(*b9*, *b10*) ∨ *p* = *path*(*b10*, *b11*)
∨ *p* = *path*(*b11*, *b12*) ∨ *p* = *path*(*b12*, *b13*)
∨ *p* = *path*(*b13*, *b14*) ∨ *p* = *path*(*b14*, *b15*) ∨ *p* = *path*(*b15*, *z*)
∨ *p* = *path*(*z*, *b15*) ∨ *p* = *path*(*b15*, *b14*) ∨ *p* = *path*(*b14*, *b13*)
∨ *p* = *path*(*b13*, *b12*) ∨ *p* = *path*(*b12*, *b11*)
∨ *p* = *path*(*b11*, *b10*) ∨ *p* = *path*(*b10*, *b9*) ∨ *p* = *path*(*b9*, *b8*)
∨ *p* = *path*(*b8*, *b7*) ∨ *p* = *path*(*b7*, *b6*) ∨ *p* = *path*(*b6*, *b5*)
∨ *p* = *path*(*b5*, *b4*) ∨ *p* = *path*(*b4*, *b3*) ∨ *p* = *path*(*b3*, *b2*)
∨ *p* = *path*(*b2*, *b1*) ∨ *p* = *path*(*b1*, *y*)
ops *RWX*, *RWZ*, *RXY*, *RZY* : *Route*
∀ *r* : *Route*
• *r* = *RWX* ∨ *r* = *RWZ* ∨ *r* = *RXY* ∨ *r* = *RZY* ∨ *r* = []
• *RWX*

```
= unitPathPair(la1, path(w, a1))
   :: (unitPathPair(la2, path(a1, a2))
      :: (unitPathPair(la3, path(a2, a3))
         :: (unitPathPair(la4, path(a3, a4))
            :: (unitPathPair(p1, path(a4, a5))
               :: (unitPathPair(la5, path(a5, a6))
                  :: (unitPathPair(la6, path(a6, a7))
                     :: (unitPathPair(la7, path(a7, a8))
                        :: (unitPathPair(la8, path(a8, a9))
                           :: (unitPathPair(la9, path(a9, a10))
                              :: (unitPathPair(la10,
                                 path(a10, a11))
                              :: (unitPathPair(p2,
                                 path(a11, a12))
                              :: (unitPathPair(la11,
                                 path(a12,
                                 a13))
                              :: (unitPathPair(la12,
                                 path(a13,
                                                    a14))
                                                 :: (unitPathPair(platform1,
                                                    path(a14,
                                                    a15))
                                                 :: (unitPathPair(la13,
                                                    path(a15,
                                                    x))
                                                 :: []))))))))))))))))
• RWZ
  = unitPathPair(la1, path(w, a1))
     :: (unitPathPair(la2, path(a1, a2))
        :: (unitPathPair(la3, path(a2, a3))
           :: (unitPathPair(la4, path(a3, a4))
              :: (unitPathPair(p1, path(a4, p1p3))
                 :: (unitPathPair(p3, path(p1p3, b6))
                    :: (unitPathPair(lb6, path(b6, b7))
                       :: (unitPathPair(lb7, path(b7, b8))
                          :: (unitPathPair(lb8, path(b8, b9))
                             :: (unitPathPair(lb9, path(b9, b10))
                                :: (unitPathPair(p4, path(b10, b11))
                                   :: (unitPathPair(lb10,
                                      path(b11, b12))
                                   :: (unitPathPair(lb11,
                                      path(b12,
                                      b13))
                                   :: (unitPathPair(lb12,
                                      path(b13,
                                      b14))
                                   :: (unitPathPair(platform2,
                                      path(b14,
                                      b15))
                                   :: (unitPathPair(lb13,
                                      path(b15,
                                      z))
                                   :: []))))))))))))))))
• RXY
  = unitPathPair(la13, path(a15, x))
     :: (unitPathPair(platform1, path(a14, a15))
        :: (unitPathPair(la12, path(a13, a14))
           :: (unitPathPair(la11, path(a12, a13))
              :: (unitPathPair(p2, path(p2p4, a12))
                 :: (unitPathPair(p4, path(b10, p2p4))
                    :: (unitPathPair(lb9, path(b9, b10))
                       :: (unitPathPair(lb8, path(b8, b9))
                        :: (unitPathPair(lb7, path(b7, b8))
                          :: (unitPathPair(lb6, path(b6, b7))
                            :: (unitPathPair(p3, path(b5, b6))
                              :: (unitPathPair(lb5,
                                path(b4, b5))
                              :: (unitPathPair(lb4,
                                path(b3, b4))
                              :: (unitPathPair(lb3,
                                path(b2,
                                b3))
                              :: (unitPathPair(lb2,
                                path(b1,
                                b2))
                              :: (unitPathPair(lb1,
                                path(y,
                                b1))
                              :: []))))))))))))))
• RZY
  = unitPathPair(lb13, path(b15, z))
    :: (unitPathPair(platform2, path(b14, b15))
      :: (unitPathPair(lb12, path(b13, b14))
        :: (unitPathPair(lb11, path(b12, b13))
          :: (unitPathPair(lb10, path(b11, b12))
            :: (unitPathPair(p4, path(b10, b11))
              :: (unitPathPair(lb9, path(b9, b10))
                :: (unitPathPair(lb8, path(b8, b9))
                  :: (unitPathPair(lb7, path(b7, b8))
                    :: (unitPathPair(lb6, path(b6, b7))
                      :: (unitPathPair(p3, path(b5, b6))
                        :: (unitPathPair(lb5,
                          path(b4, b5))
                        :: (unitPathPair(lb4,
                          path(b3, b4))
                        :: (unitPathPair(lb3,
                          path(b2,
                          b3))
                        :: (unitPathPair(lb2,
                          path(b1,
                          b2))
                        :: (unitPathPair(lb1,
                          path(y,
                          b1))
                        :: []))))))))))))))
end
```

Appendix D

Modelling Movement Authorities

```
spec CONTROL =
     COMMONSIGNATURE
then LIST[sort Unit]
then LIST[sort Region]
then sort  Region < List[Unit]
     sort  MA < List[Region]
     pred  assigned : MA × Time
     pred  ext : MA × Route × MA
     op    regions : Route → MA
     pred  canExtend : MA × Time
     pred  canReduce : MA × Time
     pred  clear : Route × Unit
     pred  __isOpenAt__ : Unit × Time

     %% No MA at 0.
     • ∀ m : MA • ¬ m = [] ⇒ ¬ assigned(m, 0)                    %(no_ma_0)%

     %% [] assigned at all times.
     • ∀ t : Time • assigned([] as MA, t)
                                                                 %([]_all_time)%

     %% Extending/reducing a MA.
     • ∀ ma1 : MA; t : Time
       • ¬ ma1 = []
         ⇒ assigned(ma1, suc(t))
            ⇒ (assigned(ma1, t) ∧ ¬ canExtend(ma1, t)
                ∧ ¬ canReduce(ma1, t))
               ∨ (¬ assigned(ma1, t) ∧ canExtend(ma1, t)
                  ∧ ¬ canReduce(ma1, t))
               ∨ (¬ assigned(ma1, t) ∧ ¬ canExtend(ma1, t)
                ∧ canReduce(ma1, t))
                                                        %(assigned_if)%
    ∀ ma1 : MA; t : Time
    • canExtend(ma1, t)
      ⇔ ∃ ma2 : MA; r : Route
          • (assigned(ma2, t)
             ∧ (¬ ma2 = [] ⇒ ¬ assigned(ma2, suc(t)))
             ∧ ext(ma2, r, ma1))
            ∧ r isOpenAt t
                                                        %(extends)%
    • canReduce(ma1, t)
      ⇔ ∃ rg : Region
          • assigned((rg :: ma1) as MA, t)
            ∧ ¬ assigned((rg :: ma1) as MA, suc(t))
                                                        %(reduces)%

    %% Only one MA changes at any time.
    • ∀ t : Time
      • ∀ m1, m2 : MA
        • (assigned(m1, suc(t)) ⇒ canExtend(m1, t))
          ∧ (assigned(m2, suc(t)) ⇒ canExtend(m2, t))
          ⇒ m1 = m2
                                                        %(one_changes)%

    %% Unit not open when.
    • ∀ r : Route; rg : Region; ma : MA
      • rg eps regions(r) ∧ rg eps ma ∧ assigned(ma, t)
        ⇒ ∃ u : Unit; upp : UnitPathPair
            • u eps rg ∧ getUnit(upp) = u ∧ upp eps r
              ∧ ¬ u isOpenAt t
                                                        %(unit_in_use)%
    %% Ext acts like ++
    • ∀ ma1, ma2 : MA; r : Route
      • ext(ma1, r, ma2) ⇒ ma2 = ma1 ++ regions(r)
                                                        %(ext_defn)%
end

spec ControlTable =
    Control
then ∀ r : Route; t : Time
    • r isOpenAt t ⇔ ∀ u : Unit • clear(r, u) ⇒ u isOpenAt t
                                                        %(Route_open_defn)%
end
```

Appendix E

DSL Lemmas in CASL

```
spec PROOF =
     SHARE
then • ∀ r : Route; t : Time; rg : Region; ma : MA
        • assigned(ma, t) ∧ rg eps ma ∧ rg eps regions(r)
          ⇒ ¬ r isOpenAt t
                                                              %(toProveForModel)%
then %implies
      %% Base case
      • ∀ m1, m2, ma1, ma2 : MA
        • share(ma1, ma2)
          ⇒ ma1 = ma2 ∨ ¬ (assigned(ma1, 0) ∧ assigned(ma2, 0))
                                                                     %(base_case)%

      %% step case
      • ∀ ma1, ma2 : MA; t : Time
        • share(ma1, ma2) ∧ assigned(ma1, suc(t))
          ∧ assigned(ma2, suc(t))
          ⇒ (assigned(ma1, t) ∧ assigned(ma2, t))
             ∨ (canExtend(ma1, t) ∧ canExtend(ma2, t))
             ∨ (assigned(ma1, t) ∧ canExtend(ma2, t))
             ∨ (assigned(ma2, t) ∧ canExtend(ma1, t))
             ∨ (canReduce(ma1, t) ∧ canReduce(ma2, t))
             ∨ (canReduce(ma1, t) ∧ canExtend(ma2, t))
             ∨ (canReduce(ma2, t) ∧ canExtend(ma1, t))
             ∨ (assigned(ma1, t) ∧ canReduce(ma2, t))
             ∨ (canReduce(ma1, t) ∧ assigned(ma2, t))
                                                                 %(case_analysis)%
      ∀ t : Time
      • (∀ ma1, ma2 : MA
         • share(ma1, ma2)
         ⇒ ma1 = ma2 ∨ ¬ (assigned(ma1, t) ∧ assigned(ma2, t)))
     ⇒ ∀ ma1, ma2 : MA
        • share(ma1, ma2)
          ∧ (assigned(ma1, t) ∧ assigned(ma2, t))
          ⇒ ma1 = ma2
             ∨ ¬ (assigned(ma1, suc(t)) ∧ assigned(ma2, suc(t)))
                                                              %(case1)%
  ∀ t : Time
  • (∀ ma1, ma2 : MA
      • share(ma1, ma2)
         ⇒ ma1 = ma2 ∨ ¬ (assigned(ma1, t) ∧ assigned(ma2, t)))
     ⇒ ∀ ma1, ma2 : MA
        • share(ma1, ma2) ∧ (canExtend(ma1, t) ∧ canExtend(ma2, t))
          ⇒ ma1 = ma2
             ∨ ¬ (assigned(ma1, suc(t)) ∧ assigned(ma2, suc(t)))
                                                              %(case2)%
  ∀ t : Time
  • (∀ ma1, ma2 : MA
      • share(ma1, ma2)
         ⇒ ma1 = ma2 ∨ ¬ (assigned(ma1, t) ∧ assigned(ma2, t)))
     ⇒ ∀ ma1, ma2 : MA
        • share(ma1, ma2)
          ∧ (assigned(ma1, t) ∧ canExtend(ma2, t))
          ⇒ ma1 = ma2
             ∨ ¬ (assigned(ma1, suc(t)) ∧ assigned(ma2, suc(t)))
                                                              %(case3)%
  ∀ t : Time
  • (∀ ma1, ma2 : MA
      • share(ma1, ma2)
         ⇒ ma1 = ma2 ∨ ¬ (assigned(ma1, t) ∧ assigned(ma2, t)))
     ⇒ ∀ ma1, ma2 : MA
        • share(ma1, ma2)
          ∧ (canExtend(ma1, t) ∧ assigned(ma2, t))
          ⇒ ma1 = ma2
             ∨ ¬ (assigned(ma1, suc(t)) ∧ assigned(ma2, suc(t)))
                                                              %(case4)%
  ∀ t : Time
  • (∀ ma1, ma2 : MA
      • share(ma1, ma2)
         ⇒ ma1 = ma2 ∨ ¬ (assigned(ma1, t) ∧ assigned(ma2, t)))
     ⇒ ∀ ma1, ma2 : MA
        • share(ma1, ma2) ∧ (canReduce(ma1, t) ∧ canReduce(ma2, t))
          ⇒ ma1 = ma2
             ∨ ¬ (assigned(ma1, suc(t)) ∧ assigned(ma2, suc(t)))
```

```
                                                              %(case5)%
    ∀ t : Time
    • (∀ ma1, ma2 : MA
      • share(ma1, ma2)
        ⇒ ma1 = ma2 ∨ ¬ (assigned(ma1, t) ∧ assigned(ma2, t)))
      ⇒ ∀ ma1, ma2 : MA
        • share(ma1, ma2) ∧ (canReduce(ma1, t) ∧ canExtend(ma2, t))
          ⇒ ma1 = ma2
            ∨ ¬ (assigned(ma1, suc(t)) ∧ assigned(ma2, suc(t)))
                                                              %(case6)%
    ∀ t : Time
    • (∀ ma1, ma2 : MA
      • share(ma1, ma2)
        ⇒ ma1 = ma2 ∨ ¬ (assigned(ma1, t) ∧ assigned(ma2, t)))
      ⇒ ∀ ma1, ma2 : MA
        • share(ma1, ma2) ∧ (canReduce(ma2, t) ∧ canExtend(ma1, t))
          ⇒ ma1 = ma2
            ∨ ¬ (assigned(ma1, suc(t)) ∧ assigned(ma2, suc(t)))
                                                              %(case7)%
    ∀ t : Time
    • (∀ ma1, ma2 : MA
      • share(ma1, ma2)
        ⇒ ma1 = ma2 ∨ ¬ (assigned(ma1, t) ∧ assigned(ma2, t)))
      ⇒ ∀ ma1, ma2 : MA
        • share(ma1, ma2)
          ∧ (canReduce(ma2, t) ∧ assigned(ma1, t))
          ⇒ ma1 = ma2
            ∨ ¬ (assigned(ma1, suc(t)) ∧ assigned(ma2, suc(t)))
                                                              %(case8)%
    ∀ t : Time
    • (∀ ma1, ma2 : MA
      • share(ma1, ma2)
        ⇒ ma1 = ma2 ∨ ¬ (assigned(ma1, t) ∧ assigned(ma2, t)))
      ⇒ ∀ ma1, ma2 : MA
        • share(ma1, ma2)
          ∧ (canReduce(ma1, t) ∧ assigned(ma2, t))
          ⇒ ma1 = ma2
            ∨ ¬ (assigned(ma1, suc(t)) ∧ assigned(ma2, suc(t)))
                                                              %(case9)%
then %implies
    ∀ t : Time
    • (∀ ma1, ma2 : MA
      • share(ma1, ma2)
        ⇒ ma1 = ma2 ∨ ¬ (assigned(ma1, t) ∧ assigned(ma2, t)))
    ⇒ ∀ $ma1, ma2 : MA$
       • $share(ma1, ma2)$
          ⇒ $ma1 = ma2$
             ∨ ¬ $(assigned(ma1, suc(t)) \wedge assigned(ma2, suc(t)))$
                                                                    %(lemma)%
end
```