Formal Specification of Moving Block Railway Interlocking using CASL

Andrew Ryan
439600

May 5, 2010

Abstract

The purpose of this project is to investigate the usability of the algebraic specification language CASL (Common Algebraic Specification Language) for the specification of industrial projects, in this case the moving block railway interlocking system. Included in this document will be an overview of the evolution of railway interlocking, looking ahead to future technology such as the moving block system. It will also outline the place of system specification within the development of such systems and within the realm of software engineering as a whole and will also compare arguments for an against the use of system specification in the development of industrial systems.

Project Dissertation submitted to the University of Wales, Swansea
in Partial Fulfillment for the Degree of Bachelor of Science

Swansea University
Prifysgol Abertawe

Department of Computer Science
University of Wales Swansea

Declaration

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

May 5, 2010

Signed:

Statement 1

This dissertation is being submitted in partial fulfillment of the requirements for the degree of a BSc in Computer Science.

May 5, 2010

Signed:

Statement 2

This dissertation is the result of my own independent work/investigation, except where otherwise stated. Other sources are specifically acknowledged by clear cross referencing to author, work, and pages using the bibliography/references. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure of this dissertation and the degree examination as a whole.

May 5, 2010

Signed:

Statement 3

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

May 5, 2010

Signed:

Contents

List of Figures

1 Introduction

COMPUTER systems are increasingly becoming apart of every aspect of modern life. What started as computational machines for carrying out complex arithmetic now exist in almost every home, business and school. Not only do we use computers to manage our social lives, our finances and our music, we also trust our lives to computers when they fly planes, control traffic systems and in a more extreme example control our weapons. In order to have trust in the reliability and safety of such systems we need to be able to prove that a system has been rigorously designed with little to nothing being left to chance.

This is the main motivation behind the development of formal specification, implementation and verification, which are combined to form an area of software engineering known as formal methods. The purpose of formal methods is to add confidence to the safety and reliability of systems where safety and correct operation are a major concern including safety critical systems such are railway interlocking and flight control, and business critical systems such as online banking and security software.

While the term formal methods may be thought to be a system of methodologies and design stages similar to other software engineering methodologies, it instead defines a system where the design and verification of systems using well formed mathematical notation with formally defined semantics in order to both design the system in the form of its specification, and test its implementation through the use of automatic verification tools. Formal methods are not designed to replace more traditional software engineering stages of design implementation and test, but are instead meant to compliment and enhance the reliability of the outputs of these stages.

The main purpose of this project was to investigate the usability of the algebraic specification language CASL in the specification of an industrial based project. In order to investigate this an attempt was made the develop a formal specification for a railway interlocking system in the chosen language CASL. CASL is a relatively new language into the area of formal specification, and has been designed to succeed the various other specification languages in order to form a standard language from specification of systems. In order to do this, CASL can easily support new functionality in the form of extensions. Support and tools of Hets (the heterogeneous toolset) which is able to parse and run static analysis on CASL specifications as well as specifications written in many extensions of CASL.

The system chosen for this task was the moving block system for railway interlocking. This system is still largely in a conceptual stage and is currently only used on light railway systems such as the London Docklands railway [RT09]. The decision was made to attempt the specification in CASL as it is a relatively new language in the system specification. While there are two books available detailing the use of CASL, the User Manual [BM04b] and The CASL Reference Manual [BM04a], little other documentation exists. Many of the exercises carried out using CASL (as well as other specification languages) are often referred to as "toy examples".

That is to say they are often of very small example systems which are often of little use to industry. While the specification carried out for this project is only of a relatively small part of the interlocking system, it has been developed with future iterations in mind. It has been formed with the intention that it can be used, wholly or partially, for future specifications of the same, similar or expanded system.

We begin in section 2 by explaining how railway traffic control systems have evolved from their inception to become one of the major examples of critical systems in use every day, as well as covering the future of traffic management systems in the form of the European Railway Traffic Management System (ERTMS)

In section 3 we look into the importance of formal methods in software development and their place, and uptake in industry and obstacles preventing widespread use of formal methods in software design.

In section 4 we go into some detail about the Common Algebraic Specification Language including its origins and purpose, the structure of a CASL specification, and the support and tools available for development and analysis of specifications.

In section 5 we look at steps required and considerations to be taken before commencing the specification of the moving block interlocking system such as informal requirements documents and how to model an abstract notion of time.

In section 6 we look closely at the specification of moving block interlocking that has been developed for the purpose of this project. Where possible we make use of automatic theorem provers (in this case SPASS) in order to validate the specification against the requirements. Where this has not been possible we instead attempt to validate the specification manually. This method requires more intuition about the system than the use of an automatic theorem prover since we are comparing a formal specification against an informal document.

2 Railway Traffic management Systems

RAILWAY traffic management systems are systems which are put in place on railway lines in order to efficiently and safely manage the flow of trains. First developed using only guardsmen and later developed to incooperate mechanical and then digital interlocking over wide areas the systems have been effective in increasing the safety and reliability of railway systems. However the rail industry continues to invests large sums of money into the development of new, safer and more efficient systems to further increase the capacity of the railway while maintaining a high level of safety for its users.

The system for which specification has been attempted in this project is know as the moving block interlocking system, a largely conceptual system which requires clear defining if it is to be understood and modeled. Before this however it is necessary to understand the current state of railway interlocking technology. Since the development of said systems have been based around evolution not revolution it is important to understand how railway interlocking has developed from its original conception.

2.1 The History and Evolution of Railway Safety Systems

The exact origins of the modern railway system is difficult to pinpoint exactly. During the industrial revolution in Britain in the 19th century many privately owned collieries had their own tram lines and wagon ways connecting coal mines to iron factories and docks all around the country. It was industrialist William James (1771-1837) who first suggested a national interconnected network of railway lines, however he became bankrupt and was taken over by George Stephenson before his vision of a national rail network could become reality [Mac07]. The concept of an interconnected rail network, which relatively easy in theory would be difficult to manage since different rail networks used varying gauges (track widths) and rail types. It wasn't until the advent of steam locomotion in 1814 [Var10], wrought iron tracks, and the introduction of the 4 feet 8 1/2 inch (1435mm) gauge that the railway became a viable option for the mass transit of both heavy goods and the general population [Var09].

By 1914 over 20,000 miles of track had been put in place in the UK [Com09] serving the general public with services such as passenger and mail routes, as well as heavy goods transportation for primary and secondary industry.

Typical to all technological advancements, and particularly as the railway became more commonplace as a form of transport for the public, questions began to be asked with regards to the safety of the system [Har99]. As the popularity of the railway system increased, greater numbers of trains were in operation and the number of incidents of trains colliding with one another increased. Accidents such as the Tylwch train crash in Powys in 1988 [Pro09] (see figure 2.1) and the Norwich disaster in 1874 were becoming increasingly common. In the Norwich disaster two trains crashed head on after a proceed order was given to a train that would normally have been safe to proceed however, due to delays the trains collided. In this crash human error was to blame however the Board of Trade inspector (the board at the time in charge of investigating industrial accidents) criticised the "laxity of the system that allowed such mistakes to occur" [Lee10]. Crashes such as these garnered a great deal of public attention due to the media exposure they received and while industrial accidents at sea, in mines and in factories were common they were not directly in the public eye, while railway accidents were widely reported on.

Figure 2.1: Image of train crash at Tylwch, Powys [Pro09]

The key obstacle faced with railway safety is that due to the weight of the engine and carriages and the speed at which the trains moved meant that breaking in time from seeing a hazard ahead is very difficult, made even more so by trees and embankments which may obscure vision of the track ahead. Even with modern breaking technology the British Rail Intercity-125 is designed to have a stopping distance of approximately 1.08 miles from full speed [Bar92].

Along a busy railway line the most obvious obstacle ahead is the train in front and so it quickly became apparent that some form of traffic management would be required in order to maintain both reliable running of the timetable and ensure safe operation of the railway lines. In the earliest example of traffic management guards would stand at intervals along a length of track equipped with a stop watch. When a train passed the guard would start the stop watch, and by means of hand or flag signals, not allow the next train to pass until a set amount of time had passed. This is the earliest example of a fixed block traffic management system from which the current railway interlocking system has been developed and was called "time interval working", with each guard protecting a "block" of track.

One of the major flaws with this system was that while the guard knew how much time had passed since the last train had entered his block, he was not aware of whether the train had successfully exited the block the other end, as blocks were often to long for the guard to observe the whole block. Such a flaw was the cause of the Armagh rail disaster in 1989 [Ray09, p. 2] where the failure of a leading train to ascend a steep gradient caused it to begin to roll back along the track and was hit by a following train. This accident in particular had a profound effect on the British railway industry. Due to the loss of life of 78 people, 22 of which were under 16 [Ray09, p.2] the British government introduced the Regulation of Railways Act of 1889 which instated the law :

(1) The Board of Trade may from time to time order a railway company to do, within a time limited by the order, and subject to any exceptions or modifications allowed by the order, any of the following things:-
(a) To adopt the block system on all or any of their railways open for the public conveyance of passengers;
(b) To provide for the interlocking of points and signals on or in connection with all or any of such railways;
⋮[Gov89]

This meant that all companies operating a public railway line could be forced to adopt the more modern block system and interlocking of its tracks and points if it was deemed necessary by the Board of Trade.

The block system and interlocking system referred to in [Gov89] was a more modern method of traffic management system and came about due to the invention of telegraph communication. This long range communication method meant that a signalman based in a signal box at the end of the block was able to signal to the guardsman that the train had successfully proceeded out of the block and it was therefore safe for the next train to proceed.

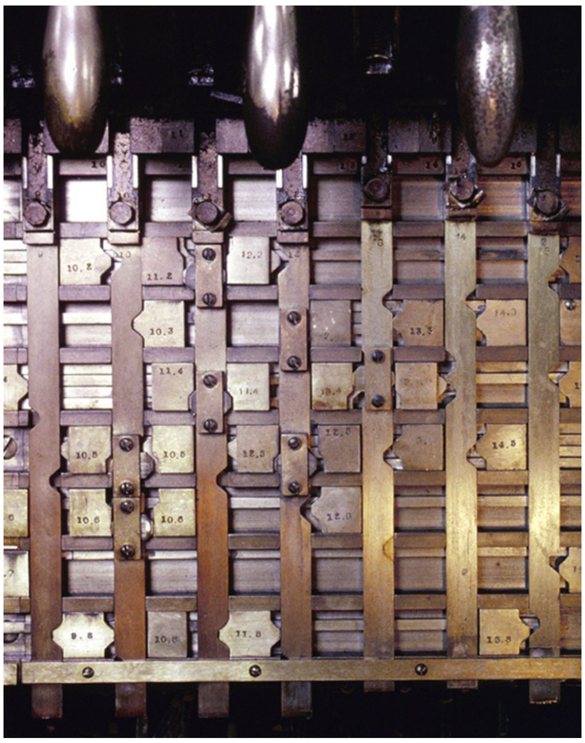


Figure 2.2: A mechanical interlocking bed, source: [Rom94]

Messages passed between between signal boxes via telegraph were translated into bell rings. certain combinations of bell rings would indicate to the signal man that it was safe for the next train to proceed. Human error included mis interpretation bell rings and points being set into the wrong positions, causing misdirection of trains or derailment.

1841 saw the first semaphores installed along UK railway lines, originally operated locally semaphores consisted of an arm mounted onto a large pole for increased viability. The position of the arm, indicated to the driver of the oncoming train whether or not it was safe to proceed.

In an attempt to eliminate human error from the system the mechanical interlocking system was developed. Signaling boxes were built along sections of railway where signalmen could remotely operate semaphores and points by means of levers. In order to remove the possibility of unsafe signaling to oncoming trains leaver boxers were constructed to be physically unable to be set into unsafe positions by means of rods and pins which would restrict the movement of the levers [Pag09] (see figure 2.2.

Human error was still a key cause of accidents and so further measures were taken to attempt to further automate the process leading to the invention of the track circuit. This system used a low voltage electrical current passed through the tracks, when a train was present on the track the metal axles would short the track circuit causing a relay switch to flip. This would then cause a changing of lights on a signal board in the signal box to show the signalman when a track as clear or not. Later the track circuit was directly linked to the signal of the block it detected, meaning that the signal would automatically show an oncoming train to stop. This is also an early example of a fail safe system with other problems on the track automatically setting the corresponding signal to red. Although track circuits were first trialled in America as far back as 1890 [Pag09] they still play a vital role in modern interlocking systems for train detection with electrical relays now being succeeded by solid state interlocking and WRSL's Westlock ® system.

Originally signals only used two "aspects", stop and proceed, represented by red and green respectively with the introduction of light signals. As train speeds increased and breaking distances lengthened it often became difficult for trains to stop in time once a stop signal is seen, especially where signals were located on bends or obstructed by vegetation. This lead to the usage of multi-aspect signals. By using a third and later fourth aspect, represented by one yellow and two yellow lights in the UK, it is possible to warn drivers that the next signal was stop and that the train driver should begin breaking. Where multi-aspect signals are not required, particularly on sections of track with long blocks, distance signals are used. These signals could either show caution (yellow) to inform the driver the next signal was stop or that he should reduce speed for safety reasons, or continue (green) to inform the driver to continue at maximum allowed speed. These distance signals are placed at a distance from the following signal according to speed restrictions on the track and breaking distance of trains in operation on the track to allow the driver enough time to stop the train before the next signal.

Further advancements to railway safety systems focused mainly on the transfer from mechanical to digital systems and reducing the chance for human error, such as automatic train protections systems which would automatically stop the train if it attempted to pass a stop signal. A decrease in moving parts and the ability to centralize signal management to larger signal boxes that could service an entire region brought down maintenance requirements and reduced costs of railway safety. Railway safety systems are now a highly competitive industry in the field of critical systems with major cooperations such as Invensys Rail group and Siemens competing for contracts all across the country.

2.2 Fixed Block Interlocking

As mentioned previously fixed block interlocking relies on the track being separated into blocks of a fixed length. The operation of fixed block interlocking, with the use of multi-aspect signals is shown in figure 2.3

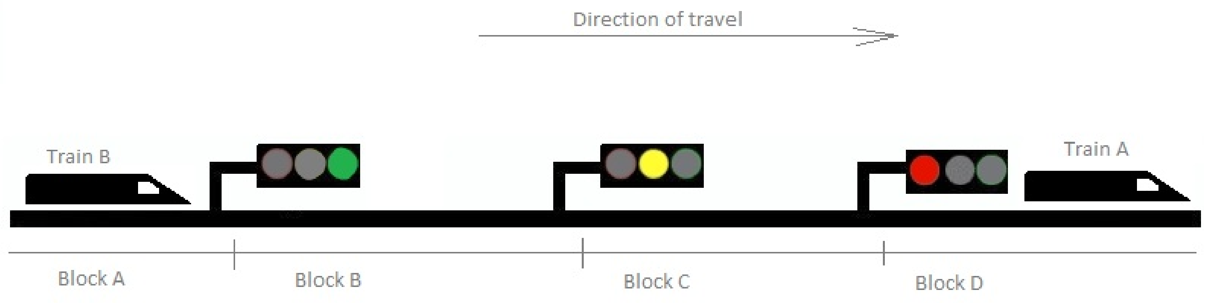


Figure 2.3: with one train following another

In this diagram the signal corresponding to Block D is set to stop (red) due to the presence of Train A. As this uses multi-aspect signals the preceding signal controlling Block C is set to caution (yellow) indicating to the next train that it should begin breaking ready to stop. The preceding signal, for Block B is set to green to show that it is safe to proceed.

Figure 2.4 shows the action of interlocking on a point. In this example the line occupied by Train B is a single track line, meaning if Train A was allowed to proceed then it would result in derailment because of the position of the point, and if the point was in the correct position then a head on collision would occur with Train B. Therefore the signal granting access to the point from the direction of Train A is set to red despite its next block being free.

This system does have drawbacks with regards to the efficiency of the railway system. Take for example the situation where Train B above has entered Block C when the corresponding signal was yellow and so started to break, however by the time this train reaches the next signal Train A may have exited block D and so Train B would see another yellow signal and so would continue at a minimal speed. This in turn results a greatly decreased capacity of the railway system reducing profits of train operators, particularly on busy lines. As a result development of railway safety systems has become increasingly concerned with increasing the capacity of railway lines, while maintaining the same level of safety.

2.3 The ERTMS and Future Railway Interlocking Technology

The European Railway Traffic Management System (ERTMS)is a system in development with the aim of standardising Europe's railway signaling systems. Currently there are 20 different

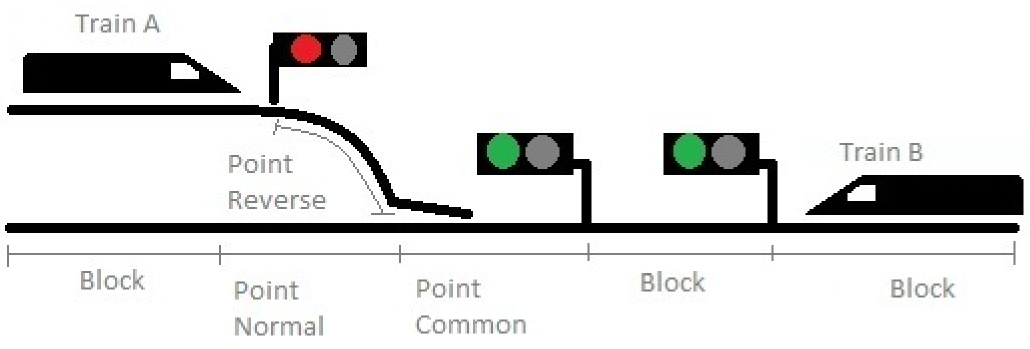


Figure 2.4: Interlocking operation at a point

train control systems across the EU [UNI09c]. This poses a major problem to making railway travel the main source of transport as it means trains which traverse through various countries have to be equipped with the correct hardware and intercontinental train drivers must also be trained in the various different signaling conventions. In order to improve inter-operability between different region's train systems, in 1998/99 the EU commissioned the founding of UNISIG, an industrial consortium consisting of six major suppliers of railway traffic management systems including Invensys Rail [UNI09b] who's remit was to develop and maintain a European standard traffic management system in cooperation with the European Railway agency. While the ERTMS was developed as a European standard it has also been adopted by other railway companies in Asia, North Africa and central America [UNI09c].

The resultant system was the ERTMS, which not only provides a standard railway signaling method throughout Europe but also takes steps to increase the efficiency railway systems. The ERTMS consists of two main components:

- ETCS: The European Train Control System, acts as an on board train protection system. [UNI09c]
- GSM-R: is a radio system build upon the GSM network originally developed for mobile phones and operates at a frequency specific to the railway system. It provides data and voice communication between the train based ETCS and the track side traffic management system.

In order to better facilitate the transition to ERTMS from existing traffic management systems UNISIG has developed three different levels of ERTMS which can be added on top of existing systems to allow for gradual implementation.

2.3.1 ERTMS Level 1

Level one of the ERTMS is designed to overlay the existing signaling system and make use of hardware already installed such as the line side signals to grant movement authorities and track circuits for train detection and track integrity checks. Communication between the track side management and the train is handled by "Eurobalise®" located at the track adjacent to line side signals and at required intervals. At this level the balise are linked to the track side management system and so the information contained within them can be changed dynamically, often to inform the driver of the next signal in advance and to inform the driver of track details such as gradient and speed restrictions. The ETCS on board the train is then able to calculate optimal speed of the train based on the next signal and the optimal breaking point based on the breaking statistics of the train and the information relayed by the balise. The ETCS is also able to implement automatic breaking measures should the train exceed speed limits or movement authorities. [UNI09a]

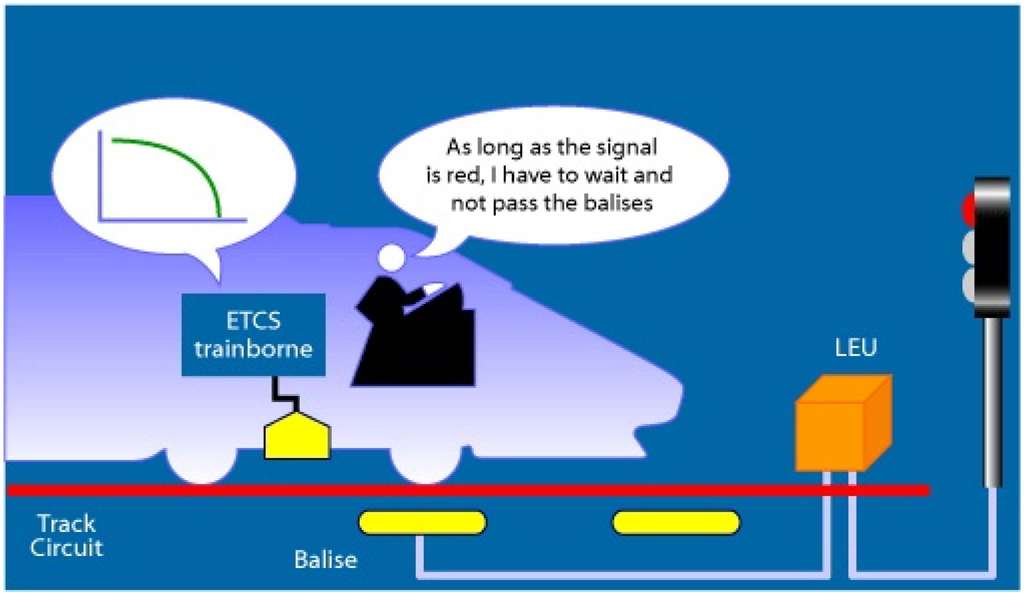
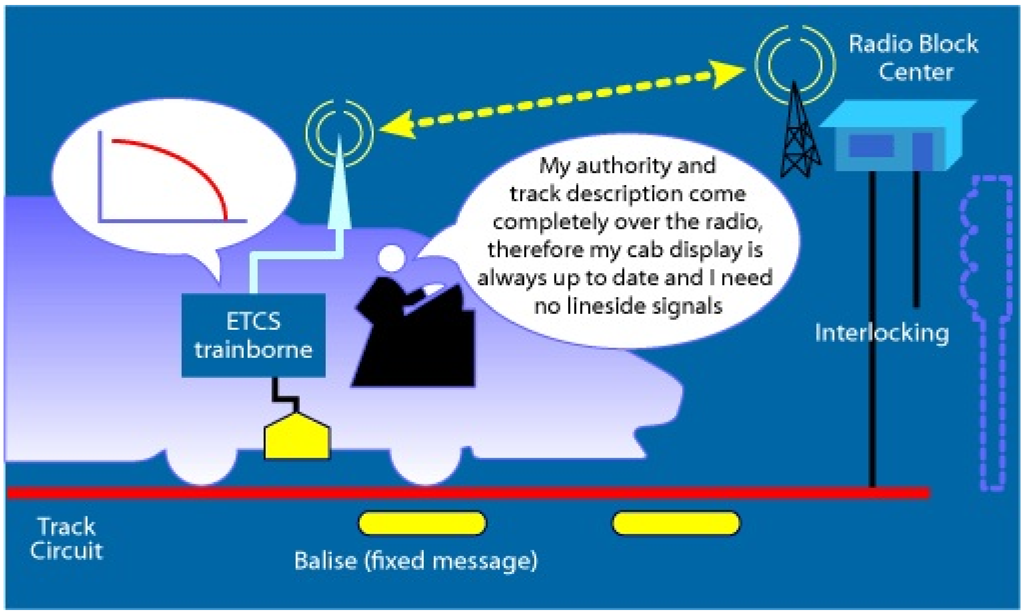

As long as the signal is red, I have to wait and not pass the balises
ETCS trainborne
LEU
Track Circuit
Balise

Figure 2.5: ERTMS Level 1 source: [UNI09a]

2.3.2 ERTMS Level 2

Level 2 of the ERTMS is the first level which implements the GSM-R. Rather than using line side signals movement authorities are updated via the GSM-R and the signal is relayed to the Eurocab system (see figure 2.6b), which is a specialist screen installed in the cab of the train where a real time stream of data is displayed to show the length of the current movement authority and the optimal speed at which to proceed. Track information is now updated via the GSM-R and so Eurobalises are only used to relay positional information to the ETCS on board the train and contains constant data.

While this level maintains use of the fixed block interlocking system it does add certain advantages. The first that because of constant updates to the trains Eurocab display, the driver is constantly receiving the most up to date information and so can drive at an optimal speed thus improving time and energy proficiency of the train. The second advantage is that the Eurocab system replaces line side signals meaning track side maintenance costs are reduced. However as Railway operation companies implement ERTMS level 2 it is likely that line side signals will still be used for sometime to allow trains which are not ERTMS Level 2 equipped to operate on the tracks. [UNI09a]


Radio Block Center
My authority and track description come completely over the radio, therefore my cab display is always up to date and I need no lineside signals
ETCS trainborne
Interlocking
Track Circuit
Balise (fixed message)

(a) ERTMS Level 2 Interactions

(b) Eurocab screen

Figure 2.6: ERTMS Level 2 source: [UNI09a]

2.3.3 ERMTS Level 3

The highest level of ERTMS, level 3, is still in the conceptual stage of development and is not yet been implemented by the industry because of its lack of maturity. It is the only level to not be based on fixed block interlocking. Instead using moving block interlocking. This again is still conceptual and only in use on light railway systems such as the previously mentioned London Docklands Railway [RT09]. Rather than using fixed blocks, the moving block system creates a block dynamically around each train. This block can be expanded or contracted based on various conditions such as track gradient, speed restrictions, breaking characteristics of the train and length of the train. The block has a large section ahead of the train and a smaller block behind the train to account for discrepancies or slight errors in train positioning which is now solely handled by the on board ETCS, as is train integrity [UNI09a]. Certain aspects of the fixed block system are required by the moving block system, such as stopping access to a single line track which is occupied by an oncoming train. In this case the full length of the single like track is treated as occupied to all trains which are coming from the opposite direction of the occupying train, where as the moving block rules apply to all trains following this one. This form of movement authorization however is done during route setting (called reserving the line) and train timetables are normally formulated to minimize cases such as this.

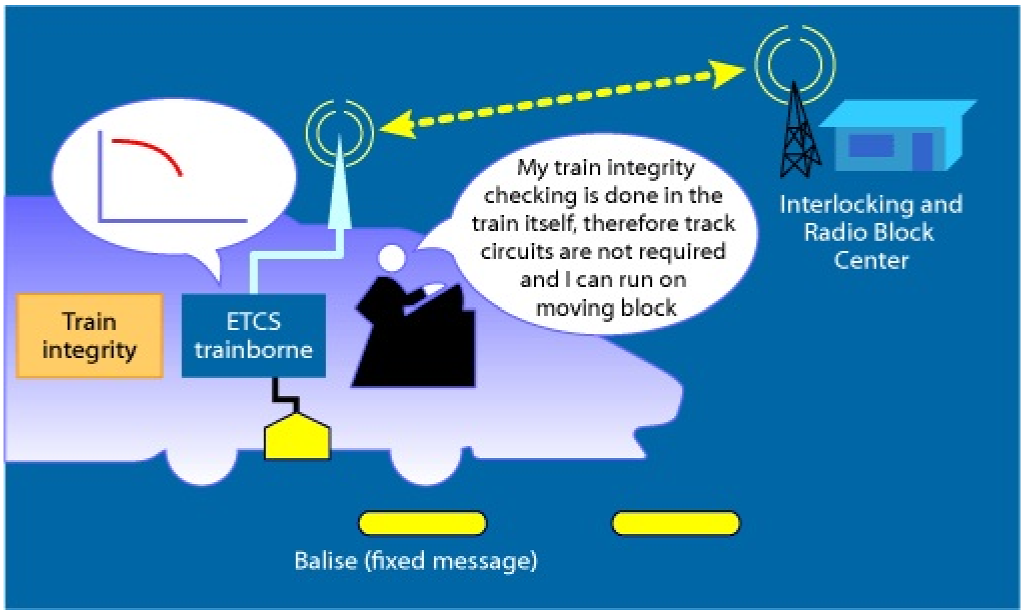
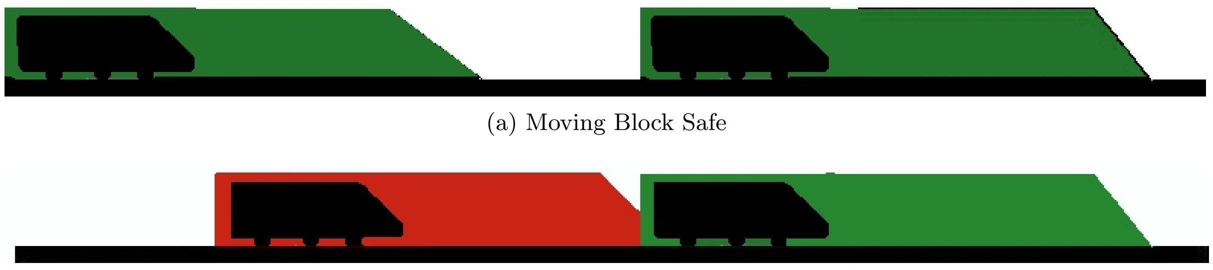

Train integrity
ETCS trainborne
My train integrity checking is done in the train itself, therefore track circuits are not required and I can run on moving block
Interlocking and Radio Block Center
Balise (fixed message)

Figure 2.7: ERTMS Level 3 source: [UNI09a]

The use of the moving block system creates several technical challenges which has so far stopped it from becoming widely used. With current technology the on board track integrity and positioning systems are not yet able to report to the control system with sufficient accuracy to be reliable. Another challenge is hardware requirements. As the system does not use predefined blocks, the control system is required to be constantly updated with the positions of all trains (known as the state of the railway) and calculate the position of blocks for all trains. The system operates as a batch system, taking a snapshot of the state of the railway, calculating block positions and evaluating which signals to send to each train for every time interval. This time interval is a fraction of a second giving the effect of a hybrid batch/real-time system.

Figure 2.8a and 2.8b represents the action of the moving block system on two trains. The green area surrounding the trains in 2.8a represents the moving block around each train, the green colour is used to show that the blocks are clear. In figure 2.8b it can be seen that the block of the rear train overlaps with the block of the front train, and so the rear train is given the signal to decrease speed (represented by the block being coloured red).


(a) Moving Block Safe

(b) Moving Block Dangerous

Figure 2.8: Representation of the moving block system

ERTMS level 3 is the most efficient of the three ERTMS levels. The moving block systems ensures that only the area in front of the train equivalent to the breaking distance at the given time must be clear, therefor meaning that minimal space is wasted, increasing the capacity of the line. Also, because of the constant updating of optimal speeds based on the preceding train's speed, as well as optimal breaking points, the energy efficiency of the trains is improved due to smooth running of the engine.

It is the moving block system for which specification has been attempted in this project. Much of the complexity of ERTMS such as automatic train protection and train integrity has not been included within the specification in order to make the specification more manageable.

3 Formal Methods

In the early days of digital computing technology computers were developed with the aim of carrying out complex mathematical arithmetic. Due to the specific requirements of each computer system, programs could be verified correct by reproducing the results manually following the functions laid out in the computer program. As computer systems became more complex and powerful, traditional methods of software verification through test cases and scenarios became increasingly difficult as the number of possible test cases became prohibitively large [BBFM82]. This large number of possible test cases means only a subset of possible test cases can be run, making testing strategies somewhat ineffective in proving the whole of a system correct. While this is still deemed acceptable by many sectors of the software engineering industry it is increasingly being shown to be inefficient in testing software in the domain of critical system such as railway interlocking systems. Within this domain it is vital that developers are able to prove that their systems meet the requirements of the customer or end user for all possible scenarios. This gives the developer assurance that if an accident does occur he/she can give evidence that the system is sufficient to requirements, otherwise there may be financial ramifications such as compensation claims as well as loss of business. This level of assurance also allows the customer to trust in the system where customer or financial safety is of high importance. The issue of inadequate testing is also known as the verification problem *"How do we show that a piece of software is correct"*

This increase in computational complexity lead to the "software crisis" [RN69] experienced in the second half of the 20th century. During this period the IT industry came under increasing scrutiny as major IT projects ran over-budget, over-time, and final products did not meet requirements and in many instances was not delivered at all [DT96]. The problem is summed up by Edsger Dijkstra in his paper "The Humble Programmer" (see Quote 3.1).

> *The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.*

Quote 3.1: Edsger Dijkstra, The Humble Programmer, Communications of the ACM [W.D72]

The response to this requirement was the development of formal methods. Formal methods was developed in order to formalize the software engineering stages of design, implementation and testing through the use of formal specification and verification. Rather than referring to a methodology, formal methods is the use of mathematical techniques for the specification and proving of software and hardware systems [Sch89]. Formal methods then, can be used in conjunction with other software engineering methodologies such as the waterfall or spiral method of software development.

Through using well formed mathematical notation with clearly defined semantics it is possible to remove all ambiguity of a system design and means that a system can be designed and built such that the behavior of a system can be accurately and precisely predicted [Bjø06]. This counteracts one of the key weaknesses of more traditional informal methods of system design which is the ambiguity of natural language. While an end user may be sure that the requirements it lays down mean one thing, differences in vocabulary between engineering disciplines, or domain specific jargon may cause the software engineer to interpret the requirements differently. By requiring the system designer to think logically about the system design, any ambiguities in requirements must be addressed at the design stage, forcing a dialog between the designer and the customer. Normally issues such as this would not become apparent until testing the implementation of the system and at this late stage bugs and errors in the system would have been more difficult to locate meaning more time, and so money, was spent on correcting errors and removing bugs which might otherwise have not been present if formal specification were used at the design stage.

Systems designed and implemented using formal methods are thus able to be verified correct by further use of mathematical tools and techniques. As systems are specified using well defined logical syntax with proven semantics it is possible to check that an implementation fulfills the specification through the use of formal verification methods. The result of using verification, whether done manually or through an automatic verification tool, is a *proof*. This is a demonstration that the program carried out the operations as laid down by the specification. While based solely on the specification and program text (no execution of the program is required) these proofs can provide a greater confidence that the use of test cases as these proofs consider all possible cases, not merely a subset of possible cases [Jac97]. The overall effect is the development of a system which is less error prone and of a superior quality code [Sch89].

3.1 Formal Methods in Industry

Despite the obvious advantages of using formal methods for the development of systems, it's use is still primarily based in labs and academia rather than on industrial projects and the usefulness of formal methods on large scale industrial problems is a heavily debated issue.

While advantages of formal methods have already been outlined in this section, using formal methods is not without its disadvantages the most obvious of which is time consumption. While using formal methods produces a final product of higher quality, this comes at the cost of time, especially at the design stage which can become considerabley extended when attempting formal specification. Proponents of formal methods would argue that this extra time spent at the design stage is offset by time savings in implementation and testing. Project managers however often aim for milestones in terms of "deliverables" usually in the form of a prototype product or at the very least some code text. This rush to begin implementation is a residual effect of software engineering still being in its infancy and is a mindset which formal methods is designed to combat.

A further disadvantage to the use of formal methods, which may not be immediately obvious is cost. Obvious cost increases come from the added time spent during the design stage of a product. A less apparent increase to cost is in people. Software engineers are in general not well trained in the area of mathematical modeling and are often not experienced in the area of requirements engineering [Abr06]. With this view in mind it is often important to have at least one member of a development team with the specific skill set required and an engineer with such a skill set could possibly come with a higher price tag.

Also hindering the use of formal methods at an industrial level is the lack of a standard. There exist several varying strategies for formal methods such as the "B-method" and "Z" as well as a wide selection of various specification languages such as "CSP", "Larch" and "CASL". To a project manager with little experience in the area formal specification, the range of decisions required before even beginning may be to much of a boundary.

Due to these disadvantages formal methods are used in the majority in the area of critical systems, such as power plant management and aircraft control, where the additional cost in system development can be justified when considering the potential cost incurred if the system was to fail. Within this area of engineering it is unlikely that software engineers and developers are well trained on the numerous calculations required in a system. It is therefore necessary to make clear the intended operations of the system for all shareholders as early as possible.

An early example of successful use of formal methods in an industrial contex was the specification and verification of gate level logic in the VIPER 32-bit microprocessor, designed specifically for safety critical control applications for the UK Ministry of Defense [Sch89, p.2]. Due to the critical nature of the VIPER processor's operations, failure would lead to serious ramifications. It was recognized that testing strategies were insufficient for the adequate testing and error detection so the VIPER gate level logic design underwent formal specification and verification to prove its operations met the requirements laid out for the system [Sch89, p.2].

In the conclusion of the academic paper on the subject([Coh87]) the author mentions:

> *"The first level of the Viper proof, for example, takes several hours of CPU time to run, and that is using an inefficient pro-totype system. It took six months, however, to organize the proof and carry it out. We expect that subsequent lower-level stages will be much more complex"*

showing the extent to which formal methods and formal specification can extend the length of the design stage, even at early stages of the specification process.

$$\begin{tikzcd}
InformalSpecification \arrow[d, leftrightarrow] \\
FormalSpecification \arrow[d, leftrightarrow] \\
Implementation
\end{tikzcd}$$

Figure 3.2: Relationship between requirements, formal specification and implementation

3.2 Specification and Validation

Formal specification as mentioned previously fits into the design stage of system development. The objective of formal specification is to form a formal definition of the requirements of a system, thus removing ambiguity and uncertainty from the design design document and implementation, instead allowing the developers to predict the behavior and properties of the system before it has been built [Bjø06]. The results of system specification are equivalent to the effect mathematics and geometry had on architecture in early civilization, removing guesswork and trail and error.

When forming a system specification only the behavior of the system is to be modeled. This means that the specification engineer can largely ignore details do to with implementation decisions such as programming languages, and resource requirements of the system. This allows the engineer to be free to focus purely on the behavior of the system resulting in an additional level of abstraction above that of the program code. The place of the specification then, is between the informal design of the system and the implemented code.

As an easy example of this added layer of abstraction let us look at the specification laid out in Example 3.3 and an equivalent code sample, Example 3.4

$$(\forall a_i \in A)\,(a_i = a_i + 1)$$

Example 3.3: Take and array and increment each value by 1

and an equivalent code sample, Example 3.4

```
for (i=0; i <= length.A; i++)
        A[i] = A[i] + 1;
```

Example 3.4: Take an array and increment each value by 1

The specification, written in no particular language, is quite simple. Its intention is to model an operation which takes as an input an array of data, and increment each elements stored value by 1. The program in Example 3.4, written in Java like syntax, executes such behavior through use of a for loop. It is however possible to write an equivalent program using a while loop, as it is possible to write an equivalent programing using functional or logical programming, so long as the output of the program meets that specified the program is said to be correct.

In order to prove that the formal specification meets the requirements as laid out by the requirements document or design we must validate the specification. This can be done through two different methods:

- Manual - manually evaluate operations to determine the outcome
- Automatic - allow software tools such as automatic theorem provers to evaluate whether the operations laid down in the specification meet predetermined goals as laid down by the requirements document.

In order to prove specifications correct the developer uses use cases. Similar to test cases used in dynamic testing methods, the use cases are used to prove that the specification holds under given conditions. Unlike test cases, due to the well formed notation of the specification, it is easy to show that the specification holds for all cases. More information regarding automatic theorem provers (specifically SPASS) can be found in Section 4.2.2

It is important to note that validation is a separate process to verification. While verification compares the behavior of an implementation to the behavior of the system laid down in the specification, where as validation requires that the specification covers all the requirements enclosed in the requirements.

4 CASL - The Common Algebraic Specification Language

The Common Algebraic Specification language (CASL) is a specification language developed by the Common Framework Initiative group (CoFI). The aim of CoFI is to develop a common framework for the algebraic specification and development of software, the lack of which is considered to be a large factor in the lack of usage of algebraic specification techniques [BM04b]. The problem in this regard is the large number of languages and tools developed independently on one another, some only differing slightly however. Before starting specification of a system the developer must first consider which specification language to use based on areas that it its strong in. This lack of a single standard framework has been a prohibitive factor in the uptake of system specification and formal methods as a whole and so CoFI aims to tackle this through the development of a standard language.

In order to improve acceptance of CASL into the specification community key features of research applications and previous languages were assimilated and cooperated into the design of CASL. While the short term goal was to make CoFI the standard framework for research, the long term goal was to make CoFI the standard framework for specification in industry through the development of a strong support structure for its use including a concise reference manual, user guide and educational features as well as being free of charge to both academic and industrial institutions [BM04b].

When designing CASL it was vitally important to maintain the best a key features of as many existing specification languages that it succeeded and build a language that was :

> ”based on a critical selection of the constructs found in existing algebraic specification frameworks” [CoF10]

Such a consensus is difficult to achieve, and creating such a language can often result in one which is large, complicated and generally unusable. It was also considered that CASL would be used by those wanting a highly expressive powerful language as well as those wanting a relatively simple language. As such it was decided that CASL should be designed with its intended purpose being for the specification of functional requirements and design of software packages as abstract data types [CoF10]

CASL itself is the centre of a “family” of specification languages, and its functionality can be both extended and restricted by the use of extensions and sub languages respectively. This allows CASL to be used by those who have very specific requirements.

CASL specifications themselves can be split into four variations of specifications

- **Basic Specifications -** This is the base form of specification written in CASL, consisting of an unstructured collection of declarations and axioms

- **Structured Specifications -** This is a specification written in CASL in a modular manner, consisting of many basic specifications. This allows a large specification to be broken down into smaller logically organized parts, allowing for improved reusability of specifications and improved readability. The structure of the specification is not reflective of the actual structure of the implementation of the system specified.
- **Architectural Specifications -** Similar to structured specifications, architectural specifications are specifications broken down into smaller singular elements. In this form however, the system is to regarded as being broken down into smaller units to be specified individually and the manner in which the individual units are to be incooperated into the overall system
- **Libraries of Specifications -** As CASL has the ability to name specifications it is possible to form libraries of specifications for use in many projects. It is important to note that specifications within a CASL library must be linear. That is to say, a specification cannot make reference to a specification that has not previously been defined.

For the specification carried out in this project the required forms of specification are Basic, Structured and Library specifications. The specification will be broken down into smaller modules for both ease of reading and ease of validation and also because the system itself would be broken down into smaller subsystems, some on board the train, others embedded in the control system, the specification will not however specify how the units fit together. This structured specification is broken down into smaller basic specifications. Initially the CASL basic libraries for Nat and Set were used, however these were surplus to requirements and so a custom library was made containing specifications of Nats and Sets consisting only of the required operations.

4.1 Structure of CASL Specifications

CASL specifications consists of two main parts, the signature and the properties. As an example we will look at the specification of a queue of trains used within this project. This data structure is to be taken as a standard first in, first out data structure with each element of the queue being a TrainID.[1]

4.1.1 The Signature

A signature within the realm of formal specification is represented as a four-tuple consisting of:

[1]Note: The TrainQueue spec used in the following sections is of an earlier iteration than the final specification and only contains a subset of the operations of the final TrainQueue as seen in Appendix B.2

- The set S of all sorts. A sort of type s cannot be used in the other sets of the tuple unless it is defined in S.
- The set TF of all total functions. Each element of this set contains the name and profile; that is, the input and output sorts, of each total function. E.G.

$$TF = \{\_\_ + \_\_ : Value \times Value \rightarrow Value, \_\_ - \_\_ : Value * Value \rightarrow Value\}$$

- The set PF of all partial functions. As in TF, each element contains the name and profile of each partial function. The output of the partial function can either be of the type defined by the function profile, or undefined "⊥". Partial functions are donated by "→?".
- The set P of all predicates. Each element contains the name and input profile of the predicate.

$$\begin{tikzcd}
\sum(& S, \arrow[d] & TF, \arrow[d] & PF, \arrow[d] & P) \arrow[d] \\
 & Sorts & TotalFunctions & PartialFunctions & Predicates
\end{tikzcd}$$

Figure 4.1: Signature

As an example let us look at the signature for the queue of trains TrainQueue. Operations on the queue included in this example are:

- empty - a constant to denote the initial empty queue
- add - add an element to the queue
- remove - remove the front element of the queue
- front - returns the front element of the queue without removing it
- elem - holds if TrainID is an element of the queue

The signature written in the format above reads as follows:

- $S = \{Queue; TrainID\}$
- $TF = \{Remove : Queue- > Queue; Add : Queue \times TrainID- > Queue\}$
- $PF = \{Front : Queue- > ?TrainID\}$
- $P = \{\_\_elem\_\_ : TrainID \times Queue\}$

```
library LIBRARY

spec TrainQueue =
     sorts  Queue, TrainID
     ops    empty : Queue;
            add : Queue × TrainID → Queue;
            remove : Queue → Queue;
            front : Queue →? TrainID
     pred   __elem__ : TrainID × Queue
end
```

Example 4.2: Signature of a queue of trains in CASL

with example 4.2 being the equivalent signature written in CASL syntax.

4.1.2 The Properties

The second part of a formal specification in CASL is the properties. In this part we define the axioms of the functions, thus defining exactly how we wish the system to behave. As long as these properties meet the requirements of the system, validated in either the automatic or manual method mentioned in section 3.2 and as long as these axioms hold in the implementation, then we can be sure the implementation of the system is correct.

Example 4.3 is the specification of the train queue with a signature as shown in the previous section, this time including the axioms giving the following properties:

- The operations front and back over an empty queue returns undefined
- Removing an element from an empty queue returns the empty queue
- Removing an element from a queue removes the earliest added element still in the queue
- Front over a queue returns the earliest added element still in the queue
- No train is an element of the empty queue
- A train is an element of a queue as long as it has been added

library LIBRARY

spec TRAINQUEUE =
 sorts *Queue*;
  TrainID
 ops $empty : Queue$;
  $add : Queue \times TrainID \rightarrow Queue$;
  $remove : Queue \rightarrow Queue$;
  $front : Queue \rightarrow ?\ TrainID$
 pred $\_\_elem\_\_ : TrainID \times Queue$
 $\forall\ q : Queue;\ t, t1, t2 : TrainID$
 • $remove(empty) = empty$ %(Remove op on the empty queue)%
 • $remove(add(empty, t)) = empty$
 %(Remove op on the queue with single element)%
 • $remove(add(add(q, t1), t2)) = add(remove(add(q, t1)), t2)$
 %(Remove op on arbitrary length queue)%
 • $\neg\ def\ front(empty)$ %(Not def front of empty queue)%
 • $front(add(empty, t)) = t$
 %(Front of queue with single element)%
 • $front(add(add(q, t1), t2)) = front(add(q, t1))$
 %(Front of arbitrary length queue)%
 • $\neg\ t\ elem\ empty$ %(Elem check on empty queue)%
 • $\neg\ t1 = t2 \Rightarrow \neg\ t1\ elem\ add(empty, t2)$
 %(t1 elem queue is false if t1 has not been added)%
 • $t\ elem\ add(q, t)$ %(t elem queue is true if t has been added)%
 • $\neg\ t1 = t2 \Rightarrow (t1\ elem\ add(q, t2) \Leftrightarrow t1\ elem\ q)$
 %(elem check on arbitrary length queue)%

end

Example 4.3: Specification of a queue of trains with properties

The text contained within the %( )% notation are annotations used for the purpose of indicating to the reader the purpose of each rule and as a listing of axioms displayed by Hets, a tool set for the parsing and analysis of CASL specifications (see section 4.2.1).

4.2 CASL Tools

In order to facilitate the uptake of CASL by both the academic and industrial community it was vital that there was adequate tools available to enable users to take full advantage of the features of the language. The most important tool is the parser since before validation of the specification can begin, we first need to check that the syntax is correct.

4.2.1 Hets

The tool central to the completion of this project has been the heterogeneous toolset (Hets). Hets consists of a number of tool which can be used for the parsing, static analysis and proof management of specifications [Mos10]. Hets is available via a Java based installer application downloadable from the Hets homepage ([Mos10]). Hets is available for Mac OS®, Linux and PC Solaris® operating systems. This I feel may result in a lack of uptake from the industrial sector as the majority of companies use Windows® based PCs for their staff, however such a situation regarding tools is fairly typical for academic environments.

The Hets installer application also downloads added tools used in the analysis of the specification. UDraw(Graph) is a tool used to for the GUI of Hets, providing the user with a graphical user interface showing the hierarchical structure of a specification as well as providing an interface for the use of the theorem provers SPASS and ISABELL, the former of which is used to validate the specification developed in this project.

Figure 4.4 shows the architecture of Hets, the logic graph showing the relationship by the various CASL extensions and sub languages supported by hets forming the CASL extension HetsCASLS, which also contains the functional programming language Haskell.

As well as these analysis and parsing tools the Hets installation app also installs the CASL standard libraries, which contains specifications of many of the standard sorts such as numbers, strings, and sets. It also provides the option of installing “CASL mode” for the text editor emacs. This provides a developer with syntax highlighting to make the specification of systems in CASL slightly easier.

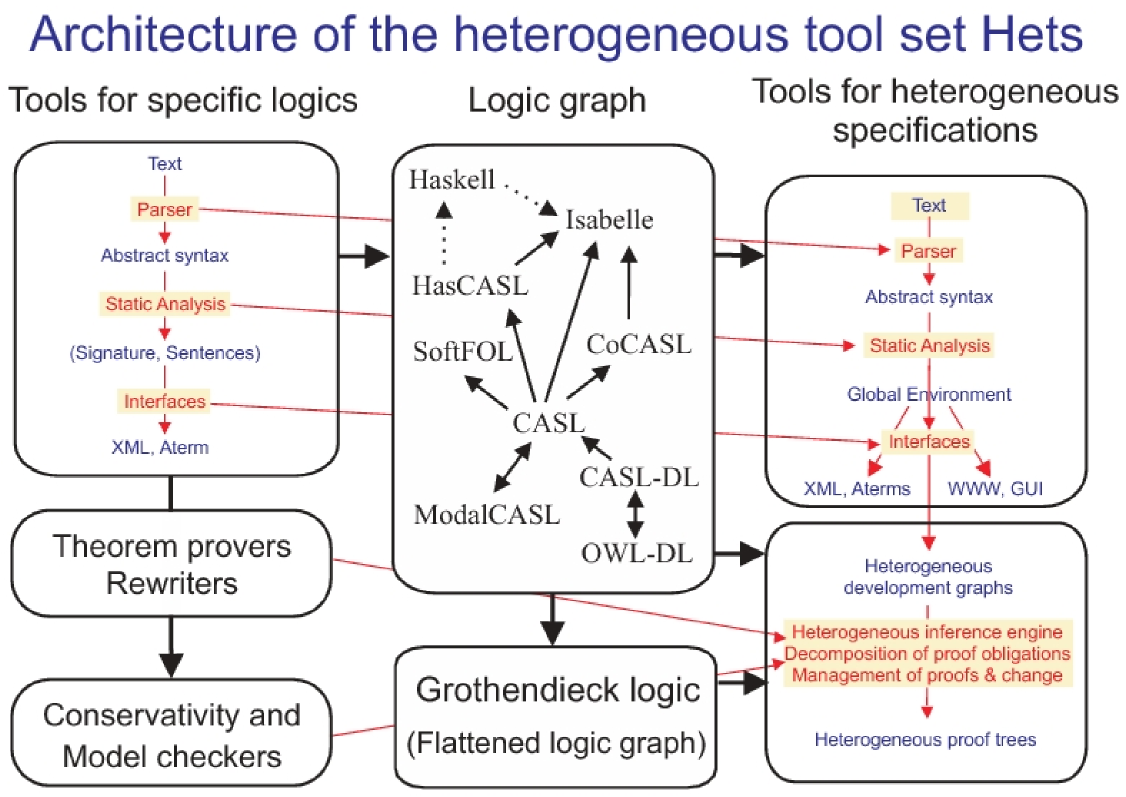


Figure 4.4: Architecture of the heterogeneous toolset Hets, source = [Mos10]

In order to begin analysis of specifications written in a Hets supported language we must execute the hets applications using the command line instruction as shown in figure 4.5.

```
hets [options] [filename]
```

Figure 4.5: Hets command in Linux terminal

This then executes the parser and static analysis of the file provided in filename. The parser first checks the file for syntax errors. It then attempts to display errors in a manner so the user is able to find the location of the error and have an idea of its nature. Figure 4.6 is an example of a syntax error message displayed by Hets. The message includes the filename in which the error was found since specifications can use multiple files, as well as the line and column numbers of the error. It also provides the expected nature of the error an possible solutions. In my experience while doing the project, the location of the error has been accurate however the nature of the error has often been incorrect with errors such as an unexpected letter, when in actuallity a semi-colon was missing from the line above. I feel that some improvement could be made to Hets in this regard.

```
    *** Error
{Directory}/{.CASL file}:7.14-7.20:
Unexpected "Q" or "-"
expecting "esort", "sort" ···
```

Figure 4.6: Syntax error message output by Hets

The second part of the Hets executable is the static analysis. This checks the well formedness of the specification by the CASL semantics. For example the static analysis will check that a sort has been declared before its use, or that the correct number of inputs has been used by an operator. Error messages displayed in this stage are similar to syntax error reporting, showing the filename, the line and column number of the error and the nature of the error. As well as these two primary functions Hets is also able to output the specification in various formats compatible with numerous document formats including .tex format which has allowed the specification developed in the project to be included in this document in a readable, orderly fashion without having to be concerned with layout of the specification. Inclusion of the specification in this method requires the downloading of the hetcasl LaTeXpackage available as a .sty file from the Hets homepage.

As previously mentioned Hets is also able to represent the hierarchy of specifications presented to it using the software tool UDraw(Graph). In order to active this mode we use the "*-g*" option. When running Hets in graphical mode the first thing to appear is two separate UDraw(Graph) windows, the first showing, as rectangular nodes, any specifications downloaded from the standard CASL library. When carrying out analysis of the specification this screen can be largely ignored since it is the second screen which contains the graphical representation of the desired system.

This graphical user interface is shown in figure 4.7 which contains the graphical representation of the TrainQueue example, this time including the specification of the use case of the train queue used for validation purposes.

The graph shown in figure 4.7 contains two elliptical nodes, each representing a specification. There is also a visible solid black arrow linking the two specifications. This represents that one specification references another, in this case that TrainQueueUSECASE refers to TrainQueue in its specification. This shows that TrainQueueUSECASE is able to make use of the sorts operations and predicates defined in TrainQueue. The blue link which both exits and enters the TrainQueueUSECASE specification is referred to in the Hets user manual as hiding, free or co-free definition links [MML10]. Other features of these graphs which which appear in graphs generated from the main specification will be explained as seen.

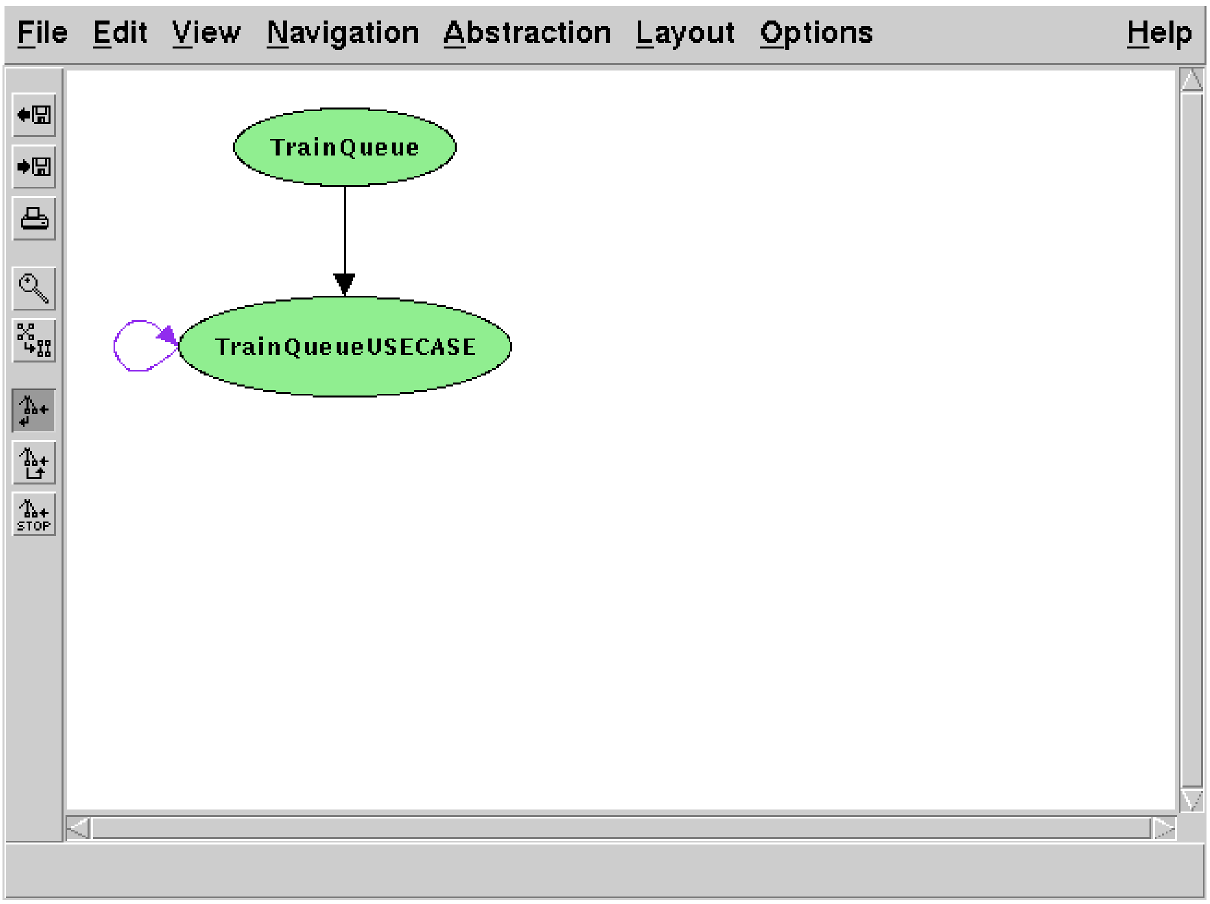


Figure 4.7: The initial window when opening hets in graphical mode

4.2.2 SPASS

There are a number of automatic theorem provers (ATPs) included in the installation of hets which all share the same GUI [MML10], the one chosen for this project is SPASS [tea3] since this is the theorem prover I have been taught during my studies.

SPASS is a first-order logic theorem prover developed for the formal analysis of software, systems, protocols and decision procedures [tea3].

In order to access the ATP interface we must first put the Hets interface into proof mode. We do this by using the edit drop down menu Edit>Proofs>Automatic. This changes the graph slightly as shown in figure 4.8.

In figure 4.7 the proof is represented by the red circular node and the green arrows correspond to proven theorem links [MML10] and black arrows again show referencing of specifications.

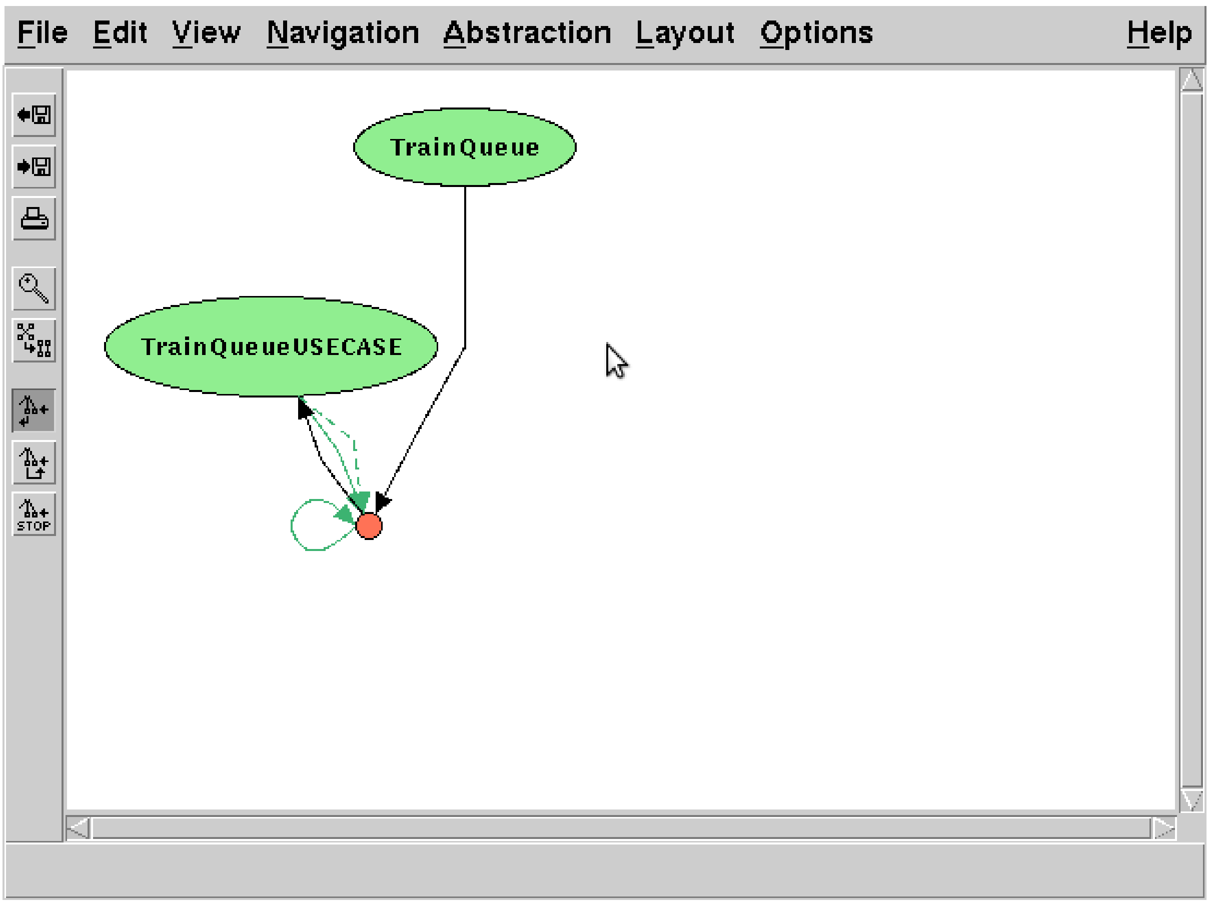
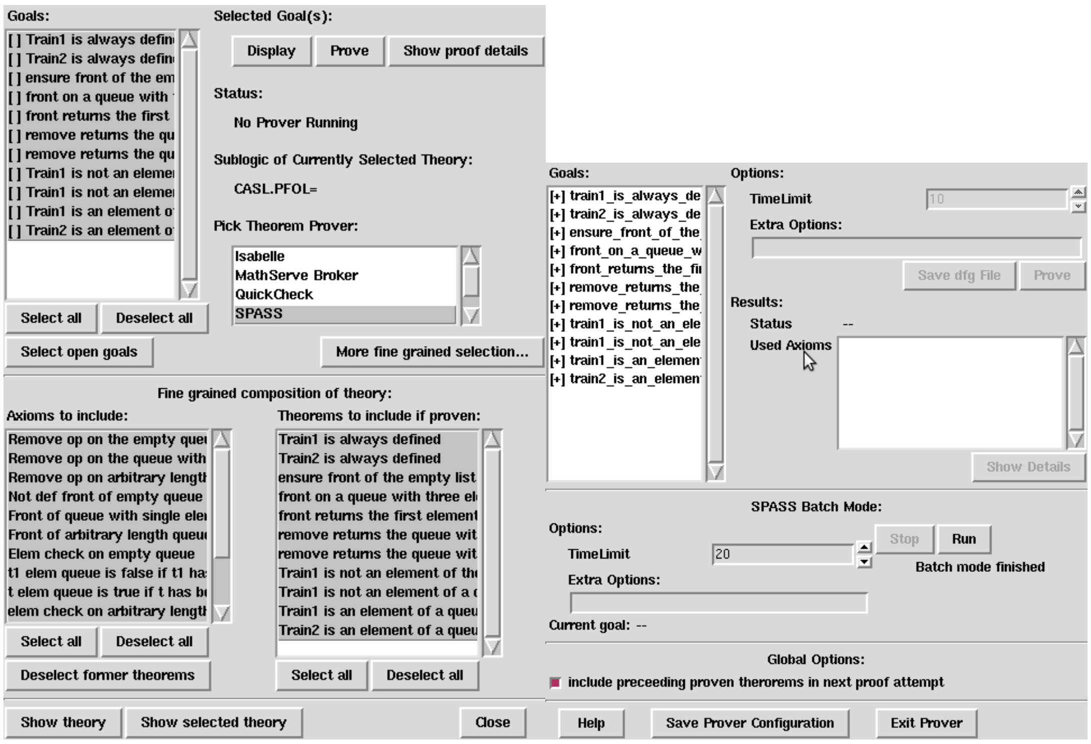


Figure 4.8: Graphical representation of the links between the proof, the specification and use case

By right clicking on on this red node we access the node menu and by click “Prove” in this menu we access the interface for ATPs as seen in figure 4.9a. The top half of this interface consists of two lists. The first shows the lists of goals we wish to prove which can be found in the TrainQueueUSECASE specification. This list allows us to choose which goals we wish to try and prove. The second list on the top half of the page allows us to choose which theorem prover to use. The lower half of this window allows us to choose which axioms to include, and which theorems to include if proven. The names of axioms in the list are those given by the “ %( )%” anotations as seen in section 4.1.2. If these anotations were not included in the specification the axiom names revert to a automatic numbered naming system.

By selecting SPASS and pressing the “Prove” button on the upper half of this interface we are presented with the SPASS interface. This again shows us the list of goals we wish to prove and allows us to set time limits for the prover. It is possible to either attempt the proofs one at a time or in batch mode. Figure 4.9b demonstrates this window after the prover has been run. Beside each goal there are “[]” which filled depending on the current state of the goal as



Goals:
[] Train1 is always defin
[] Train2 is always defin
[] ensure front of the em
[] front on a queue with
[] front returns the first
[] remove returns the qu
[] remove returns the qu
[] Train1 is not an eleme
[] Train1 is not an eleme
[] Train1 is an element o
[] Train2 is an element o

Selected Goal(s):

Status:
No Prover Running

Sublogic of Currently Selected Theory:
CASL.PFOL=

Pick Theorem Prover:
Isabelle
MathServe Broker
QuickCheck
SPASS

Fine grained composition of theory:

Axioms to include:
Remove op on the empty quei
Remove op on the queue with
Remove op on arbitrary lengtl
Not def front of empty queue
Front of queue with single ele
Front of arbitrary length queu
Elem check on empty queue
t1 elem queue is false if t1 ha
t elem queue is true if t has b
elem check on arbitrary lengtl

Theorems to include if proven:
Train1 is always defined
Train2 is always defined
ensure front of the empty list
front on a queue with three el
front returns the first element
remove returns the queue wit
remove returns the queue wit
Train1 is not an element of th
Train1 is not an element of a
Train1 is an element of a queu
Train2 is an element of a queu

Goals:
[+] train1_is_always_de
[+] train2_is_always_de
[+] ensure_front_of_the
[+] front_on_a_queue_w
[+] front_returns_the_fi
[+] remove_returns_the
[+] remove_returns_the
[+] train1_is_not_an_ele
[+] train1_is_not_an_ele
[+] train1_is_an_elemen
[+] train2_is_an_elemen

Options:
TimeLimit 10
Extra Options:

Results:
Status --
Used Axioms

SPASS Batch Mode:
Options:
TimeLimit 20
Extra Options:
Batch mode finished
Current goal: --

Global Options:
include preceeding proven therorems in next proof attempt

(a) Goals and Prove Interface (b) SPASS Interface

Figure 4.9: Proving Interfaces

follows:

- [ ] - Open, or Open (Time is up!). Shows goal is yet to be proved. If prover has been run then shows the goal could not be proven in the given time limit.
- [+] - Proved. Goal has been proven correct
- [x] - Proved (Theory Inconsistent). Shows the goal has been proved in inconsistent results.
- [−] - Disproved. Goal has been shown incorrect.

As it can be seen in Figure 4.9b all goals have been proved showing that the specification TrainQueue is correct to the requirements laid down in section 4.1.1. exiting the prover to view the graphical representation again it can be seen that the proof node colour is now green, denoting the proof has been satisfied.

5 Considerations Before Specification

Before commencing with the specification of the moving block interlocking system, it was obviously necessary to obtain a good understanding of the system to be specified. This has been the purpose of the previous sections of this document, to gain a knowledge in both railway interlocking systems and knowledge in formal methods, CASL and tools required for the specification process.

Knowledge regarding the railway interlocking systems has also been gained through experience as a summer intern at Westinghouse Railway Systems Ltd. which forms part of Invensys Rail group, one of six companies involved in the formation of the UNISIG consortium.

In order to avoid becoming overwhelmed with details of the system we must define a subsection of the system which we are defining. For this early stage of specification we will be attempting to specify the critical aspect of the basic moving block system. That is to say we will be attempting to model a track of arbitrary length, with an arbitrary number of linked control systems and an arbitrary number of trains. The track will be unidirectional and will not contain more complex features such as points, crossings or stations. Section 5.1 will contain more details of the system to be specified.

5.1 Informal Requirements

Before specification of the system can begin it is vitally important that the requirements of the system are laid out in a requirements document, often written in an informal manner for readability purposes. The full informal requirements of the system to be specified can be found in Appendix A.

5.2 Concepts of Time in Interactive Systems

Another aspect to be considered before commencing with the specification is how to model the passing of time in an algebraic specification. In Control systems a common method is to model the system as a series of states, while also modeling what operations are available from the current state and the possible reachable states. Such modeling can be represented graphically as finite state automita. This is the approach taken in the steam boiler case study in [BM04b], which initially acted as a template for the specification carried out for this project.

The other method for modeling time in interactive systems, and the approach I have taken is the stream method. In this method streams of input and output data are used as well as a time measurement (usually the natural numbers). While not strictly accurate the system does allow the developer to specify how the system should behave over a space of time rather than over a series of states. In the specification of the train system we still use states, in the form of predicates, to denote if an error has occurred in the transfer of messages. Due to using this method of time model certain considerations had to be considered when carrying out the specification. The first is that speed values, since they are of the form distance traveled per time, should in someway reflect the relative speed of the train compared to the cycle time of the system. In railway systems, the update cycle occurs every few fractions of a second and so consideration should be given as to whether the unit of speed is sufficiently small to indicate the small distance traveled at each time, or the value given to speed for validation purposes should be sufficiently small. For this level of specification we will use the later since units of measure are something that could be modeled at a later date. For this level of specification, abstraction allows us to remove this concern. Using the approach did present problems when using the automatic theorem prover SPASS. It has been suggested that automatic theorem provers are built with the assumption that the state modeling method will be used and as such are unable to cope with the stream method of time modeling demonstrated by the excessive time required to prove the TrainWithMessagesUSECASE specification, because of the this SPASS has been demonstrated using the TrainQueue specification, while other specifications will be validated by hand in a somewhat intuitive manner since they are being validated against and informal requirements document.

5.3 Approach to Specification

The approach taken towards the specification was similar to approaches usually done in programming. For readability and ease of analysis a modular specification was required prompting the use of a structured specification. The specification process was carried out in an iterative manner similar to the spiral approach to software engineering [Som06]. This meant starting with an initial specification containing the minimum aspect of the system, starting with the concept of a Value (a sub sort of Nat), and the specification for a train including its operations for speed, position and length. Just as in the spiral method at each iteration it was ensured that each new aspect to the specification was correct to requirements, only then could the next iteration begin.

During meetings with the supervisor of the project, the specification technique became somewhat similar to the extreme programming technique. As in extreme programming, with two focusing on the same specification on one computer the second specifier is able to identify errors and make suggestions as the first types. These meetings were very constructive and this is definitely an approach to specification I would suggest be considered in future projects, especially as my relative lack of experience in the field was complimented by the supervisor's knowledge and expertise.

6 Analysis of the Specification

In this chapter we will look at the specification of the moving block railway

The system will consist of two main physical units, the control system and the trains. Communication between the two systems will be in the form of Train Messages (messages from the train to the control system) and Control Messages (messages from the control system to the train). We will look at each specification individually and validate its rules against the requirements laid down in Appendix A. Where appropriate I will attempt to link axioms, operations and predicates to requirement numbers included in appendix A.

Appendix D shows the graphic representation of the hierarchy of the specification to be analysed, indicating the references to other specifications made. In the analysis of the specification we will follow the order that the specification was written and is shown in appendix B.2.

The analysis in the following sections will contain sections of the final specification of the system which is available in full in appendix B.2

6.1 Train Library

The specifications contained here are not a requirement of the specification since they form a subset of the specification of the natural numbers and the specification of a generated set as found in the standard CASL libraries [BM04a]. Initially it is these standard libraries that were used however when parsing the specification using Hets the entire file of the imported specification would be used increasing parsing time considerably. This library was then developed to improve the speed of parsing for more efficient development.

6.1.1 VALUE

The first specification, VALUE, is a subtype of the natural numbers containing the standard operations of addition, subtraction, multiplication and the predicate less than. This can be seen in the signature shown in figure 6.1. This type is then used later to define the sorts "speed", "position", "length" and "time"

free type $Value ::= 0 \mid suc(Value)$
ops $\_\_+\_\_ : Value \times Value \to Value, assoc, comm, unit\ 0;$
$\_\_-\_\_ : Value \times Value \to Value;$
$\_\_*\_\_ : Value \times Value \to Value, comm;$
%%Operations to represent the natural numbers as digits
$1 : Value = suc(0);$
$2 : Value = suc(1);$
$3 : Value = suc(2);$
$4 : Value = suc(3);$
$5 : Value = suc(4);$
$6 : Value = suc(5);$
$7 : Value = suc(6);$
$8 : Value = suc(7);$
$9 : Value = suc(8);$
$\_\_@@\_\_(x : Value; y : Value) : Value$
$= (x * suc(9)) + y$
pred $\_\_<\_\_ : Value \times Value$

Figure 6.1: Signature of VALUE

The operators in figure 6.1 under the heading "*Operations to represent the natural numbers as digits*" are used to allow the use of digits to represent Values.

6.1.2 Addition

The rules for addition are shown below.

- $0 + x = x$ %(Value add zero)%
- $suc(x) + y = suc(x + y)$ %(Value Addition)%

To validate this is correct we take the use case $2 + 3 = 5$

$$\begin{aligned} 2 + 3 = suc(1) + 3 = suc(1 + 3) = suc(suc(0) + 3) = \\ suc(suc(0 + 3) = suc(suc(3)) = suc(4) = 5 \end{aligned} \qquad (6.1.2.1)$$

6.1.3 Subtraction

The rules given for subtraction are:

- $0 - x = 0$ %(Zero sub Value)%
- $suc(x) - 0 = suc(x)$ %(Value sub zero)%
- $suc(x) - suc(y) = x - y$ %(Value Subtraction)%

These rules show that ”-” is used to denote $\dot{-}$ of natural numbers

Valdiate with use case $3 - 2 = 1$:

$$\begin{aligned} 3 - 2 = suc(2) - suc(1) = 2 - 1 = \\ suc(1) - suc(0) = 1 - 0 = 1 \end{aligned} \tag{6.1.3.1}$$

Use case $2 - 3 = 0$

$$\begin{aligned} 2 - 3 = suc(1) - suc(2) = 1 - 2 = \\ suc(0) - suc(1) = 0 - 1 = 0 \end{aligned} \tag{6.1.3.2}$$

6.1.4 Multiplication

The rules for multiplication are as follows:

- $x * 0 = 0$ %(Value mult zero)%
- $x * suc(y) = (x * y) + x$ %(Value Multiplication)%

To validate we take use case of $2 * 3 = 6$

$$\begin{aligned} 2 * 3 = 2 * suc(2) = (2 * 2) + 2 = \\ (2 * suc(1)) + 2 = ((2 * 1) + 2) + 2 = \\ ((2 * suc(0)) + 2) + 2 = (((2 * 0) + 2) + 2) + 2 = \\ (((0) + 2) + 2) + 2 = 2 + 2 + 2 = 6(\text{By \%(Value Addition)\%}) \end{aligned} \tag{6.1.4.1}$$

6.1.5 Less Than

Rules for Less than predicate ”<”:

spec MYGENERATESET[**sort** *Elem*] = %**mono**
    generated type
    $Set[Elem] ::= \{\} \mid \_\_+\_\_(Set[Elem]; Elem)$
    pred $\_\_eps\_\_ : Elem \times Set[Elem]$
    $\forall\, x, y : Elem;\ M, N : Set[Elem]$
    • $\neg\ x\ eps\ \{\}$ %(elemOf_empty_Set)%
    • $x\ eps\ M + y \Leftrightarrow x = y \vee x\ eps\ M$
    %(elemOf_NonEmpty_Set)%
    • $M = N \Leftrightarrow \forall\, x : Elem \bullet x\ eps\ M \Leftrightarrow x\ eps\ N$ %(equality_Set)%
end

Figure 6.2: Specification of a generated set

• $0 < suc(x)$ %(0 is the lowest Value)%
• $\neg\ suc(x) < 0$ %(No Value less than 0)%
• $suc(x) < suc(y) \Leftrightarrow x < y$ %(Value less than def)%

Validation of true with use case 2 < 3:

$$\begin{aligned} 2 < 3 = suc(1) < suc(2) \Leftrightarrow 1 < 2 = \\ suc(0) < suc(1) \Leftrightarrow 0 < 1 = TRUE \end{aligned} \tag{6.1.5.1}$$

Validation of false with use case 3 < 2:

$$\begin{aligned} 3 < 2 = suc(2) < suc(1) \Leftrightarrow 2 < 1 = \\ suc(1) < suc(0) \Leftrightarrow 1 < 0 = FALSE \end{aligned} \tag{6.1.5.2}$$

6.1.6 MyGenerateSet

The specification for the Set type used for the sets of messages in the specification is given by the specification MyGenerateSet given in figure 6.2.

This specification defines a set as being of a "*generated type*". This means that the coresponding sort is constrained to be generated by the declared constructors, meaning any value of this sort is built by application of constructors [BM04b].

The predicate "*eps*" returns true if an Element is in the given set. Its validation can be done intuitively:

- x is not in the empty set
- if $x = y$ then obviously x is in the set $M + y$
- if $x \neq y$ then x is in the set $M + y$ if and only if x is in the set M

I do not feel that more validation is required for this predicate or set equality.

6.2 Basic Train

Now we look at the specification of a basic train. This is the abstract data type of a train and is later expanded to be able to operate on messages received from the control system. It can be seen that the specification of a train contains the sorts speed, time, position, length and breakingdist, each of which is equal to the sort Value. This means that these sorts inherit all the operations of Value. The use of individual sorts is done to indicate that in future iterations of specification these may take on different units.

For now, the specification of a Train contains the following operations:

- Speed: TrainID $\times$ time $\rightarrow$ speed - returns the speed of the given train at given time (Requirement 9.1)
- MaxSpeed: TrainID $\rightarrow$ speed - returns the maximum speed of the given train (Requirement 9.2)
- Position: TrainID $\times$ time $\rightarrow$ position - returns the position of the given train at given time (Requirement 9.6)
- Length: TrainID $\rightarrow$ length - returns the length of a given train (Requirement 9.7)
- BreakingDist: TrainID $\times$ time $\rightarrow$ brakingdist - returns the breaking distance of the given train at given time
- BreakingFunction: speed $\rightarrow$ Value - returns the breaking distance at a given speed

and is given by the following CASL specification

spec TRAIN =
VALUE
then sorts *TrainID*;
speed = *Value*;
time = *Value*;

```
        $position = Value;$
        $length = Value;$
        $brakingdist = Value$
    **ops** $Speed : TrainID \times time \to speed;$
        $MaxSpeed : TrainID \to speed;$
        $Position : TrainID \times time \to position;$
        $Length : TrainID \to length;$
        $BrakingDist : TrainID \times time \to brakingdist;$
        $BrakingFunction : speed \to Value$
    $\forall\ train : TrainID;\ s : speed;\ t : time;\ p : position;\ l : length$
⋮
**end**
```

The specification above contains axioms for the operations Position, Braking distance and BreakingFunction.

6.2.1 Position

The rules for Position are:

- $Position(train, suc(t))$
  $= Position(train, t) + Speed(train, t)$
  %(Train Position calculation)%

Since this is a simple addition of two sorts equal to Value, and the validation of addition is carried out in 6.1.2 we know that this rule holds. The value calculated in thes operations goes towards the calculation of block boundaries for requirement 3.

6.2.2 BreakingDist and BreakingFunction

The rules for these two fuctions are:

- $BrakingDist(train, t) = BrakingFunction(Speed(train, t))$
  %(Breaking distance is braking function of the speed at time t)%
- $s < 2 \Rightarrow BrakingFunction(s) = s$

%(Breaking function = s if s < 2)%

• $2 < s$
$\Rightarrow BrakingFunction(s) = s + BrakingFunction(s - 2)$
%(Braking function = s + brakingfunction after braking)%

It can be seen that BrakingDist is equal to the BrakingFuntion over the speed of the given train at the given time so we focus our analysis on the breaking function.

In the informal requirements (appendix A) we laid out that the train would have two methods of braking, the first is deceleration of one unit of speed per time, the other breaking at two units of speed per time. Using these requirements and the unit of speed as distance per unit time we can know that the breaking distance of the train is equal to the distance the train will travel reducing its speed by two at each unit time until it reaches 0, therefore when the speed of the train is less than 3 the train is able to stop in one unit time leading to the base case. We then go on to give the recursive definition of the braking function as being the speed of the train at the given time plus the the breaking function over the speed of the train after breaking for one unit time.

Validation with use case $Speed(Train, t) = 10$, $BreakingDist(Train, t) = 30$

$$\begin{gathered} BreakingDistance(Train, t) = BreakingFunction(Speed(Train, t)) \\ = BreakingFunction(10) = 10 + BreakingFunction(10 - 2) = \\ 10 + 8 + BreakingFunction(8 - 2) = 10 + 8 + 6 + BreakingFunction(6 - 2) \\ 10 + 8 + 6 + 4 + BreakingFunction(4 - 2) = 10 + 8 + 6 + 4 + 2 = 30 \end{gathered} \tag{6.2.2.1}$$

6.3 Messages

Messages are used for two purposes in the system:

- Train Messages are messages sent by the train to the control system reporting its speed, position, and length. (Requirements 10.1 - 10.3)
- Control Messages are messages sent by the control system instructing the train in what action should be taken. (Requirements 11.1 - 11.5

The specifications for the messages, as well as for the sets of messages used to transfer the messages are given below

spec TRAIN_MESSAGES =

```
      TRAIN
then free type
      T_Message
      ::= TRAINPOSITION(position)
        | TRAINSPEED(speed)
        | TRAINLENGTH(length)
end

spec CONTROL_MESSAGES =
      VALUE
then free type
      C_Message
      ::= CONTINUE
        | BREAK
        | DECELERATE
        | ACCELERATE
        | STOP
end

spec MESSAGES =
      MYGENERATESET[TRAIN_MESSAGES fit Elem ↦ T_Message]
and   MYGENERATESET
      [CONTROL_MESSAGES fit Elem ↦ C_Message]
end
```

Free data type decelerations as shown above denotes that the values of sorts defined by free type can be nothing other than those declared. E.G. the value of a sort T_Message can only be TRAINPOSITION(position), TRAINSPEED(speed) or TRAINLENGTH(length).

In the later specification of the control system it can be seen that there is only ever one element in the set Control_Messages, so a reader would be justified in asking why use a set at all. The answer for this is to better facilitate future enhancements to the system, in particular in the case the the control system may need to send more information the the train that the instructions included in this specification such as distance to next station etc.

Since these specifications do not contain axioms, only constructors of the data types validation is not required.

6.4 Train With Messages

We now look to extend the previous specification to take into account the specifications of messages and set of messages. In this specification the following operations:

- MessagesFromTrain - Returns the set of messages to be sent from a given train at given time. (Requirement 1)
- MessagesToTrain - Returns the set of messages received by a given train at given time. (Requirement 4

As well as the following predicates:

- MessagesOK - returns true if the given set of messages is valid, (in this case contains a unique C_Message).
- TransmissionOk - returns true the set of messages received by a given train at give time returns true from MessagesOK.

these operations and predicates are given by the following CASL specification:

spec TRAINWITHMESSAGES =
MESSAGES
then ops $\mathit{MessagesFromTrain}$
$: \mathit{TrainID} \times \mathit{time} \rightarrow \mathit{Set}[\mathit{T\_Message}];$
$\mathit{MessagesToTrain} : \mathit{TrainID} \times \mathit{time} \rightarrow \mathit{Set}[\mathit{C\_Message}]$
preds $\mathit{MessagesOk} : \mathit{Set}[\mathit{C\_Message}];$
$\mathit{TransmissionOk} : \mathit{TrainID} \times \mathit{time}$
$\forall\ \mathit{train} : \mathit{TrainID};\ s : \mathit{speed};\ t : \mathit{time};\ p : \mathit{position};\ l : \mathit{length}$

The first rule in this specification is the rule defining MessagesFromTrain:

$\forall\ \mathit{train} : \mathit{TrainID};\ s : \mathit{speed};\ t : \mathit{time};\ p : \mathit{position};\ l : \mathit{length};$
$\mathit{brakingFactor} : \mathit{Value}$
$\bullet\ \mathit{MessagesFromTrain}(\mathit{train},\ t)$
$= ((\{\} + \mathit{TRAINSPEED}(\mathit{Speed}(\mathit{train},\ t)))$
$+ \mathit{TRAINPOSITION}(\mathit{Position}(\mathit{train},\ t)))$
$+ \mathit{TRAINLENGTH}(\mathit{Length}(\mathit{train}))$
%(Messages sent to control System calc)%

This simply states that the set of messages from a given train at a given time is the speed, position and length of the train at that time and does not require further analysis.

The axioms for the predicats MessagesOK and TransmisionOk are:

- $\forall\ m : Set[C\_Message]$
  - $MessagesOk(m)$
    $\Leftrightarrow \exists!\ message : C\_Message \bullet message\ eps\ m$
    %(Check that a unique message was recived from control system)%
  - $TransmissionOk(train,\ t)$
    $\Leftrightarrow MessagesOk(MessagesToTrain(train,\ t))$

This rule states that MessageOK is true if and only if MessagesToTrain for the given train and time contain a single C_Message. This is indicated by the use of the unique existential quantifier ∃!

6.4.1 Messages received analysis

It was planned to use the automatic theorem prover SPASS to validate the specification for a train with messages. The use case used however does not seem compatible with SPASS since the proof takes an improbable amount of time to complete and thus remains open. Therefore I will attempt to prove the following use case manually.

spec TRAINWITHMESSAGESUSECASE =
TRAINWITHMESSAGES
then op *train1 : TrainID*
%**initialisation**, speed 9 position 0 at time 0%
- $Speed(train1,\ 0) = 9$
- $Position(train1,\ 0) = 0$
- $MessagesToTrain(train1,\ 0) = \{\} + ACCELERATE$
- $MessagesToTrain(train1,\ 1) = \{\} + CONTINUE$
- $MessagesToTrain(train1,\ 2) = \{\} + STOP$
- $MessagesToTrain(train1,\ 3) = \{\} + ACCELERATE$
- $MessagesToTrain(train1,\ 4) = \{\} + ACCELERATE$
- $MessagesToTrain(train1,\ 5) = \{\} + ACCELERATE$
- $MessagesToTrain(train1,\ 6) = \{\} + ACCELERATE$
- $MessagesToTrain(train1,\ 7) = \{\} + DECELERATE$
- $MessagesToTrain(train1,\ 8) = \{\} + BREAK$

then %implies

- $Position(train1,\ 1) = 9$
  %(The position at time 1 is equal to the position at time 0 plus the speed at time 0)%
- $Speed(train1,\ 1) = 10$
  %(the speed of the train increases by 1 if the train recieves an accelerate message)%
- $Position(train1,\ 2) = 19$
  %(the position of the train at time 2 is equal to the position at time 1 + the position)%
- $Speed(train1,\ 2) = 10$
  %(The Train stops if it recieves a stop signal)%
- $Position(train1,\ 3) = 29$
  %(The train does not move when speed is zero)%
- $Speed(train1,\ 3) = 0$
  %(The train is able to start again once stopped with an accelerate signal)%
- $Speed(train1,\ 7) = 4$ %(The speed at time 7 = 4)%
- $Speed(train1,\ 8) = 3$ %(The speed has been reduced by 1)%
- $Speed(train1,\ 9) = 1$ %(The speed has been reduced by 1)%

end

The axioms listed under "initialisation" are in place to simulate a stream of messages occurring over 10 time intervals in order to simulate how speed is effected.

At time 0 we have the case where the speed of train1 = 9 and position = 0. We prove the axiom "*The position at time 1 is equal to the position at time 0 + speed at time 0*" as

$$\begin{aligned} Position(train1, 1) &= Position(train1, suc(1)) + Speed(train1, suc(1)) \\ &= Position(train1, 0) + Speed(train1, 0) = 0 + 9 = 9 \end{aligned} \tag{6.4.1.1}$$

In order to calculate Speed(train1, 1) we make used of the rule:

- $ACCELERATE\ eps\ MessagesToTrain(train,\ t)$
  $\wedge\ \neg\ Speed(train,\ t) = MaxSpeed(train)$
  $\wedge\ TransmissionOk(train,\ t)$
  $\Rightarrow Speed(train,\ t + 1) = Speed(train,\ t) + 1$

since we know:
$ACCELERATE\ \epsilon\ MessagesToTrain(train1, 0) = TRUE$
and
$Speed(train1, 0) \neq MaxSpeed(train1)$
and can deduce:

$MessagesOK(MessagesToTrain(train1, 0)) = TRUE$ since ACCELERATE is the only element of set
then
$Speed(train1, 1) = Speed(train1, 0) + 1 = 10$ (Requirement 11.2)

For Speed(train,2) we follow the same method using the rule:

- *CONTINUE eps MessagesToTrain*(*train*, *t*)
  ∧ *TransmissionOk*(*train*, *t*)
  ⇒ *Speed*(*train*, *t* + 1) = *Speed*(*train*, *t*)
  %(Messages to Train CONTINUE evaluation)%

we know
$CONTINUE \epsilon MessagesToTrain(train1, 1) = TRUE$
and
$MessagesOK(MessagesToTrain(train1, 1)) = TRUE$ since CONTINUE is the only element of set
then:
$Speed(train1, 2) = Speed(train1, 1) = 10$ (Requirement 11.3)

we can now work out the position of the train at time 3 by:

$$\begin{aligned} Position(train1, 3) = Position(train1, 2) + Speed(train1, 2) = \\ (Position(train1, 1) + Speed(train1, 1)) + 10 = \\ 9 + 10 + 10 = 29 \text{(Requirement 9.6)} \end{aligned} \tag{6.4.1.2}$$

The next messaged received by the train (at time 2) is STOP and so we use the rule:

- *STOP eps MessagesToTrain*(*train*, *t*)
  ∧ *TransmissionOk*(*train*, *t*)
  ⇒ *Speed*(*train*, *t* + 1) = 0
  %(Messages to Train STOP evaluation)%

we can deduce from:
$STOP \epsilon MessagesToTrain(train1, 2) = TRUE$
and
$MessagesOk(MessagesToTrain(train1, 2) = TRUE$ since STOP is the only element of set
then
$Speed(train1, 3) = 0$ (Requirement 11.1)

Now that we have validated the ACCELERATE evaluation we move ahead to time 7. At this point the train as received four ACCELERATE messages since the STOP message so $Speed(train1, 7) = 4$ there for using the rule:

- $DECELERATE\ eps\ MessagesToTrain(train,\ t)$
  $\wedge \neg\ Speed(train,\ t) = 0 \wedge TransmissionOk(train,\ t)$
  $\Rightarrow Speed(train,\ t + 1) = Speed(train,\ t) - 1$
  %(Messages to train DECELERATE evaluation)%

we know
$DECELERATE \ \epsilon \ MessagesToTrain(train1, 7) = TRUE$
and
$MessagesOk(MessageToTrain(train1, 7) = TRUE$ since DECELERATE is the only element of set
and
$Speed(train1, 7) \neq 0$
then
$Speed(train1, 8) = Speed(train1, 7) - 1 = 3$ (Requirement 11.4)

The next message we have is $MessagesToTrain(train1, 8) = +BREAK$, we therefore take the rule:

- $BREAK\ eps\ MessagesToTrain(train,\ t)$
  $\wedge \neg\ Speed(train,\ t) < 2 \wedge TransmissionOk(train,\ t)$
  $\Rightarrow Speed(train,\ t + 1) = Speed(train,\ t) - 2$
  %(Messages to train BREAK evaluation)%

we know:
$BREAK \ \epsilon \ MessagesToTrain(train1, 8) = TRUE$
and
$MessagesOk(MessagesToTrain(train1, 8) = TRUE$ since BREAK is the only element of the set
and
$\not Speed(train1, 8) < 3$
then
$Speed(train1, 9) = Speed(train1, 8) - 2 = 3 - 2 = 1$ (Requirement 11.5)

6.5 Control System

The next specification is that of a single control system. This specification contains the following operations:

- MessagesFromTrain - gives the set of messages received from a give train at a given time (Requirement 1)
- MessagesToTrain - gives the set of messages to be sent to a given train at a given time (Requirement 4)
- BlockFront - gives the position of the front of a block of a given train at a given time (Requirement 3)
- BlockFrontAcceled - as BlockFront but gives front of block as if train was 1 unit speed faster (Requirement 3)
- BlockFrontDeceled - as BlockFront but gives front of block as if train was 1 unit speed slower (Requirement 3)
- seenSpeed|Position|Length - Gives the seen speed, position, and length accordingly for a given train at a given time (Requirement 2)
- messagesToSpeed|Position|Length - Extracts speed, position and length data from a set of T_Message (Requirement 2)

As well as the following predicates:

- FailureMode - indicates if the system is in failure mode at a give time
- SafeMode - indicates if the system is in safe mode at a give time
- TransmissionOk - indicates that all messages were received correctly at a give time
- MessagesOK - indicates that the set of T_Message contains valid data.

These predicates are used to show the state that the control system is in either safe or failure state.

These operators and predicates are given by the following signature:

spec ControlSystem =
Messages

then ops $MessagesFromTrain$
$: TrainID \times time \to Set[T\_Message];$
$MessagesToTrain$
$: TrainID \times time \to Set[C\_Message];$
$BlockFront : TrainID \times time \to position;$
$BlockFrontAcceled : TrainID \times time \to position;$
$BlockFrontDeceled : TrainID \times time \to position;$
$BlockRear : TrainID \times time \to position;$
$seenSpeed : TrainID \times time \to ?\ speed;$
$seenPosition : TrainID \times time \to ?\ position;$
$seenLength : TrainID \times time \to ?\ length;$
$messagesToSpeed : Set[T\_Message] \to ?\ speed;$
$messagesToPosition : Set[T\_Message] \to ?\ position;$
$messagesToLength : Set[T\_Message] \to ?\ length$
preds $FailureMode : time;$
$SafeMode : time;$
$TransmissionOk : time;$
$MessagesOK : Set[T\_Message]$

6.5.1 FailureMode Analysis

The purpose of the FailureMode and SafeMode predicates is to indicate whether the control system is in a safe state. At this level of specification we have only taken into account the correct transmission of messages from the trains to the control system. The correct format of these messages is laid out in the rule:

• $\forall\ m : Set[T\_Message]$
  • $MessagesOK(m)$
    $\Leftrightarrow \exists!\ s : speed;\ p : position;\ l : length$
      • $TRAINSPEED(s)\ eps\ m$
        $\wedge\ TRAINPOSITION(p)\ eps\ m$
        $\wedge\ TRAINLENGTH(l)\ eps\ m$
%(Check that messages were recieved correctly)%
$\forall\ t : time$
• $TransmissionOk(t)$
  $\Leftrightarrow \forall\ train : TrainID$
    • $MessagesOK(MessagesFromTrain(train,\ t))$
%(Check transmission from all trains successful)%

The rule states that for each set of messages recieved from the train there should be a unique TRAINSPEED, TRAINPOSITION and TRAINLENGTH message. This is required otherwise the messagesToSpeed|Position|Length operations would return inconsistent results for each T_message resulting in inconsistent BlockFront resuls causing unsafe operation.

We now use the rules to indicate that if not TransmissionOK then FailureMode is true and so all trains should be sent the STOP message, this results in the failsafe operation of the system as is vital in safety crital systems such as this:

- $\neg$ *TransmissionOk*$(t) \Rightarrow$ *FailureMode*(t)
- *FailureMode*(t)
  $\Rightarrow \forall$ *train* : *TrainID*
    - *MessagesToTrain*$(train,\ t + 1) = \{\} + STOP$
- *SafeMode*$(t) \Leftrightarrow \neg$ *FailureMode*(t)

%(Check transmission from all trains successful)%

6.5.2 seenSpeed|Position|Length and messagesToSpeed|Position|Length

Next we look at the rules for the the seenSpeed and messagesToSpeed operations. The purpose of the seenSpeed operation is to return the speed of the given train at the given time as it is perceived by the control system. It obtains this data using the messagesTospeed operation as shown below:

$\forall$ *train* : *TrainID*; *t* : *time*
- $\forall$ *m* : *Set*[*T_Message*]; *s* : *speed*
  - *messagesToSpeed*$(m) = s$
    $\Leftrightarrow$ *TRAINSPEED*(s) *eps* $m \wedge$ *TransmissionOk*(t)

%(Get speed data from train message TRAINSPEED)%
- *seenSpeed*$(train,\ t)$
  $=$ *messagesToSpeed*(*MessagesFromTrain*$(train,\ t)$)

%(Speed of train as perceived by Control System)%

These rules state that seenSpeed(train, t) is equal to the value returned by messagesToSpeed(MessagesFromT
MessagesToSpeed(MessagesFromTrain(train,t) returns the speed s if and only if TRAINSPEED(s) is an element of MessagesFromTrain(train,t) and TransmissionOk(t) meaning if there was an error in transmission of either no TRAINSPEED message or more than one TRAINSPEED message then no speed is reported, hence its definition as a partial function.

The same reasoning is then applied to the seenPosition and seenLength rules (Requirement 2):

- $\forall\ m : Set[T\_Message];\ p : position$
  - $messagesToPosition(m) = p$
    $\Leftrightarrow TRAINPOSITION(p)\ eps\ m \wedge TransmissionOk(t)$
    %(get position data from trainmessage TRAINPOSITION)%
- $seenPosition(train,\ t)$
  $= messagesToPosition(MessagesFromTrain(train,\ t))$
  %(Position of train as percieved by Control System)%
- $\forall\ m : Set[T\_Message];\ l : length$
  - $messagesToLength(m) = l$
    $\Leftrightarrow TRAINLENGTH(l)\ eps\ m \wedge TransmissionOk(t)$
    %(Get Length Data from train message TRAINLENGTH)%
- $seenLength(train,\ t)$
  $= messagesToLength(MessagesFromTrain(train,\ t))$
  %(Length of train as percieved by Control System)%

6.5.3 BlockFront, BlockFrontAcceled, BlockFrontDeceled

The purpose of the the BlockFrontAcceled and BlockFrontDeceled operations is to predict the position of the block front as if the train was one unit faster or one unit slower respectively. This added more flexibility in the changing of speeds. Upon analysis if the BlockFront did not overlap with the block of the train in front, but BlockFrontAcceled did, then rather than accelerate the train should continue with the same speed. If the BlockFront did over lap with the block of the train in front, we then analyse BlockFrontDeceled, if this value still overlaps then deceleration is not sufficient to bring the train out of a dangerous state so the train must break.

The rules for these operations are as follows:

- $BlockFront(train,\ t)$
  $= seenPosition(train,\ t)$
  $+ BrakingFunction(seenSpeed(train,\ t))$
  %(Block Front Calculation)%
- $BlockFrontAcceled(train,\ t)$
  $= seenPosition(train,\ t)$
  $+ BrakingFunction(seenSpeed(train,\ t) + 1)$
  %(Block Front of train as if accelerated)%
- $BlockFrontDeceled(train,\ t)$

$= seenPosition(train,\ t)$
$\quad + BrakingFunction(seenSpeed(train,\ t) - 1)$

%(Block Front of train as if decelerated)%

- $BlockRear(train,\ t)$
  $= seenPosition(train,\ t) - seenLength(train,\ t)$

  %(Block Rear Calculation)%

It can been seen that the returned value for each of the BlockFront. . . operations is the current position of the train plus the breakingdistance of the train given its current speed, or its current speed increased or decreased by one (Requirements 3). By rules later defined in the specification ControlSystemChain, we will be able to define that the should be no over lap between the blocks of two trains, and that the rear train of the two should act accordingly to return to a safe position. It may be necessary at a later stage to apply an error function to the block front and block rear to take into account discrepancies between the train's actual position and it's perceived position. Again due to the level of abstraction which specification allows such an error function was not required at the stage.

Validation In order to attempt to validate the specification of the control system a use case specification was developed at passed through SPASS. This however did not appear to work as none of the goals present in the use case in under 10 minutes without returning "SPASS error 1". Little other information on this error seems to be available however I expect it may be related to memory usage since the validation was attempted on a machine with fairly low memory.

The use case specification is shown below:

spec ControlSystemUSECASE =
ControlSystem
then ops $train1 : TrainID;$
$t : time$

- $MessagesFromTrain(train1,\ t)$
  $= ((\{\} + TRAINSPEED(5)) + TRAINPOSITION(2))$
  $\quad + TRAINLENGTH(1)$

then **%implies**

- $seenSpeed(train1,\ t) = 5$

  %(seenSpeed is equal to the speed of the train at time t)%
- $seenPosition(train1,\ t) = 2$

  %(seenSPosition is equal to the position of the train a time t)%
- $seenLength(train1,\ t) = 1$

  %(seenLength is equal to the length of the train at time t)%

- *BlockRear*(*train1*, t) = 1

  %(BlockRear is equal to train position − train length at time t)%
- *BlockFront*(*train1*, t) = 11

  %(BlockFront is equal to position of train plus the breaking function of the current speed)%
- *BlockFrontAcceled*(*train1*, t) = 14

  %(BlockFrontAcceled is equal to BlockFront as if speed inc by 1)%
- *BlockFrontDeceled*(*train1*, t) = 8

  %(BlockFrontDeceled is equal to BlockFront as if speed dec by 1)%

end

We can form the proof of the use case as follows:

seenSpeed

$$seenSpeed(train, t) = messagesToSpeed(messagesFromTrain(train, t)$$

We know:

$$MessagesOK(MessagesFromTrain(train, t)) = TRUE$$

and:

$$TRAINSPEED(5) \epsilon MessagesFromTrain(train, t)$$

Therefor:

$$seenSpeed(train, t) = 5 \qquad (6.5.3.1)$$

seePosition

$$seenPosition(train, t) = messagesToPosition(messagesFromTrain(train, t)$$

We know:

$$MessagesOK(MessagesFromTrain(train, t)) = TRUE$$

and:

$$TRAINSPOSITION(2) \epsilon MessagesFromTrain(train, t)$$

Therefor:

$$seenPosition(train, t) = 2 \qquad (6.5.3.2)$$

seenLength

$$\begin{aligned} seenLength(train, t) = messagesToLength(messagesFromTrain(train, t)\\ \text{We know:}\\ MessagesOK(MessagesFromTrain(train, t)) = TRUE\\ \text{and:}\\ TRAINLENGTH(1)\epsilon MessagesFromTrain(train, t)\\ \text{Therefor:}\\ seenLength(train, t) = 1 \end{aligned} \tag{6.5.3.3}$$

BlockRear

$$\begin{aligned} BlockRear(train, t) = seenPosition(train, t) - seenLength(train, t) =\\ 2 - 1 = 1 \end{aligned} \tag{6.5.3.4}$$

BlockFront

$$\begin{aligned} BlockRear(train, t) = seenPosition(train, t) + BrakingDist(seenSpeed(train, t)) =\\ 2 + (5 + BrakingDist(5 - 2)) = 5 + (3 + BrakingDist(3 - 2)) =\\ 2 + 5 + 3 + 1 = 11 \end{aligned} \tag{6.5.3.5}$$

BlockFrontAcceled

$$\begin{aligned} BlockRear(train, t) = seenPosition(train, t) + BrakingDist(seenSpeed(train, t) + 1) =\\ 2 + (6 + BrakingDist(6 - 2)) = 6 + (4 + BrakingDist(4 - 2)) =\\ 2 + 6 + 4 + 2 = 14 \end{aligned} \tag{6.5.3.6}$$

BlockFrontDecelled

$$\begin{aligned} BlockRear(train, t) = seenPosition(train, t) + BrakingDist(seenSpeed(train, t) - 1) =\\ 2 + (4 + BrakingDist(4 - 2)) = 2 + 4 + 2 = 8 \end{aligned} \tag{6.5.3.7}$$

6.6 Train Queue

The next specification we look at is that of a TrainQueue. The necessity of this specification is the need for a method of ordering trains within a control system (Requirement 5). Since trains on a length of track are unable to overtake (without moving onto an extra piece of track) then trains on a track resembles a queue of trains. So it was decided that a specific TrainQueue data structure would be specified to provide the first in first out features of a queue as well as operations required specifically by the control system. Once again in this specification we turn to the generated type notation to indicate that a Queue can only possibly be of a value formed by its constructor, In this case that a train is either empty of formed by adding a TrainID to an existing queue

The operations of a TrainQueue are:

- empty - represents the empty queue
- add - adds a train to a queue by its TrainID
- remove - removes the front element of the queue
- front - gives the TrainID of the front of the queue
- back - gives the trainID of the back of the queue
- queueLength - returns the length of the queue
- getTrain - returns the TrainID of the train at a given position in the queue

As well as the predicate:

- __elem__ - which returns true if a TrainID is an element of the queue

As mentioned previously in section4.2.2 it has been possible to build a proof of the train queue specification by using the automatic theorem prover SPASS, so we include all of the CASL specification for the TrainQueue (expanded from what was seen in section 4.2.1) below:

spec TRAINQUEUE =
  TRAIN
then generated type
  $Queue ::= empty \mid add(Queue; TrainID)$
  ops $add : Queue \times TrainID \rightarrow Queue;$
    $remove : Queue \rightarrow Queue;$

$front : Queue \rightarrow ?\ TrainID;$
$back : Queue \rightarrow ?\ TrainID;$
$queueLength : Queue \rightarrow Value;$
$getTrain : Queue \times Value \rightarrow ?\ TrainID$
pred $\_\_elem\_\_ : TrainID \times Queue$
$\forall\ q : Queue;\ t, t1, t2 : TrainID;\ v : Value$
• $remove(empty) = empty$ %(Remove op on the empty queue)%
• $remove(add(empty, t)) = empty$
%(Remove op on the queue with single element)%
• $remove(add(add(q, t1), t2)) = add(remove(add(q, t1)), t2)$
%(Remove op on arbitrary length queue)%
• $\neg\ def\ front(empty)$ %(Not def front of empty queue)%
• $front(add(empty, t)) = t$
%(Front of queue with single element)%
• $front(add(add(q, t1), t2)) = front(add(q, t1))$
%(Front of arbitrary length queue)%
• $\neg\ def\ back(empty)$ %(not def back of empty queue)%
• $back(add(q, t)) = t$ %(back of arbitrary length queue)%
• $\neg\ t\ elem\ empty$ %(Elem check on empty queue)%
• $\neg\ t1 = t2 \Rightarrow \neg\ t1\ elem\ add(empty, t2)$
%(t1 elem queue is false if t1 has not been added)%
• $t\ elem\ add(q, t)$ %(t elem queue is true if t has been added)%
• $\neg\ t1 = t2 \Rightarrow (t1\ elem\ add(q, t2) \Leftrightarrow t1\ elem\ q)$
%(elem check on arbitrary length queue)%
• $queueLength(empty) = 0$ %(Length of the empty queue)%
• $queueLength(add(q, t1)) = suc(queueLength(q))$
%(Length of arbirary length qeueue)%
• $\neg\ def\ getTrain(empty, v)$
%(Not def getTrain on empty queue)%
• $getTrain(add(q, t1), 0) = t1$
%(get most recently added train)%
• $getTrain(add(q, t1), suc(v)) = getTrain(q, v)$
%(get any added train)%

end

6.6.1 TrainQueue Validation

As mentioned previously it is possible to validate the TrainQueue specification automatically using SPASS. To do so we will use the following specification:

spec TRAINQUEUEUSECASE =

TRAINQUEUE
then ops $Train1, Train2, Train3 : TrainID;$
$queue : Queue$

- $\neg\ Train1 = Train2$ %(Train1 does not equal Train2)%
- $\neg\ Train1 = Train3$ %(Train1 does not equal Train3)%
- $\neg\ Train2 = Train3$ %(Train2 does not equal Train3)%
- $\neg\ queue = empty$ %(queue does not equal empty)%

then %implies

- $def\ Train1$ %(Train1 is always defined)%
- $def\ Train2$ %(Train2 is always defined)%
- $\neg\ front(empty) = Train1$
  %(ensure front of the empty list returns undefined)%
- $\neg\ back(empty) = Train1$
  %(ensure back of the empty list returns undefined)%
- $front(add(add(add(empty, Train1), Train2), Train3)) = Train1$
  %(front on a queue with three elements returns the front element)%
- $front(add(add(empty, Train1), Train2)) = Train1$
  %(front returns the first element added to the list)%
- $back(add(empty, Train1)) = Train1$
  %(back on a queue with a single element returns that element)%
- $back(add(add(queue, Train1), Train2)) = Train2$
  %(back always returns the last element added to the queue)%
- $remove(add(add(empty, Train1), Train2))$
  $= add(empty, Train2)$
  %(remove returns the queue with the least recently added item removed)%
- $remove(add(add(add(empty, Train1), Train2), Train3))$
  $= add(add(empty, Train2), Train3)$
  %(remove returns the queue with the least recently added item removed with >2 elements)%
- $\neg\ Train1\ elem\ empty$
  %(Train1 is not an element of the empty queue)%
- $\neg\ Train1\ elem\ add(empty, Train2)$
  %(Train1 is not an element of a queue containing only Train2)%
- $Train1\ elem\ add(queue, Train1)$
  %(Train1 is an element of a queue when it is the most recent added element)%
- $Train2\ elem\ add(add(add(queue, Train1), Train2), Train3)$
  %(Train2 is an element of a queue when it is the middle added element to the queue)%
- $queueLength(add(empty, Train1)) = 1$
  %(queueLength on a trainqueue with 1 elements returns 1)%
- $queueLength(add(add(add(empty, Train1), Train2), Train3))$
  $= 3$
  %(queueLength on a trainQueue with 3 elements returns 3)%
- $\neg\ getTrain(empty, 0) = Train1$
  %(getTrain on the empty list returns undefined)%
- $getTrain(add(empty, Train1), 0) = Train1$

%(element 0 of the queue with one element returns that element)%

- *getTrain*(*add*(*queue*, *Train1*), 0) = *Train1*

%(element 0 of any queue returns the most recent element)%

- *getTrain*(*add*(*add*(*add*(*queue*, *Train1*), *Train2*), *Train3*), 1)
  = *Train2*

%(getTrain can retrieve the second latest added train from the queue)%

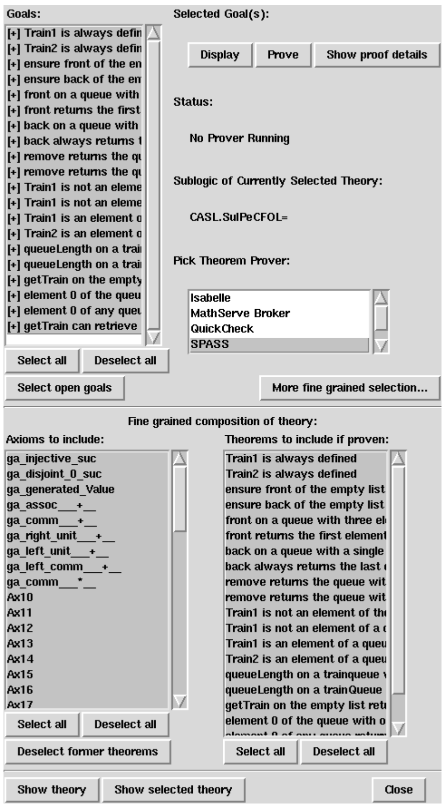
end

The automatic theorem prover will try to prove each of the goals according to the axioms laid out in the TrainQueue specification. The figure 6.3 shows the prover interface after running the prover. The "[+]" beside each goal indicates that each goal has been proven. Appendix C contains the details of the proof tree, an example of which is shown below:

```
---------------------------------------------------------------------------
front returns the first element added to the list
    Com: id_CASL.SulPeCFOL=;CASL2SubCFOL;CASL2SoftFOL : CASL -> SoftFOL
        Status: Proved
        Used axioms: "ga_totality_24", "Train1 is always defined",
                     "Train2 is always defined", "Front of queue with single element",
                     "Front of arbitrary length queue"
        Used time: 00:00:01.949999999999
        Prover: SPASS
        Tactic script
        Proof tree

---------------------------------------------------------------------------
```

This shows the proof for the "*front returns the first element added to the list*" goal. It indicates its current status, the axioms used to reach its goal and other details including the time taken to reach the proof and the prover used.

For full details of the proof please see appendix C



```
Goals:
[+] Train1 is always defin
[+] Train2 is always defin
[+] ensure front of the en
[+] ensure back of the en
[+] front on a queue with
[+] front returns the first
[+] back on a queue with
[+] back always returns t
[+] remove returns the qu
[+] remove returns the qu
[+] Train1 is not an eleme
[+] Train1 is not an eleme
[+] Train1 is an element o
[+] Train2 is an element o
[+] queueLength on a trai
[+] queueLength on a trai
[+] getTrain on the empty
[+] element 0 of the queu
[+] element 0 of any queu
[+] getTrain can retrieve

Selected Goal(s):

Status:
  No Prover Running

Sublogic of Currently Selected Theory:
  CASL.SulPeCFOL=

Pick Theorem Prover:
  Isabelle
  MathServe Broker
  QuickCheck
  SPASS

Fine grained composition of theory:

Axioms to include:
ga_injective_suc
ga_disjoint_0_suc
ga_generated_Value
ga_assoc___+__
ga_comm___+__
ga_right_unit___+__
ga_left_unit___+__
ga_left_comm___+__
ga_comm___*__
Ax10
Ax11
Ax12
Ax13
Ax14
Ax15
Ax16
Ax17

Theorems to include if proven:
Train1 is always defined
Train2 is always defined
ensure front of the empty list
ensure back of the empty list
front on a queue with three el
front returns the first element
back on a queue with a single
back always returns the last
remove returns the queue wit
remove returns the queue wit
Train1 is not an element of th
Train1 is not an element of a
Train1 is an element of a queu
Train2 is an element of a queu
queueLength on a trainqueue
queueLength on a trainQueue
getTrain on the empty list ret
element 0 of the queue with o
```

Figure 6.3: Spass Prover interface showing all goals of ProofQueue proven

6.7 ControlSystemChain

This is the last specification of the moving block system. Since the moving block system we have specified contains an infinite length of track we will take into account that there may be multiple control systems spanning the length of the track. Each control system will have a control system it perceives as being next, and one it perceives as previous. Each control system will also contain positions of its in boundary and out boundary. All trains present on its tracks as stored in the train queue in the order which they came into the control systems region of control.

The ControlSystemChain specification contains the following Operations:

- NextControlSystem - returns the next control system in the chain (if there is one)
- PreviousControlSystem - returns the previous control system in the chain (if there is one)
- TrainQueue - returns the queue of trains on the track covered by the control system

- InBoundary - returns the position of the start of the control system's track
- OutBoundary - returns the position of the end of the control system's track

As well as the following predicates:

- Accel - True if train should accelerate
- Proceed - True if train should proceed
- Break - True if train should break

6.7.1 Accel, Proceed and Break predicates

These three predicates are used in the determination of whether a train should break, decelerate, continue at current speed or accelerate. The outcome of these predicates is given by the axioms below:

- $\mathit{Proceed}(\mathit{train}, \mathit{train1}, t)$
  $\Leftrightarrow \mathit{BlockFront}(\mathit{train}, t) < \mathit{BlockRear}(\mathit{train1}, t)$
  %(Proceed if safe too)%
- $\mathit{Accel}(\mathit{train}, \mathit{train1}, t)$
  $\Leftrightarrow \mathit{Proceed}(\mathit{train}, \mathit{train1}, t)$
  $\wedge\ \mathit{BlockFrontAcceled}(\mathit{train}, t) < \mathit{BlockRear}(\mathit{train1}, t)$
  %(Accelerate if can proceed and safe to accelerate)%
- $\mathit{Break}(\mathit{train}, \mathit{train1}, t)$
  $\Leftrightarrow \neg\ \mathit{Proceed}(\mathit{train}, \mathit{train1}, t)$
  $\wedge\ \mathit{BlockRear}(\mathit{train1}, t) < \mathit{BlockFrontDeceled}(\mathit{train}, t)$
  %(Break if decelerating not sufficient)%

The purpose of these rules is to reduce repeon of statements in later axioms thus reducing error.

- Proceed only evaluates to true if the BlockFront of the first train parameter is not greater that the BlockRear of the second train parameter. This is the only condition under which is it safe to proceed.
- Accel only evaluates true if Proceed evaluates to true (meaning it is safe to proceed), and the BlockFrontAcceled of the first train is not greater than the BlockRear of the second train, thus meaning that accelerating will not put the train into a dangerous position.

- Break only evaluates to true if Proceed on the two trains is false (thus meaning the train must reduce speed), and if the BlokFrontDeceled for the first train is greater than the BlockRear of the second train, thus meaning that decelerating it insufficient to bring the train out of the dangerous position and so should break.
- By these rules the deceleration only occurs if not Proceed and not Break.
- by the axiom for Break, it is impossible for Proceed AND Break to be TRUE simultaneously

6.7.2 Action to Take Analysis

Now that we have given the axioms for the predicates above we are able to define the axioms of how the train should behave given its position relative to the train in front. We first define the axioms determining the behaviour of the train at the front of the TrainQueue. This train can be referred to as a special case since its block front is first compared with the OutBoundary of the control system currently controlling it. Only if the block front is greater than the OutBoundary is the block front compared with the BlockRear of the back train of the TrainQueue of the next control system.

The axioms controlling the MessagesToTrain for the front of the TrainQueue are:

- $\mathit{BlockFront}(\mathit{front}(\mathit{TrainQueue}(\mathit{cs1},\ t)),\ t)$
  $< \mathit{InBoundary}(\mathit{NextControlSystem}(\mathit{cs1}))$
  $\Rightarrow \mathit{MessagesToTrain}(\mathit{front}(\mathit{TrainQueue}(\mathit{cs1},\ t)),\ t + 1)$
  $= \{\} + \mathit{ACCELERATE}$
  %(if block of front train does not overlap with next control system then accelerate)%
- $\mathit{InBoundary}(\mathit{NextControlSystem}(\mathit{cs1}))$
  $< \mathit{BlockFront}(\mathit{front}(\mathit{TrainQueue}(\mathit{cs1},\ t)),\ t)$
  $\wedge \neg\ \mathit{Proceed}(\mathit{front}(\mathit{TrainQueue}(\mathit{cs1},\ t)),$
  $\mathit{back}(\mathit{TrainQueue}(\mathit{NextControlSystem}(\mathit{cs1}),$
  $t)),$
  $t)$
  $\wedge \neg\ \mathit{Break}(\mathit{front}(\mathit{TrainQueue}(\mathit{cs1},\ t)),$
  $\mathit{back}(\mathit{TrainQueue}(\mathit{NextControlSystem}(\mathit{cs1}),\ t)),$
  $t)$
  $\Rightarrow \mathit{MessagesToTrain}(\mathit{front}(\mathit{TrainQueue}(\mathit{cs1},\ t)),\ t + 1)$
  $= \{\} + \mathit{DECELERATE}$
  %(DECELERATE MessageToTrain calc for front train with CS overlap)%
- $\mathit{InBoundary}(\mathit{NextControlSystem}(\mathit{cs1}))$
  $< \mathit{BlockFront}(\mathit{front}(\mathit{TrainQueue}(\mathit{cs1},\ t)),\ t)$
  $\wedge \neg\ \mathit{Proceed}(\mathit{front}(\mathit{TrainQueue}(\mathit{cs1},\ t)),$

$back(TrainQueue(NextControlSystem(cs1),$
$t)),$
$t)$
$\wedge\ Break(front(TrainQueue(cs1,\ t)),$
$back(TrainQueue(NextControlSystem(cs1),\ t)),\ t)$
$\Rightarrow MessagesToTrain(front(TrainQueue(cs1,\ t)),\ t+1)$
$= \{\} + BREAK$
%(BREAK MessageToTrain calc for front train with CS overlap)%

- $InBoundary(NextControlSystem(cs1))$
  $< BlockFront(front(TrainQueue(cs1,\ t)),\ t)$
  $\wedge\ Proceed(front(TrainQueue(cs1,\ t)),$
  $back(TrainQueue(NextControlSystem(cs1),\ t)),$
  $t)$
  $\wedge\ \neg\ Accel(front(TrainQueue(cs1,\ t)),$
  $back(TrainQueue(NextControlSystem(cs1),\ t)),$
  $t)$
  $\Rightarrow MessagesToTrain(front(TrainQueue(cs1,\ t)),\ t+1)$
  $= \{\} + CONTINUE$
  %(CONTINUE MessageToTrain calc for front train with CS overlap)%
- $InBoundary(NextControlSystem(cs1))$
  $< BlockFront(front(TrainQueue(cs1,\ t)),\ t)$
  $\wedge\ Proceed(front(TrainQueue(cs1,\ t)),$
  $back(TrainQueue(NextControlSystem(cs1),\ t)),$
  $t)$
  $\wedge\ Accel(front(TrainQueue(cs1,\ t)),$
  $back(TrainQueue(NextControlSystem(cs1),\ t)),\ t)$
  $\Rightarrow MessagesToTrain(front(TrainQueue(cs1,\ t)),\ t+1)$
  $= \{\} + ACCELERATE$
  %(ACCELERATE MessageToTrain calc for front train with CS overlap)%

These axioms make use of the Accel, Proceed and Break predicates, an important note here is that the rear train of the two trains being compared is always passed as the first parameter to the predicates.

The following axioms define how the other trains in the TrainQueue should behave:

- $\forall\ n : Value$
  - $\neg\ Proceed(getTrain(TrainQueue(cs1,\ t),\ n),$
    $getTrain(TrainQueue(cs1,\ t),\ n+1),\ t)$
    $\wedge\ \neg\ Break(getTrain(TrainQueue(cs1,\ t),\ n),$
    $getTrain(TrainQueue(cs1,\ t),\ n+1),\ t)$
    $\Rightarrow MessagesToTrain(getTrain(TrainQueue(cs1,\ t),\ n),$

```
                        t + 1)
        = {} + DECELERATE
                              %(DECELERATE MessageToTrain calc for all other trains)%
  • ∀ n : Value
    • ¬ Proceed(getTrain(TrainQueue(cs1, t), n),
                   getTrain(TrainQueue(cs1, t), n + 1), t)
      ∧ Break(getTrain(TrainQueue(cs1, t), n),
                 getTrain(TrainQueue(cs1, t), n + 1), t)
      ⇒ MessagesToTrain(getTrain(TrainQueue(cs1, t), n),
                        t + 1)
        = {} + BREAK
                                   %(BREAK MessageToTrain calc for all other trains)%
  • ∀ n : Value
    • Proceed(getTrain(TrainQueue(cs1, t), n),
                 getTrain(TrainQueue(cs1, t), n + 1), t)
      ∧ Accel(getTrain(TrainQueue(cs1, t), n),
                 getTrain(TrainQueue(cs1, t), n + 1), t)
      ⇒ MessagesToTrain(getTrain(TrainQueue(cs1, t), n),
                        t + 1)
        = {} + ACCELERATE
                              %(ACCELERATE MessageToTrain calc for all other trains)%
  • ∀ n : Value
    • Proceed(getTrain(TrainQueue(cs1, t), n),
                 getTrain(TrainQueue(cs1, t), n + 1), t)
      ∧ ¬ Accel(getTrain(TrainQueue(cs1, t), n),
                   getTrain(TrainQueue(cs1, t), n + 1), t)
      ⇒ MessagesToTrain(getTrain(TrainQueue(cs1, t), n),
                        t + 1)
        = {} + CONTINUE
                                %(CONTINUE MessageToTrain calc for all other trains)%
```

The behaviour of these axioms is again defined using the predicates Accel, Proceed and Break in the following way:

- IF Proceed AND Accel THEN ACCELERATE (Requirement 7.1)
- IF Proceed AND not Accel THEN CONTINUE (Requirement 7.2)
- IF not Proceed AND not Break THEN DECELERATE (Requirement 7.3)
- IF Not Proceed AND Break THEN BREAK (Requirement 7.4)

This caters for the four of the five possible C_Message that can be passed to a train, the other (STOP) already has its axiom defined in the specification ControlSystem

Upon closer inspection of these axioms it can be seen that the first parameter passed to these predicates is always the current train analysed and the second parameter passed is always the train in front in the queue since the getTrain operation of a TrainQueue numbers the the elements of the queue from back being 0 to queueLength of the queue.

6.7.3 Movement of Trains Between Control Systems

In order for trains to be able to pass from one control system to another we include the following axiom:

- $InBoundary(cs1) < Position(train, t)$
  $\wedge$ $train\ elem\ TrainQueue(PreviousControlSystem(cs1), t)$
  $\wedge \neg$ $train\ elem\ TrainQueue(cs1, t)$
  $\Rightarrow TrainQueue(cs1, t + 1) = add(TrainQueue(cs1, t), train)$
  $\wedge\ TrainQueue(PreviousControlSystem(cs1), t + 1)$
  $= remove(TrainQueue(PreviousControlSystem(cs1), t))$
  %(Pass train to next control system when passing boundary)%

This axiom states that:

- IF the position of a given train at a given time is greater than the InBoundary of the control system
- AND the train is an element of the previous control system's TrainQueue
- AND the train is not already an element of the current control system's TrainQueue
- THEN
  - add the train to the TrainQueue of the current control system
  - remove the train from the TrainQueue of the previous control system

  the add the train to the TrainQueue of the current control system

This axiom then meets requirement 8, that a train is passed to the next control system when passing the out boundary.

7 Conclusion

The objectives of this project was to investigate the usability of the common algebraic specification language (CASL) for the specification of larger industrial problems. To demonstrate this I have attempted to specify a simplified version of the moving block system of railway interlocking as it is planned for level 3 of the ERTMS. This is especially useful since the main area of engineering in which formal methods and specification is used is in the realm of critical systems.

In carrying out the specification a few obstacles were met along the way. My previous experience with system specification was through a module on the subject taught with practical examples using the specification language Maude. While holding syntactic similarities with Maude, CASL is also very different, and is considered by many to be a more versatile language. My inexperience in the field of system specification lead to difficulties especially in the area of abstraction. It was a difficult task having come from a programming background, to train my mind to think statically about the behaviour of the system rather than procedurally. This problem of procedural thinking also made it difficult to envision how to model the flow of control, this was overcome however by use of the stream method. It is possibly worth investigating whether specification languages exist that better facilitate the modeling of time such as the process algebra CSP or possibly the use of languages such as Spark Ada which, through the use of Hoare logic allow the expected inputs and outputs of procedures and functions to be defined. This would allow a specifier to both think procedurally of the problem, use flow control methods such as loops, as well as specify expected outputs for all possible inputs.

While initially these obstacles lead me to believe that specification of such a system was infeasible at best, impossible at worst, through increasing knowledge of system specification and added practice in the area through specifying VALUE, MyGenerateSet and the TrainQueue, I gained the knowledge required to build the model submitted with this project.

When analysing and validating the specification against requirements attempts were made to use automatic theorem provers by developing use case specifications. This approach was met with mixed success, with the proof for the TrainQueue being built in a relatively short time, while the use case for a control system, while simple enough to deduce by hand, could not be analysed in under 10 minutes per goal. It has been suggested that this may be because automatic theorem provers are developed with the state method of modeling in mind rather than the stream method. Since the use case of TrainWithMessages has similar issues and also makes use of the stream method while the specification of TrainQueue does not, this may be the case and warrants further investigation.

Despite the successes of this specification I still feel it is reasonable to question the use of specification as means of a design for a system rather than a carefully written design document written in standard English with a well defined structure. Computer aided software engineering (CASE) systems exist to facilitate the writing of such documents to such a degree that they strongly encourage software developers to think very closely about the system, and while lacking the verification options of formal methods the output of such systems is more easily read than the specification output from this project, which I feel would not be well accepted by programmers without training and knowledge in the field of system specification however it may be worth formally investigating this further with experienced programmers. I do on the other hand feel that system specification is very useful while the system is still in a conceptual stage as the extra level of abstraction means the designer does not need to be concerned with implementation issues such as program language choice and hardware requirements.

While it has been possible to model the flow of time in the system, an area that the specification is still severely lacking is in that of units of speed, distance and time. With the current specification it is unclear as to how much time flows in one arbitrary time unit, and how far a train realistically travels in a given time. This is clearly an area for expansion of the specification. Another area of expansion possible is adding more features of the railway, such as crossings, points, and different forms of track as well as track properties such as gradient and speed restrictions however to do so in this project would have made it unfeasible due to project deadlines and specification being a time consuming exercise.

Having carried out the specification from scratch I would have rather I had carried out a few smaller practice examples before attempting to specify the moving block system as I feel this would have better facilitated my learning of CASL and specification techniques rather than jumping in at such a large system. I do however feel comfortable that the specification produced as the outcome of the project can be reused and extended for further investigation into the specification of the moving block system of railway interlocking and as a positive experience in the specification of industrial problems using CASL.

Appendices

A Informal Requirements

A.1 Introduction and Description

Before we can begin specification we first must build an informal requirements document to know what we want the system to do. The system we will be specifying is a simplified version of the moving block system

Physical Description of the System

Physical Units The system will consist of the following types of physical units

- The control system
- An infinite length of track, this track will be broken into segments through the use of chain of control systems, each covering a segment of track
- Trains, identified using a TrainID

More physical units will be added in future iterations such as points and crossings.

The Control System The control system is responsible for safe operation of the track segment covered by it. It is a hybrid batch real time system in that while it requests all data from all trains under its coverage at the same time, it does so at each clock cycle which is fast enough for it to appear to be real time.

at each time interval the control system will:

1. Requirement 1: Receive the data required from trains to calculate safe operation (position, speed and length).
2. Requirement 2: Predict the state of the railway from this data.
3. Requirement 3: Calculate the block boundaries of every train on its track according to breaking distance.

4. Requirement 4: Send each train the corresponding signal (STOP, BREAK, DECELERATE, CONTINUE or ACCELERATE) depending on its current position.

Requirement 5: The control system will store the trains present on its track in a queue like data structure

Requirement 6: If an error in transmission of messages occurs all trains will stop, entering a fail safe state

Requirement 7: Assuming no errors occur in transmission of errors the control system will calculate the block boundaries (the front of the block and the rear of the block) for each train along its track in the follwoing manner:

- Requirement 7.1: If the block of the train does not over lap with the block of the train in front and will not overlap if the train accelerates then the train may accelerate.
- Requirement 7.2: If the block of the train does not over lap with the block of the train in front and will overlap if the train accelerates then the train should continue at the same speed
- Requirement 7.3: If the block of the train does over lap with the block of the train in front but will not overlap after deceleration then the train should decelerate (see requirement 9.3)
- Requirement 7.4: If the block of the train does over lap with the block of the train in front and will continue to do so after decelerating then the train should break (see requirement 9.4)

Requirement 8: The Control system must be able to pass a train to the next control system when it exits its segment of track

The Trains The first system iteration will include identical trains, consisting of the following behaviour:

- Requirement 9.1: The speed of the train at a given time
- Requirement 9.2: To have a constant maximum speed
- Requirement 9.3: When decelerating its speed is reduced by 1 unit speed
- Requirement 9.4: When breaking its speed is reduced by 2 unit speed
- Requirement 9.5: An acceleration of 1 unit per unit time

- Requirement 9.6: A position along the track calculated by speed and time
- Requirement 9.7: report the length of the train
- Requirement 9.8: A method of sending data to the control system
- Requirement 9.9: A method of receiving messages from the control system

System Messages

Train Messages Requirement 10: The messages passed from the trains to the control system are:

- Requirement 10.1: TRAINSPEED(speed): This will tell the control system the current speed of the train at a given time. From this the control system will carry out operations to calculate the block boundries for the train.
- Requirement 10.2: TRAININPOSITION(bool): This will tell the control system the current position of the train along the track
- Requirement 10.3: TRAINLENGTH(length): This will tell the control system the length of the train so that it my calculate the back position of the trains block. Will be constant.

A train can only send one form of each message at a given time. If more than one of a message type is detected for a given train then the control system enters a falure state and all trains are sent the STOP message

Control Messages

- Requirement 11.1: STOP: Tells the recipient of the message it should stop. Is sent if a transmission error has occured.
- Requirement 11.2: ACCELERATE - Increase the speed of the train by 1 up to its maximum speed
- Requirement 11.3: CONTINUE - continue at current speed
- Requirement 11.4: DECELERATE - decrease the speed of the train by 1 to a minimum of 0
- Requirement 11.5: BREAK - decrease the speed of the train by 2 to a minimum of 0

Values, Variables, and Constants Currently, values used within the system are all arbitrary, however since in the physical sense speed is an operation of distance over time there needs to be some sort of scaling.

A position is an arbitrary value along the piece of track s.t $0 <= position < lengthoftrack$ and will essentially be the distance the train has traveled since entering the track. Due to the speed of operation of the system the speed will be of unit value/time which is the distance the train travels per clock cycle.

For this first iteration of the system speed will be capped at a maximum value of 100 and will increase and decrease by 1 when required. Later breaking and acceleration curves may be built in

In future iterations speed will be a unit of meters/second and position, the length of the train and the length of the track will be measured in meters though this is subject to change if a higher degree of accuracy is required for safe operation.

B Specification of the Moving Block System

B.1 Train Library

library TrainLibrary

spec VALUE =
  free type *Value* ::= 0 | *suc*(*Value*)
  ops $\_\_+\_\_ : Value \times Value \to Value$, *assoc*, *comm*, *unit* 0;
    $\_\_-\_\_ : Value \times Value \to Value$;
    $\_\_*\_\_ : Value \times Value \to Value$, *comm*;
    %%Operations to represent the natural numbers as digits
    $1 : Value = suc(0)$;
    $2 : Value = suc(1)$;
    $3 : Value = suc(2)$;
    $4 : Value = suc(3)$;
    $5 : Value = suc(4)$;
    $6 : Value = suc(5)$;
    $7 : Value = suc(6)$;
    $8 : Value = suc(7)$;
    $9 : Value = suc(8)$;
    $\_\_@@\_\_(x : Value; y : Value) : Value$
    $= (x * suc(9)) + y$
  pred $\_\_<\_\_ : Value \times Value$
  $\forall\ x, y, z : Value$
  • $0 + x = x$ %(Value add zero)%
  • $suc(x) + y = suc(x + y)$ %(Value Addition)%
  • $0 - x = 0$ %(Zero sub Value)%
  • $suc(x) - 0 = suc(x)$ %(Value sub zero)%
  • $suc(x) - suc(y) = x - y$ %(Value Subtraction)%
  • $x * 0 = 0$ %(Value mult zero)%
  • $x * suc(y) = (x * y) + x$ %(Value Multiplication)%
  • $0 < suc(x)$ %(0 is the lowest Value)%
  • $\neg\ suc(x) < 0$ %(No Value less than 0)%
  • $suc(x) < suc(y) \Leftrightarrow x < y$ %(Value less than def)%
end

spec MyGenerateSet[**sort** *Elem*] = **%mono**
  generated type
  $Set[Elem] ::= \{\} \mid \_\_+\_\_(Set[Elem]; Elem)$
  pred $\_\_eps\_\_ : Elem \times Set[Elem]$
  $\forall\ x, y : Elem; M, N : Set[Elem]$

• $\neg\ x\ eps\ \{\}$ %(elemOf_empty_Set)%

• $x\ eps\ M + y \Leftrightarrow x = y \vee x\ eps\ M$

%(elemOf_NonEmpty_Set)%

• $M = N \Leftrightarrow \forall\ x : Elem \bullet x\ eps\ M \Leftrightarrow x\ eps\ N$ %(equality_Set)%

end

B.2 Moving Block Interlocking

library LIBRARY

%number __@@__

from TRAINLIBRARY **get** VALUE

from TRAINLIBRARY **get** MYGENERATESET

spec TRAIN =
VALUE
then sorts $TrainID$;
$speed = Value$;
$time = Value$;
$position = Value$;
$length = Value$;
$brakingdist = Value$
ops $Speed : TrainID \times time \rightarrow speed$;
$MaxSpeed : TrainID \rightarrow speed$;
$Position : TrainID \times time \rightarrow position$;
$Length : TrainID \rightarrow length$;
$BrakingDist : TrainID \times time \rightarrow brakingdist$;
$BrakingFunction : speed \rightarrow Value$

$\forall\ train : TrainID;\ s : speed;\ t : time;\ p : position;\ l : length$

• $Position(train,\ suc(t))$
$= Position(train,\ t) + Speed(train,\ t)$

%(Train Position calculation)%

• $BrakingDist(train,\ t) = BrakingFunction(Speed(train,\ t))$

%(Breaking distance is braking function of the speed at time t)%

• $s < 2 \Rightarrow BrakingFunction(s) = s$

%(Breaking function = s if s < 2)%

• $2 < s$
$\Rightarrow BrakingFunction(s) = s + BrakingFunction(s - 2)$

%(Braking function = s + brakingfunction after braking)%

end

```
spec TRAIN_MESSAGES =
     TRAIN
then free type
     T_Message
     ::= TRAINPOSITION(position)
       | TRAINSPEED(speed)
       | TRAINLENGTH(length)
end

spec CONTROL_MESSAGES =
     VALUE
then free type
     C_Message
     ::= CONTINUE
       | BREAK
       | DECELERATE
       | ACCELERATE
       | STOP
end

spec MESSAGES =
     MYGENERATESET[TRAIN_MESSAGES fit Elem ↦ T_Message]
and  MYGENERATESET
     [CONTROL_MESSAGES fit Elem ↦ C_Message]
end

spec TRAINWITHMESSAGES =
     MESSAGES
then ops   MessagesFromTrain
             : TrainID × time → Set[T_Message];
           MessagesToTrain : TrainID × time → Set[C_Message]
     preds MessagesOk : Set[C_Message];
           TransmissionOk : TrainID × time
     ∀ train : TrainID; s : speed; t : time; p : position; l : length
     • MessagesFromTrain(train, t)
       = (({} + TRAINSPEED(Speed(train, t)))
          + TRAINPOSITION(Position(train, t)))
         + TRAINLENGTH(Length(train))
                                        %(Messages sent to control System calc)%
     • ∀ m : Set[C_Message]
       • MessagesOk(m)
         ⇔ ∃! message : C_Message • message eps m
                          %(Check that a unique message was recived from control system)%
```

- $\mathit{TransmissionOk}(\mathit{train}, t)$
  $\Leftrightarrow \mathit{MessagesOk}(\mathit{MessagesToTrain}(\mathit{train}, t))$
- $\mathit{DECELERATE}\ \mathit{eps}\ \mathit{MessagesToTrain}(\mathit{train}, t)$
  $\wedge \neg\ \mathit{Speed}(\mathit{train}, t) = 0 \wedge \mathit{TransmissionOk}(\mathit{train}, t)$
  $\Rightarrow \mathit{Speed}(\mathit{train}, t + 1) = \mathit{Speed}(\mathit{train}, t) - 1$
  %(Messages to train DECELERATE evaluation)%
- $\mathit{DECELERATE}\ \mathit{eps}\ \mathit{MessagesToTrain}(\mathit{train}, t)$
  $\wedge\ \mathit{Speed}(\mathit{train}, t) = 0 \wedge \mathit{TransmissionOk}(\mathit{train}, t)$
  $\Rightarrow \mathit{Speed}(\mathit{train}, t + 1) = 0$
  %(Messages to train DECELERATE evaluation at stop)%
- $\mathit{BREAK}\ \mathit{eps}\ \mathit{MessagesToTrain}(\mathit{train}, t)$
  $\wedge \neg\ \mathit{Speed}(\mathit{train}, t) < 3 \wedge \mathit{TransmissionOk}(\mathit{train}, t)$
  $\Rightarrow \mathit{Speed}(\mathit{train}, t + 1) = \mathit{Speed}(\mathit{train}, t) - 2$
  %(Messages to train BREAK evaluation)%
- $\mathit{BREAK}\ \mathit{eps}\ \mathit{MessagesToTrain}(\mathit{train}, t)$
  $\wedge\ \mathit{Speed}(\mathit{train}, t) < 3 \wedge \mathit{TransmissionOk}(\mathit{train}, t)$
  $\Rightarrow \mathit{Speed}(\mathit{train}, t + 1) = 0$
  %(Messages to train BREAK evaluation at near stop)%
- $\mathit{ACCELERATE}\ \mathit{eps}\ \mathit{MessagesToTrain}(\mathit{train}, t)$
  $\wedge \neg\ \mathit{Speed}(\mathit{train}, t) = \mathit{MaxSpeed}(\mathit{train})$
  $\wedge\ \mathit{TransmissionOk}(\mathit{train}, t)$
  $\Rightarrow \mathit{Speed}(\mathit{train}, t + 1) = \mathit{Speed}(\mathit{train}, t) + 1$
  %(Messages to Train ACCELERATE evaluation)%
- $\mathit{ACCELERATE}\ \mathit{eps}\ \mathit{MessagesToTrain}(\mathit{train}, t)$
  $\wedge\ \mathit{Speed}(\mathit{train}, t) = \mathit{MaxSpeed}(\mathit{train})$
  $\wedge\ \mathit{TransmissionOk}(\mathit{train}, t)$
  $\Rightarrow \mathit{Speed}(\mathit{train}, t) = \mathit{Speed}(\mathit{train}, t)$
  %(Messages to Train ACCELERATE evaluation at max speed)%
- $\mathit{CONTINUE}\ \mathit{eps}\ \mathit{MessagesToTrain}(\mathit{train}, t)$
  $\wedge\ \mathit{TransmissionOk}(\mathit{train}, t)$
  $\Rightarrow \mathit{Speed}(\mathit{train}, t + 1) = \mathit{Speed}(\mathit{train}, t)$
  %(Messages to Train CONTINUE evaluation)%
- $\mathit{STOP}\ \mathit{eps}\ \mathit{MessagesToTrain}(\mathit{train}, t)$
  $\wedge\ \mathit{TransmissionOk}(\mathit{train}, t)$
  $\Rightarrow \mathit{Speed}(\mathit{train}, t + 1) = 0$
  %(Messages to Train STOP evaluation)%

end

spec TrainwithMessagesUSECASE =
TrainWithMessages
then op *train1* : *TrainID*
%**initialisation**, speed 9 position 0 at time 0%
- $\mathit{Speed}(\mathit{train1}, 0) = 9$
- $\mathit{Position}(\mathit{train1}, 0) = 0$

- $MessagesToTrain(train1, 0) = \{\} + ACCELERATE$
- $MessagesToTrain(train1, 1) = \{\} + CONTINUE$
- $MessagesToTrain(train1, 2) = \{\} + STOP$
- $MessagesToTrain(train1, 3) = \{\} + ACCELERATE$
- $MessagesToTrain(train1, 4) = \{\} + ACCELERATE$
- $MessagesToTrain(train1, 5) = \{\} + ACCELERATE$
- $MessagesToTrain(train1, 6) = \{\} + ACCELERATE$
- $MessagesToTrain(train1, 7) = \{\} + DECELERATE$
- $MessagesToTrain(train1, 8) = \{\} + BREAK$

then %implies

- $Position(train1, 1) = 9$
  %(The position at time 1 is equal to the position at time 0 plus the speed at time 0)%
- $Speed(train1, 1) = 10$
  %(the speed of the train increases by 1 if the train recieves an accelerate message)%
- $Position(train1, 2) = 19$
  %(the position of the train at time 2 is equal to the position at time 1 + the position)%
- $Speed(train1, 2) = 10$
  %(The Train stops if it recieves a stop signal)%
- $Position(train1, 3) = 29$
  %(The train does not move when speed is zero)%
- $Speed(train1, 3) = 0$
  %(The train is able to start again once stopped with an accelerate signal)%
- $Speed(train1, 7) = 4$ %(The speed at time 7 = 4)%
- $Speed(train1, 8) = 3$ %(The speed has been reduced by 1)%
- $Speed(train1, 9) = 1$ %(The speed has been reduced by 1)%

end

spec CONTROLSYSTEM =
MESSAGES
then ops *MessagesFromTrain*
: $TrainID \times time \to Set[T\_Message]$;
MessagesToTrain
: $TrainID \times time \to Set[C\_Message]$;
$BlockFront : TrainID \times time \to position$;
$BlockFrontAcceled : TrainID \times time \to position$;
$BlockFrontDeceled : TrainID \times time \to position$;
$BlockRear : TrainID \times time \to position$;
$seenSpeed : TrainID \times time \to ? speed$;
$seenPosition : TrainID \times time \to ? position$;
$seenLength : TrainID \times time \to ? length$;
$messagesToSpeed : Set[T\_Message] \to ? speed$;
$messagesToPosition : Set[T\_Message] \to ? position$;
$messagesToLength : Set[T\_Message] \to ? length$
preds $FailureMode : time$;

```
        $SafeMode : time;$
        $TransmissionOk : time;$
        $MessagesOK : Set[T\_Message]$
  $\bullet\ \forall\ m : Set[T\_Message]$
    $\bullet\ MessagesOK(m)$
      $\Leftrightarrow \exists!\ s : speed;\ p : position;\ l : length$
        $\bullet\ TRAINSPEED(s)\ eps\ m$
          $\wedge\ TRAINPOSITION(p)\ eps\ m$
          $\wedge\ TRAINLENGTH(l)\ eps\ m$
                                        %(Check that messages were recieved correctly)%
  $\forall\ t : time$
  $\bullet\ TransmissionOk(t)$
    $\Leftrightarrow \forall\ train : TrainID$
      $\bullet\ MessagesOK(MessagesFromTrain(train,\ t))$
                                        %(Check transmission from all trains successful)%
  $\bullet\ \neg\ TransmissionOk(t) \Rightarrow FailureMode(t)$
  $\bullet\ FailureMode(t)$
    $\Rightarrow \forall\ train : TrainID$
      $\bullet\ MessagesToTrain(train,\ t + 1) = \{\} + STOP$
  $\bullet\ SafeMode(t) \Leftrightarrow \neg\ FailureMode(t)$
                                        %(Check transmission from all trains successful)%
  $\forall\ train : TrainID;\ t : time$
  $\bullet\ \forall\ m : Set[T\_Message];\ s : speed$
    $\bullet\ messagesToSpeed(m) = s$
      $\Leftrightarrow TRAINSPEED(s)\ eps\ m \wedge TransmissionOk(t)$
                                        %(Get speed data from train message TRAINSPEED)%
  $\bullet\ seenSpeed(train,\ t)$
    $= messagesToSpeed(MessagesFromTrain(train,\ t))$
                                        %(Speed of train as percieved by Control System)%
  $\bullet\ \forall\ m : Set[T\_Message];\ p : position$
    $\bullet\ messagesToPosition(m) = p$
      $\Leftrightarrow TRAINPOSITION(p)\ eps\ m \wedge TransmissionOk(t)$
                                        %(get position data from trainmessage TRAINPOSITION)%
  $\bullet\ seenPosition(train,\ t)$
    $= messagesToPosition(MessagesFromTrain(train,\ t))$
                                        %(Position of train as percieved by Control System)%
  $\bullet\ \forall\ m : Set[T\_Message];\ l : length$
    $\bullet\ messagesToLength(m) = l$
      $\Leftrightarrow TRAINLENGTH(l)\ eps\ m \wedge TransmissionOk(t)$
                                        %(Get Length Data from train message TRAINLENGTH)%
  $\bullet\ seenLength(train,\ t)$
    $= messagesToLength(MessagesFromTrain(train,\ t))$
                                        %(Length of train as percieved by Control System)%
  $\bullet\ BlockFront(train,\ t)$
```

$= \mathit{seenPosition}(\mathit{train}, t)$
$+ \mathit{BrakingFunction}(\mathit{seenSpeed}(\mathit{train}, t))$
%(Block Front Calculation)%

- $\mathit{BlockFrontAcceled}(\mathit{train}, t)$
$= \mathit{seenPosition}(\mathit{train}, t)$
$+ \mathit{BrakingFunction}(\mathit{seenSpeed}(\mathit{train}, t) + 1)$
%(Block Front of train as if accelerated)%
- $\mathit{BlockFrontDeceled}(\mathit{train}, t)$
$= \mathit{seenPosition}(\mathit{train}, t)$
$+ \mathit{BrakingFunction}(\mathit{seenSpeed}(\mathit{train}, t) - 1)$
%(Block Front of train as if decelerated)%
- $\mathit{BlockRear}(\mathit{train}, t)$
$= \mathit{seenPosition}(\mathit{train}, t) - \mathit{seenLength}(\mathit{train}, t)$
%(Block Ream Calculation)%

end

spec CONTROLSYSTEMUSECASE =
CONTROLSYSTEM
then ops $\mathit{train1} : \mathit{TrainID};$
$t : \mathit{time}$
- $\mathit{MessagesFromTrain}(\mathit{train1}, 1)$
$= ((\{\} + \mathit{TRAINSPEED}(5)) + \mathit{TRAINPOSITION}(2))$
$+ \mathit{TRAINLENGTH}(1)$

then %implies
- $\mathit{seenSpeed}(\mathit{train1}, t) = 5$
%(seenSpeed is equal to the speed of the train at time t)%
- $\mathit{seenPosition}(\mathit{train1}, t) = 2$
%(seenPosition is equal to the position of the train a time t)%
- $\mathit{seenLength}(\mathit{train1}, t) = 1$
%(seenLength is equal to the length of the train at time t)%
- $\mathit{BlockRear}(\mathit{train1}, t) = 1$
%(BlockRear is equal to train position − train length at time t)%
- $\mathit{BlockFront}(\mathit{train1}, t) = 11$
%(BlockFront is equal to position of train plus the breaking function of the current speed)%
- $\mathit{BlockFrontAcceled}(\mathit{train1}, t) = 14$
%(BlockFrontAcceled is equal to BlockFront as if speed inc by 1)%
- $\mathit{BlockFrontDeceled}(\mathit{train1}, t) = 8$
%(BlockFrontDeceled is equal to BlockFront as if speed dec by 1)%

end

spec TRAINQUEUE =
TRAIN
then generated type
$\mathit{Queue} ::= \mathit{empty} \mid \mathit{add}(\mathit{Queue}; \mathit{TrainID})$

ops $add : Queue \times TrainID \to Queue;$
$remove : Queue \to Queue;$
$front : Queue \to ?\ TrainID;$
$back : Queue \to ?\ TrainID;$
$queueLength : Queue \to Value;$
$getTrain : Queue \times Value \to ?\ TrainID$
pred $\_\_elem\_\_ : TrainID \times Queue$
$\forall\ q : Queue;\ t, t1, t2 : TrainID;\ v : Value$
• $remove(empty) = empty$ %(Remove op on the empty queue)%
• $remove(add(empty, t)) = empty$
%(Remove op on the queue with single element)%
• $remove(add(add(q, t1), t2)) = add(remove(add(q, t1)), t2)$
%(Remove op on arbitrary length queue)%
• $\neg\ def\ front(empty)$ %(Not def front of empty queue)%
• $front(add(empty, t)) = t$
%(Front of queue with single element)%
• $front(add(add(q, t1), t2)) = front(add(q, t1))$
%(Front of arbitrary length queue)%
• $\neg\ def\ back(empty)$ %(not def back of empty queue)%
• $back(add(q, t)) = t$ %(back of arbitrary length queue)%
• $\neg\ t\ elem\ empty$ %(Elem check on empty queue)%
• $\neg\ t1 = t2 \Rightarrow \neg\ t1\ elem\ add(empty, t2)$
%(t1 elem queue is false if t1 has not been added)%
• $t\ elem\ add(q, t)$ %(t elem queue is true if t has been added)%
• $\neg\ t1 = t2 \Rightarrow (t1\ elem\ add(q, t2) \Leftrightarrow t1\ elem\ q)$
%(elem check on arbitrary length queue)%
• $queueLength(empty) = 0$ %(Length of the empty queue)%
• $queueLength(add(q, t1)) = suc(queueLength(q))$
%(Length of arbirary length qeueue)%
• $\neg\ def\ getTrain(empty, v)$
%(Not def getTrain on empty queue)%
• $getTrain(add(q, t1), 0) = t1$
%(get most recently added train)%
• $getTrain(add(q, t1), suc(v)) = getTrain(q, v)$
%(get any added train)%

end

spec TRAINQUEUEUSECASE =
TRAINQUEUE
then ops $Train1, Train2, Train3 : TrainID;$
$queue : Queue$
• $\neg\ Train1 = Train2$ %(Train1 does not equal Train2)%
• $\neg\ Train1 = Train3$ %(Train1 does not equal Train3)%
• $\neg\ Train2 = Train3$ %(Train2 does not equal Train3)%

- $\neg$ *queue* = *empty* %(queue does not equal empty)%

then %implies

- *def Train1* %(Train1 is always defined)%
- *def Train2* %(Train2 is always defined)%
- $\neg$ *front*(*empty*) = *Train1*
  %(ensure front of the empty list returns undefined)%
- $\neg$ *back*(*empty*) = *Train1*
  %(ensure back of the empty list returns undefined)%
- *front*(*add*(*add*(*add*(*empty*, *Train1*), *Train2*), *Train3*)) = *Train1*
  %(front on a queue with three elements returns the front element)%
- *front*(*add*(*add*(*empty*, *Train1*), *Train2*)) = *Train1*
  %(front returns the first element added to the list)%
- *back*(*add*(*empty*, *Train1*)) = *Train1*
  %(back on a queue with a single element returns that element)%
- *back*(*add*(*add*(*queue*, *Train1*), *Train2*)) = *Train2*
  %(back always returns the last element added to the queue)%
- *remove*(*add*(*add*(*empty*, *Train1*), *Train2*))
  = *add*(*empty*, *Train2*)
  %(remove returns the queue with the least recently added item removed)%
- *remove*(*add*(*add*(*add*(*empty*, *Train1*), *Train2*), *Train3*))
  = *add*(*add*(*empty*, *Train2*), *Train3*)
  %(remove returns the queue with the least recently added item removed with >2 elements)%
- $\neg$ *Train1 elem empty*
  %(Train1 is not an element of the empty queue)%
- $\neg$ *Train1 elem add*(*empty*, *Train2*)
  %(Train1 is not an element of a queue containing only Train2)%
- *Train1 elem add*(*queue*, *Train1*)
  %(Train1 is an element of a queue when it is the most recent added element)%
- *Train2 elem add*(*add*(*add*(*queue*, *Train1*), *Train2*), *Train3*)
  %(Train2 is an element of a queue when it is the middle added element to the queue)%
- *queueLength*(*add*(*empty*, *Train1*)) = 1
  %(queueLength on a trainqueue with 1 elements returns 1)%
- *queueLength*(*add*(*add*(*add*(*empty*, *Train1*), *Train2*), *Train3*))
  = 3
  %(queueLength on a trainQueue with 3 elements returns 3)%
- $\neg$ *getTrain*(*empty*, 0) = *Train1*
  %(getTrain on the empty list returns undefined)%
- *getTrain*(*add*(*empty*, *Train1*), 0) = *Train1*
  %(element 0 of the queue with one element returns that element)%
- *getTrain*(*add*(*queue*, *Train1*), 0) = *Train1*
  %(element 0 of any queue returns the most recent element)%
- *getTrain*(*add*(*add*(*add*(*queue*, *Train1*), *Train2*), *Train3*), 1)
  = *Train2*
  %(getTrain can retrieve the second latest added train from the queue)%

```
end

spec ControlSystemChain =
     ControlSystem
and  TrainQueue
then sort  ControlSystem
     ops   NextControlSystem : ControlSystem →? ControlSystem;
           PreviousControlSystem
           : ControlSystem →? ControlSystem;
           TrainQueue : ControlSystem × time → Queue;
           InBoundary : ControlSystem → position;
           OutBoundary : ControlSystem → position
     preds Accel : TrainID × TrainID × time;
           Proceed : TrainID × TrainID × time;
           Break : TrainID × TrainID × time
     ∀ cs1 : ControlSystem; train, train1 : TrainID; t : time
     • InBoundary(cs1) < Position(train, t)
       ∧ train elem TrainQueue(PreviousControlSystem(cs1), t)
       ∧ ¬ train elem TrainQueue(cs1, t)
       ⇒ TrainQueue(cs1, t + 1) = add(TrainQueue(cs1, t), train)
          ∧ TrainQueue(PreviousControlSystem(cs1), t + 1)
            = remove(TrainQueue(PreviousControlSystem(cs1), t))
                                    %(Pass train to next control system when passing boundary)%
     • Proceed(train, train1, t)
       ⇔ BlockFront(train, t) < BlockRear(train1, t)
                                                                   %(Proceed if safe too )%
     • Accel(train, train1, t)
       ⇔ Proceed(train, train1, t)
          ∧ BlockFrontAcceled(train, t) < BlockRear(train1, t)
                                          %(Accelerate if can proceed and safe to accelerate)%
     • Break(train, train1, t)
       ⇔ ¬ Proceed(train, train1, t)
          ∧ BlockRear(train1, t) < BlockFrontDeceled(train, t)
                                                     %(Break if decelerating not sufficient)%
     • BlockFront(front(TrainQueue(cs1, t)), t)
       < InBoundary(NextControlSystem(cs1))
       ⇒ MessagesToTrain(front(TrainQueue(cs1, t)), t + 1)
          = {} + ACCELERATE
             %(if block of front train does not overlap with next control system then accelerate)%
     • InBoundary(NextControlSystem(cs1))
       < BlockFront(front(TrainQueue(cs1, t)), t)
       ∧ ¬ Proceed(front(TrainQueue(cs1, t)),
                   back(TrainQueue(NextControlSystem(cs1),
                                   t)),
                t)
   ∧ ¬ Break(front(TrainQueue(cs1, t)),
               back(TrainQueue(NextControlSystem(cs1), t)),
               t)
   ⇒ MessagesToTrain(front(TrainQueue(cs1, t)), t + 1)
      = {} + DECELERATE
                  %(DECELERATE MessageToTrain calc for front train with CS overlap)%
• InBoundary(NextControlSystem(cs1))
   < BlockFront(front(TrainQueue(cs1, t)), t)
   ∧ ¬ Proceed(front(TrainQueue(cs1, t)),
                back(TrainQueue(NextControlSystem(cs1),
                                 t)),
                t)
   ∧ Break(front(TrainQueue(cs1, t)),
            back(TrainQueue(NextControlSystem(cs1), t)), t)
   ⇒ MessagesToTrain(front(TrainQueue(cs1, t)), t + 1)
      = {} + BREAK
                      %(BREAK MessageToTrain calc for front train with CS overlap)%
• InBoundary(NextControlSystem(cs1))
   < BlockFront(front(TrainQueue(cs1, t)), t)
   ∧ Proceed(front(TrainQueue(cs1, t)),
              back(TrainQueue(NextControlSystem(cs1), t)),
              t)
   ∧ ¬ Accel(front(TrainQueue(cs1, t)),
              back(TrainQueue(NextControlSystem(cs1), t)),
              t)
   ⇒ MessagesToTrain(front(TrainQueue(cs1, t)), t + 1)
      = {} + CONTINUE
                   %(CONTINUE MessageToTrain calc for front train with CS overlap)%
• InBoundary(NextControlSystem(cs1))
   < BlockFront(front(TrainQueue(cs1, t)), t)
   ∧ Proceed(front(TrainQueue(cs1, t)),
              back(TrainQueue(NextControlSystem(cs1), t)),
              t)
   ∧ Accel(front(TrainQueue(cs1, t)),
            back(TrainQueue(NextControlSystem(cs1), t)), t)
   ⇒ MessagesToTrain(front(TrainQueue(cs1, t)), t + 1)
      = {} + ACCELERATE
                  %(ACCELERATE MessageToTrain calc for front train with CS overlap)%
• ∀ n : Value
   • ¬ Proceed(getTrain(TrainQueue(cs1, t), n),
                getTrain(TrainQueue(cs1, t), n + 1), t)
     ∧ ¬ Break(getTrain(TrainQueue(cs1, t), n),
                getTrain(TrainQueue(cs1, t), n + 1), t)
```

```
        ⇒ $MessagesToTrain(getTrain(TrainQueue(cs1,\ t),\ n),$
                              $t + 1)$
          $= \{\} + DECELERATE$
                                  %(DECELERATE MessageToTrain calc for all other trains)%
    • $\forall\ n : Value$
      • $\neg\ Proceed(getTrain(TrainQueue(cs1,\ t),\ n),$
                    $getTrain(TrainQueue(cs1,\ t),\ n + 1),\ t)$
        $\wedge\ Break(getTrain(TrainQueue(cs1,\ t),\ n),$
                   $getTrain(TrainQueue(cs1,\ t),\ n + 1),\ t)$
        ⇒ $MessagesToTrain(getTrain(TrainQueue(cs1,\ t),\ n),$
                              $t + 1)$
          $= \{\} + BREAK$
                                       %(BREAK MessageToTrain calc for all other trains)%
    • $\forall\ n : Value$
      • $Proceed(getTrain(TrainQueue(cs1,\ t),\ n),$
                 $getTrain(TrainQueue(cs1,\ t),\ n + 1),\ t)$
        $\wedge\ Accel(getTrain(TrainQueue(cs1,\ t),\ n),$
                 $getTrain(TrainQueue(cs1,\ t),\ n + 1),\ t)$
        ⇒ $MessagesToTrain(getTrain(TrainQueue(cs1,\ t),\ n),$
                              $t + 1)$
          $= \{\} + ACCELERATE$
                                  %(ACCELERATE MessageToTrain calc for all other trains)%
    • $\forall\ n : Value$
      • $Proceed(getTrain(TrainQueue(cs1,\ t),\ n),$
                 $getTrain(TrainQueue(cs1,\ t),\ n + 1),\ t)$
        $\wedge\ \neg\ Accel(getTrain(TrainQueue(cs1,\ t),\ n),$
                    $getTrain(TrainQueue(cs1,\ t),\ n + 1),\ t)$
        ⇒ $MessagesToTrain(getTrain(TrainQueue(cs1,\ t),\ n),$
                              $t + 1)$
          $= \{\} + CONTINUE$
                                    %(CONTINUE MessageToTrain calc for all other trains)%
end
```

C TrainQueue Proof Details

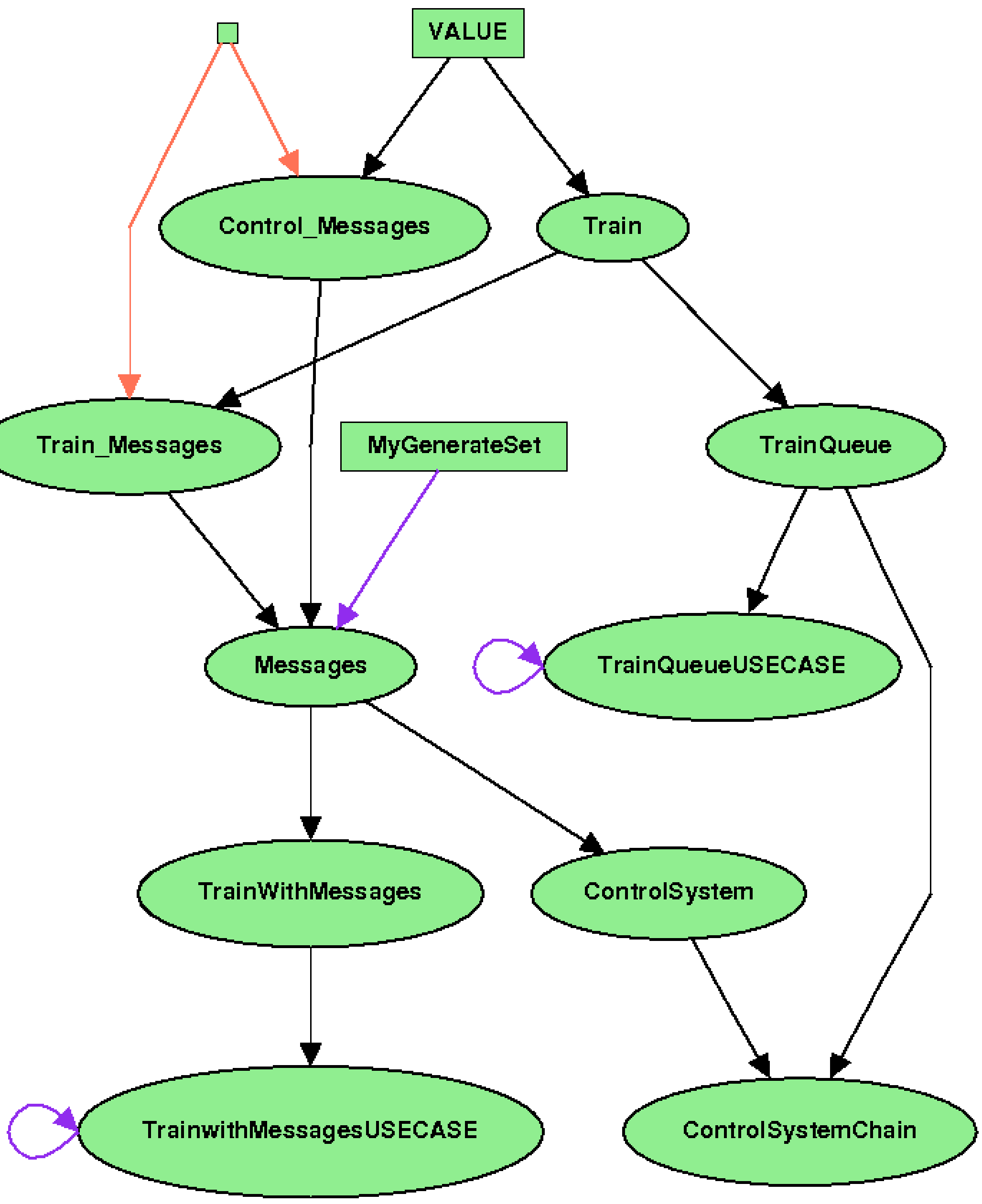
```
---------------------------------------------------------------------------
Train1 is always defined
    Com: id_CASL.SulPeCFOL=;CASL2SubCFOL;CASL2SoftFOL : CASL -> SoftFOL
        Status: Proved
        Used axioms: "ga_totality_16"
        Used time: 00:00:00.309999999999
        Prover: SPASS
        Tactic script
        Proof tree

---------------------------------------------------------------------------
Train2 is always defined
    Com: id_CASL.SulPeCFOL=;CASL2SubCFOL;CASL2SoftFOL : CASL -> SoftFOL
        Status: Proved
        Used axioms: "ga_totality_17"
        Used time: 00:00:00.239999999999
        Prover: SPASS
        Tactic script
        Proof tree

---------------------------------------------------------------------------
ensure front of the empty list returns undefined
    Com: id_CASL.SulPeCFOL=;CASL2SubCFOL;CASL2SoftFOL : CASL -> SoftFOL
        Status: Proved
        Used axioms: "Train1 is always defined",
                     "Not def front of empty queue"
        Used time: 00:00:00.28
        Prover: SPASS
        Tactic script
        Proof tree

---------------------------------------------------------------------------
ensure back of the empty list returns undefined
    Com: id_CASL.SulPeCFOL=;CASL2SubCFOL;CASL2SoftFOL : CASL -> SoftFOL
        Status: Proved
        Used axioms: "Train1 is always defined",
                     "not def back of empty queue"
        Used time: 00:00:00.22
        Prover: SPASS
        Tactic script
        Proof tree
```

```
---------------------------------------------------------------------------
front on a queue with three elements returns the front element
    Com: id_CASL.SulPeCFOL=;CASL2SubCFOL;CASL2SoftFOL : CASL -> SoftFOL
        Status: Proved
        Used axioms: "ga_totality_24", "ga_totality_18",
                     "Train1 is always defined", "Train2 is always defined",
                     "Front of queue with single element", "ga_totality_23",
                     "Front of arbitrary length queue"
        Used time: 00:00:01.79
        Prover: SPASS
        Tactic script
        Proof tree

---------------------------------------------------------------------------
front returns the first element added to the list
    Com: id_CASL.SulPeCFOL=;CASL2SubCFOL;CASL2SoftFOL : CASL -> SoftFOL
        Status: Proved
        Used axioms: "ga_totality_24", "Train1 is always defined",
                     "Train2 is always defined", "Front of queue with single element",
                     "Front of arbitrary length queue"
        Used time: 00:00:01.949999999999
        Prover: SPASS
        Tactic script
        Proof tree

---------------------------------------------------------------------------
back on a queue with a single element returns that element
    Com: id_CASL.SulPeCFOL=;CASL2SubCFOL;CASL2SoftFOL : CASL -> SoftFOL
        Status: Proved
        Used axioms: "ga_totality_24", "Train1 is always defined",
                     "back of arbitrary length queue"
        Used time: 00:00:00.27
        Prover: SPASS
        Tactic script
        Proof tree

---------------------------------------------------------------------------
back always returns the last element added to the queue
    Com: id_CASL.SulPeCFOL=;CASL2SubCFOL;CASL2SoftFOL : CASL -> SoftFOL
        Status: Proved
        Used axioms: "ga_totality_25", "Train1 is always defined",
                     "Train2 is always defined", "ga_totality_23",
                     "back of arbitrary length queue"
        Used time: 00:00:00.88
        Prover: SPASS
        Tactic script
        Proof tree

---------------------------------------------------------------------------
remove returns the queue with the least recently added item removed
    Com: id_CASL.SulPeCFOL=;CASL2SubCFOL;CASL2SoftFOL : CASL -> SoftFOL
        Status: Proved
        Used axioms: "ga_totality_24", "Train1 is always defined",
                     "Train2 is always defined",
                     "Remove op on the queue with single element",
                     "Remove op on arbitrary length queue"
        Used time: 00:00:57.929999999999
        Prover: SPASS
        Tactic script
        Proof tree

---------------------------------------------------------------------------
remove returns the queue with the least recently added item removed with >2 elements
    Com: id_CASL.SulPeCFOL=;CASL2SubCFOL;CASL2SoftFOL : CASL -> SoftFOL
        Status: Proved
        Used axioms: "ga_totality_24", "ga_totality_18",
                     "Train1 is always defined", "Train2 is always defined",
                     "remove returns the queue with the least recently added item removed",
                     "ga_totality_23", "t elem queue is true if t has been added",
                     "Remove op on arbitrary length queue"
        Used time: 00:00:58.909999999999
        Prover: SPASS
        Tactic script
        Proof tree

---------------------------------------------------------------------------
Train1 is not an element of the empty queue
    Com: id_CASL.SulPeCFOL=;CASL2SubCFOL;CASL2SoftFOL : CASL -> SoftFOL
        Status: Proved
        Used axioms: "Elem check on empty queue",
                     "ga_predicate_strictness_1"
        Used time: 00:00:00.27
        Prover: SPASS
        Tactic script
        Proof tree

---------------------------------------------------------------------------
```

```
Train1 is not an element of a queue containing only Train2
    Com: id_CASL.SulPeCFOL=;CASL2SubCFOL;CASL2SoftFOL : CASL -> SoftFOL
        Status: Proved
        Used axioms: "Train2 is always defined",
                     "Train1 does not equal Train2", "ga_predicate_strictness_1",
                     "t1 elem queue is false if t1 has not been added"
        Used time: 00:00:00.27
        Prover: SPASS
        Tactic script
        Proof tree

---------------------------------------------------------------------------
Train1 is an element of a queue when it is the most recent added element
    Com: id_CASL.SulPeCFOL=;CASL2SubCFOL;CASL2SoftFOL : CASL -> SoftFOL
        Status: Proved
        Used axioms: "ga_totality_25", "Train1 is always defined",
                     "t elem queue is true if t has been added"
        Used time: 00:00:00.289999999999
        Prover: SPASS
        Tactic script
        Proof tree

---------------------------------------------------------------------------
Train2 is an element of a queue when it is the middle added element to the queue
    Com: id_CASL.SulPeCFOL=;CASL2SubCFOL;CASL2SoftFOL : CASL -> SoftFOL
        Status: Proved
        Used axioms: "ga_totality_25", "ga_totality_18",
                     "Train1 is always defined", "Train2 is always defined",
                     "Train2 does not equal Train3",
                     "Train1 is an element of a queue when it is the most recent added eleme
                     "ga_predicate_strictness_1",
                     "t elem queue is true if t has been added",
                     "elem check on arbitrary length queue"
        Used time: 00:00:01.04
        Prover: SPASS
        Tactic script
        Proof tree

---------------------------------------------------------------------------
queueLength on a trainqueue with 1 elements returns 1
    Com: id_CASL.SulPeCFOL=;CASL2SubCFOL;CASL2SoftFOL : CASL -> SoftFOL
        Status: Proved
        Used axioms: "ga_totality_24", "Train1 is always defined",
                     "Length of the empty queue", "Ax10",
                 "Length of arbirary length qeueue"
        Used time: 00:00:01.659999999999
        Prover: SPASS
        Tactic script
        Proof tree

---------------------------------------------------------------------------
queueLength on a trainQueue with 3 elements returns 3
    Com: id_CASL.SulPeCFOL=;CASL2SubCFOL;CASL2SoftFOL : CASL -> SoftFOL
        Status: Proved
        Used axioms: "ga_totality_24", "ga_totality_18",
                     "Train1 is always defined", "Train2 is always defined", "Ax12",
                     "Ax11",
                     "back on a queue with a single element returns that element",
                     "queueLength on a trainqueue with 1 elements returns 1",
                     "ga_strictness", "ga_totality_23",
                     "Length of arbirary length qeueue"
        Used time: 00:00:01.989999999999
        Prover: SPASS
        Tactic script
        Proof tree

---------------------------------------------------------------------------
getTrain on the empty list returns undefined
    Com: id_CASL.SulPeCFOL=;CASL2SubCFOL;CASL2SoftFOL : CASL -> SoftFOL
        Status: Proved
        Used axioms: "ga_totality", "Train1 is always defined",
                     "Not def getTrain on empty queue"
        Used time: 00:00:01.459999999999
        Prover: SPASS
        Tactic script
        Proof tree

---------------------------------------------------------------------------
element 0 of the queue with one element returns that element
    Com: id_CASL.SulPeCFOL=;CASL2SubCFOL;CASL2SoftFOL : CASL -> SoftFOL
        Status: Proved
        Used axioms: "ga_totality_24", "Train1 is always defined",
                     "get most recently added train"
        Used time: 00:00:00.27
        Prover: SPASS
        Tactic script
        Proof tree
```

```
---------------------------------------------------------------------------
element 0 of any queue returns the most recent element
    Com: id_CASL.SulPeCFOL=;CASL2SubCFOL;CASL2SoftFOL : CASL -> SoftFOL
        Status: Proved
        Used axioms: "ga_totality_25", "Train1 is always defined",
                     "get most recently added train"
        Used time: 00:00:00.27
        Prover: SPASS
        Tactic script
        Proof tree

---------------------------------------------------------------------------
getTrain can retrieve the second latest added train from the queue
    Com: id_CASL.SulPeCFOL=;CASL2SubCFOL;CASL2SoftFOL : CASL -> SoftFOL
        Status: Proved
        Used axioms: "ga_totality_25", "ga_totality_18", "ga_totality",
                     "Train1 is always defined", "Train2 is always defined",
                     "Train2 does not equal Train3", "Ax10",
                     "Train1 is an element of a queue when it is the most recent added eleme
                     "Train2 is an element of a queue when it is the middle added element to
                     "ga_predicate_strictness_1", "ga_totality_23",
                     "get most recently added train", "get any added train",
                     "elem check on arbitrary length queue"
        Used time: 00:00:16.6
        Prover: SPASS
        Tactic script
        Proof tree
```

D Graphic Representation of Specification



E Bibliography

References

[Abr06] Jean-Raymond Abrial. Formal methods in industry: achievements, problems, future. *International Conference on Software Engineering archive. Proceedings of the 28th international conference on Software engineering*, 2006.

[Bar92] Roger Barnett. British rail's intercity 125 and 225. *California High Speed Rail Series*, (UCTC No. 114), June 1992.

[BBFM82] Berg, Boebert, Franta, and Moher. *Formal Methods of Program Verification and Specification.* Prentice-Hall Inc., 1982.

[Bjø06] Dines Bjørner. *Software Engineering 1: Abstraction and Modelling.* Springer, June 2006.

[BM04a] Michel Bidoit and Peter Mosses. *CASL Reference Manual: The Complete Documentation of the Common Algebraic Specification Language.* Springer, 2004.

[BM04b] Michel Bidoit and Peter Mosses. *CASL user manual : introduction to using the Common algebraic specification language CASL.* Springer, 2004.

[CoF10] CoFI. Cofi: Casl, the common algibraic specification language. `http://www.brics.dk/Projects/CoFI/`, 7 February 2010.

[Coh87] Avra Cohn. A proof of correctness of the viper microprocessor: the first level. In *Proceedings of the Calgary Hardware Verification Workshop*, Calgary Canada, January 1987.

[Com09] Railway Heritage Committee. Railway heritage committee: Britain's railway history. `http://www.railwayheritage.org.uk/history.asp`, 13 October 2009.

[DT96] M. Dorfman and R.H. Thayer. *Software Engineering.* Wiley-IEEE Computer Society Press, 1996.

[Gov89] UK Governement. Regulation of railways act 1889. `http://www.statutelaw.gov.uk/content.aspx?ActiveTextDocId=1061371`, 20 August 1889.

[Har99] Ralph Harrington. The railway accident: trains, trauma and technological crisis in nineteenth-century britain. `http://www.railwayheritage.org.uk/history.asp`, 1999.

[Jac97] Johnathan Jacky. *The way of Z: Practical Programming with Formal Methods.* Cambridge University Press, 1997.

[Lee10] Julia Lee. The london illustrated news - accidents and disasters. http://www.flickr.com/photos/museemccordmuseum/2919039636, 21 April 2010.

[Mac07] Miles Macnair. *William James (1771-1837): the man who discovered George Stephenson.* Oxford: Railway and Canal Historical Society, 2007.

[MML10] Till Mossakowski, Christian Maeder, and Klaus Luttich. Hets user guide version 0.85. http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/hets/UserGuide.pdf, 3 February 2010.

[Mos10] Dr. Till Mossakowski. Hets homepage. http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/hets/index_e.htm, 3 February 2010.

[Pag09] Railway Technical Web Pages. The development and principles of uk signalling. http://www.railway-technical.com/sigtxt1.shtml, 30 September 2009.

[Pro09] Powys Digital History Project. A train crash at tylwch in 1899. http://history.powys.org.uk/school1/llanidloes/crash3.shtml, 13 October 2009.

[Ray09] Wilson H. Raynar. *The Safety of British Railways; Or, Railway Accidents: How Caused and How Prevented.* BiblioBazaar, LLC, 2009.

[RN69] Brian Randell and Peter Naur. Report on a conference sponsored by the nato science committee garmisch, germany, 7th to 11th october 1968. http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF, January 1969.

[Rom94] Jon Roma. Interlocking machine locking bed.jpeg. http://en.eikipedia.org/wiki/File:Interlocking_machine_locking_bed.jpg, 1994.

[RT09] Railway-Technology. Docklands light railway capacity upgrading, london london, united kingdom. http://www.flickr.com/photos/museemccordmuseum/2919039636, 15 October 2009.

[Sch89] P.N Scharbach. *Formal Methods: Theory and Practice.* BSP Professional Books, 1989.

[Som06] Ian Sommerville. *Software Engineering: (Update) (8th Edition) (International Computer Science Series).* Addison Wesley, June 2006.

[tea3] The SPASS team. Max-plank-institut-f́ur informatik - auntomation of logic: Spass. http://www.spass-prover.org/, May 3.

[UNI09a] UNISIG. Ertms levels: Different ertms/etcs application levels to match customers' needs. http://www.ertms.com/2007v2/factsheets/ERTMS%20Factsheet%20-%20ERTMS%20Levels.pdf, 15 October 2009.

[UNI09b] UNISIG. Unisig an industrial consortium to develop ertms/etcs technical specifications. http://www.ertms.com/2007v2/factsheets/ERTMS%20Factsheet%20-%20UNISIG.pdf, 15 October 2009.

[UNI09c] UNISIG. What is ertms. `http://www.ertms.com/2007v2/what.html`, 15 October 2009.

[Var09] Various. Wikipedia: History of rail transport. `http://en.wikipedia.org/wiki/History_of_rail_transport`, 13 October 2009.

[Var10] Various. Wikipedia: George stephenson. `http://en.wikipedia.org/wiki/George_Stephenson`, 21 April 2010.

[W.D72] Edsger W.Dijkstra. The humble programmer. `http://www.cs.virginia.edu/~evans/cs655/readings/ewd498.html`, February 1972.