Formal Verification of Ladder Logic

Karim Kanso

A thesis submitted to the University of Wales in candidature for the degree of Master of Research

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

December 3, 2008

Abstract

This project studied whether a digital interlocking which had been programmed with ladder logic (*Boolean program*) would obey generic safety properties. This was carried out by translating the ladder logic into an alternate representation and applying various techniques to allow specification of safety properties. Finally, a proof engine was used to formally verify if these properties were fulfilled and if they are not, then human readable documentation would be generated.

Declaration

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

Signed .... (candidate)

Date ....

Statement 1

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

Signed .... (candidate)

Date ....

Statement 2

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

Signed .... (candidate)

Date ....

Contents

Acknowledgements

I would like to thank Simon Chadwick, Nick Smith and Peter Duggen from Westinghouse Rail Systems Limited for their prompt replies to questions and their guidance.

Oliver Kullmann for guidance with the Boolean Satisfaction Problem, Anton Setzer and Faron Moller from Swansea University for their guidance and supervision throughout the project.

Markus Roggenbach from Swansea University and Achim Jung from The University of Birmingham for their comments and help improving this thesis.

List of Tables

List of Figures

Chapter 1

Overview

1.1 Invensys

In 1869 George Westinghouse patented the air brake for trains and started the Westinghouse Air Brake Company, latter became Invensys. This device allows trains to brake with fail safe precision, and works by using air pressure to keep the brake pads off the wheels, so as soon as the pressure is released the train stops. This happens when power is cut in the case of an emergency or guaranteeing that a train stops at a platform, "Arms" are placed on the track such that when a train goes over, they release the air pressure making a train brake with fail safe precision.

Continuing along the line of railway safety Westinghouse has become one of the largest suppliers of railway control equipment in the world. Some of the countries that rely on Westinghouse include Australia, Hong Kong, Germany, Spain and the UK.

Westinghouse Train Radio and Advanced Control Equipment (*Westrace*) which is the interlocking the project is concerned with; these are used in over 12 countries. Westrace's are programmed with Ladder logic which is a graphical representation of Boolean valued assignments. The Westrace was first designed over 10 years ago and is continuously upgraded as new technologies are discovered. This upgrade process is greatly simplified due to the modularised design of the Westrace.

Invensys contacted Swansea University in the summer of 2007 to initiate a project for formal verification of Westrace's programmed with ladder logic.

1.1.1 Objectives

Westinghouse has many objectives, one of which is "*to ensure foreseeable technical risk is systematically engineered out of our products and systems and that the risks associated with the construction, maintenance and use of our products and systems are identified, assessed and combated at their source*". This project falls under the above mentioned objective.

After a discussion with Westinghouse, it was decided that the project will deliver a functioning prototype capable of formally verifying that a specific interlocking ratifies with arbitrary signalling principles.

The project was split into two problems, the first is concerned with the actual verification of an interlocking with respect to a safety condition and the second is concerned with the verification of signalling principles.

1.2 Problem ① – Verification

Problem ① can be described as "verifying whether a safety condition holds in a specific Westrace interlocking". This task is composed of four sub-tasks,

1. Building a propositional model of the ladder logic program to be verified, and

2. Generating propositional proof formulæ that can verify whether the ladder logic fulfils safety conditions, and

3. Entering the proof formulæ into a proof engine (*SAT-Solver*), and

4. In the case a counter example is identified, documentation is generated.

Tasks 1-3 are the main tasks of interest, task 4 is concerned with processing the output of task 3 (proving).

Task 1 is discussed in detail in Sect. 5, Ladder logic canonically translates to a Boolean program consisting of a list of assignments, where variables are assigned evaluated propositional formulæ. The ladder logic is what represents the assembly language level for the interlocking as it is at the lowest level. Higher level languages are translated into ladder logic.

Execution of the ladder logic commences by first initialising variables to initial values, then starting a repetitive loop. See *Fig. 1.1* for the execution strategy.

```
Initialise Variables
while(true) {
   Output
   Input
   $x_1$ := $\varphi_1$
   ⋮
   $x_n$ := $\varphi_n$
}
```

Figure 1.1: Execution Strategy of Ladder, where $\mathtt{x}_i$ are variables and φ_j are propositional formula

The loop repetitively outputs values of variables to the hardware, inputs values from the controls and hardware to variables and evaluates the ladder which updates variables.

An example assignment $d := (a \land \neg b) \lor c$ would be represented in ladder logic as

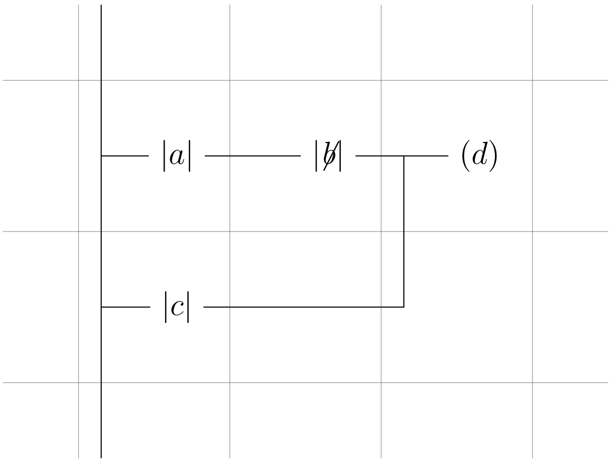


Figure 1.2: Example Rung in Ladder Logic

a, b, c and d are latches, $\not{b}$ is a negation and the brackets around d indicate that it an output. The diagram's semantics are very similar to that of a circuit diagram, as ladder logic was originally used to program microchips.

The second task of proof formula generation constructs propositional formula used for the proving stage. See Sect. 6 for a detailed discussion of the proof formula. In short, the aim is to show that a safety condition ψ holds always after each execution of the ladder. There are different techniques that can be used, for this project the principle of induction is applied with promising results.

The principle of induction requires that two formulæ hold, the base case and the inductive step. The base case shows that from the initial state φ_I and after one execution of the ladder $\varphi_{\mathcal{L}}$ the safety condition ψ holds.

$$\varphi_I \wedge \varphi_{\mathcal{L}} \rightarrow \psi' \tag{1.1}$$

The inductive step, requires that from an arbitrary state where the safety condition holds and after one execution of the ladder the safety condition holds.

$$\psi \wedge \varphi_{\mathcal{L}} \rightarrow \psi' \tag{1.2}$$

where $\varphi_{\mathcal{L}}$ has the form $(x'_1 \leftrightarrow \varphi'_1) \wedge \cdots \wedge (x'_n \leftrightarrow \varphi'_n)$. x'_i is a new proposition representing the state of variable x_i after execution. φ'_i is the result of replacing $x_1,\ldots,x_{i-1}$ by $x'_1,\ldots,x'_{i-1}$ in φ_i, where φ_i is the same as φ_i in *Fig. 1.1.* ψ' is the result of replacing x_i by x'_i in ψ.

If its possible to falsify formulæ 1.1 or 1.2, then this indicates *possible* existence of a counter example. The counter example is only a real counter example if it is in a reachable state. Two techniques are used to characterise reachable states, the first technique is to identify valid combinations of inputs and the second is to mathematically prove invariance about the system using formulæ 1.1 and 1.2. i.e. a signal should not be red and green simultaneously. The first technique is used to restrict possible inputs. I.e. a switch that can be in one of three positions, is represented by three propositional variables A, B and C, one for each position. At most one of these variables should be true.

These invariances can be used to weaken the proof formulæ, allowing for many unreachable states to be discarded. Given an invariance φ_{Inv}, the proof formulæ become:

$$\varphi_I \wedge \varphi_{\mathcal{L}} \wedge \psi_{Inv} \rightarrow \psi' \tag{1.3}$$

and

$$\psi \wedge \varphi_{\mathcal{L}} \wedge \psi_{Inv} \rightarrow \psi' \tag{1.4}$$

Task 3, proving can be performed by a SAT-Solver, for the proof it is required to show that formulæ 1.3 and 1.4 hold always. Thus, the negation of these formulæ should never hold.

$$\neg(\varphi_I \wedge \varphi_{\mathcal{L}} \wedge \psi_{Inv} \rightarrow \psi') \tag{1.5}$$

and

$$\neg(\psi \wedge \varphi_{\mathcal{L}} \wedge \psi_{Inv} \rightarrow \psi') \tag{1.6}$$

The two formulæ 1.5 and 1.6 are entered into a SAT-Solver, in the case that the solver indicates unsatisfiability the safety condition ψ holds always. Otherwise, a counter example has been identified and documentation is produced (task 4).

1.2.1 Example

To help explain the principles of problem ① a scenario of a *Pelican*[1] crossing has been constructed, see *Fig. 1.5* for the ladder logic diagram that controls the "toy" crossing. See *Fig. 1.3* for an image of a Pelican crossing and *Fig. 1.4* for a diagram of the Pelican crossing used in the scenario.

Figure 1.3: Picture of a Pelican Crossing in Swansea, UK

[1] Pelican being an acronym for **PE**destrian **LI**ght **CON**trolled crossing.

Scenario for the example:

A simple "Pelican crossing" consisting of two traffic lights that can be either red or green, two pedestrian lights that can be either red or green, two push buttons and an audible signal for the blind. The traffic signals should never show a green aspect at the same time that pedestrians see a green aspect.

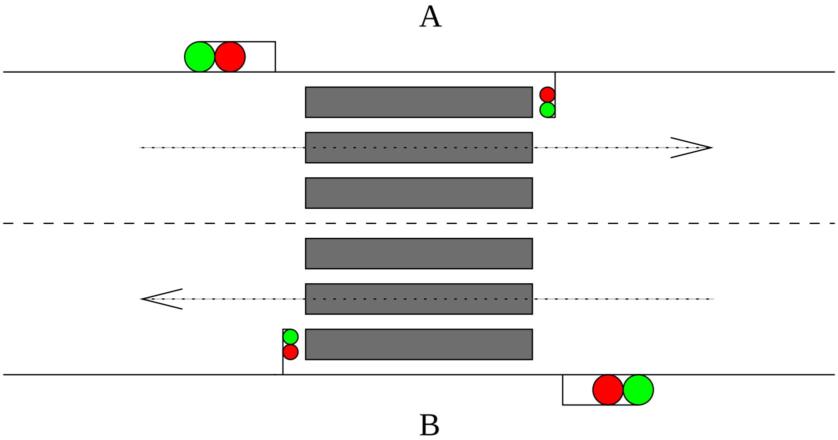


Figure 1.4: Diagram of a Pelican Crossing

There are 8 variables used in the ladder logic, consisting of 1 input, 5 outputs and 2 latches. See *Fig. 1.5* for the ladder.

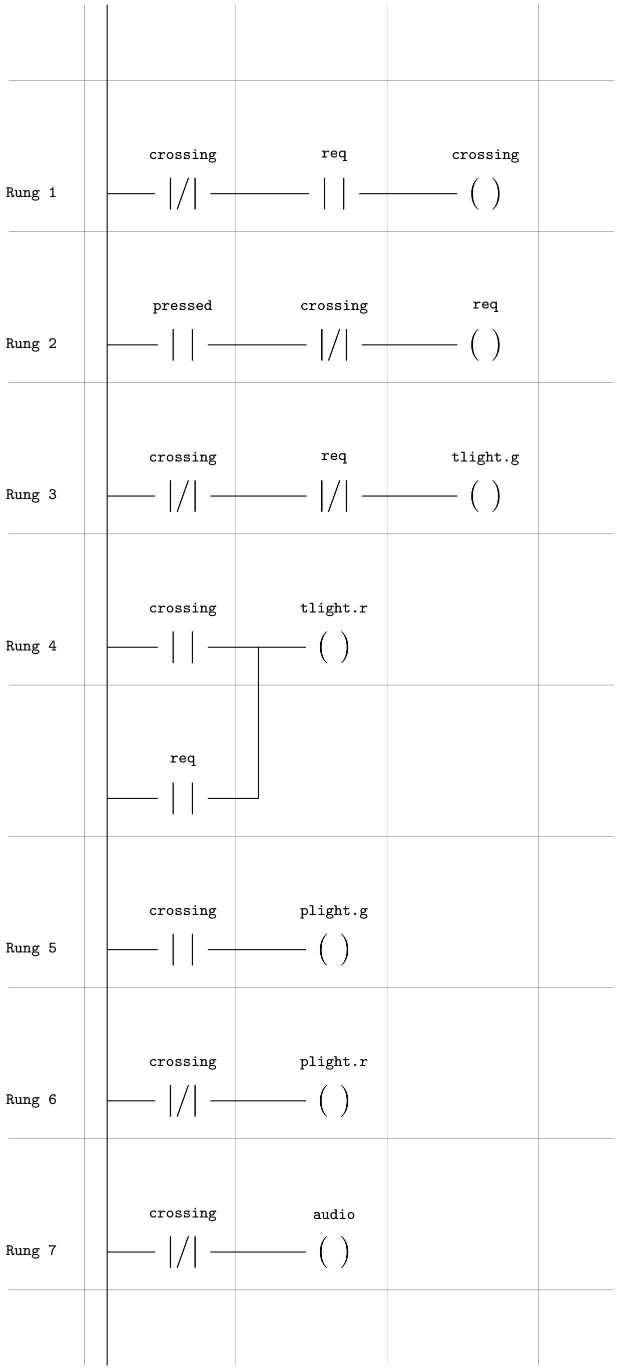


Figure 1.5: Example pelican ladder logic digram.

The system has one input, namely `pressed` that goes high when a pedestrian has their finger on the request crossing button. There are also 5 outputs that control the hardware, namely

```
plight.g  tlight.g  audio
plight.r  tlight.r
```

The `p`'s and `t's` represent pedestrians and traffic respectively. `audio` is used to drive an audible signal such that blind people can use the crossing.

Internally the system has two latches which control the state of the system, `crossing` and `req`. The inputs (and past inputs) determine the state of these latches which are then used to determine the outputs. The state of latch `crossing` indicates whether the system is letting pedestrians cross the road and the state of `req` to indicate that a pedestrian has requested to cross the road.

A simple analysis shows that the system has four possible states, namely

`crossing`	`req`	**State**
`false`	`false`	Neither crossing or pedestrians waiting.
`false`	`true`	Requested but not crossing.
`true`	`false`	Crossing but not requested.
`true`	`true`	Crossing and requested.

Table 1.1: Traffic Scenario States

Reachable States The state where `crossing` and `req` are both `true` is not a reachable state, if the system is letting people cross the road, then it is not possible (*according to the ladder logic*) to also request a crossing.

It should be noted that this example is simplified in the sense that time is not considered, between requesting to cross the road and receiving a green signal to cross would be almost instant, there is a delay on one cycle which can be measured in milliseconds.

Converting the 7 rungs in the ladder into a simple program consisting of Boolean valued assignments would be:

$$\begin{aligned}
crossing &:= \neg crossing \wedge req \\
req &:= pressed \wedge \neg crossing \\
tlight.g &:= \neg crossing \wedge \neg req \\
tlight.r &:= crossing \wedge req \\
plight.g &:= crossing \\
plight.r &:= \neg crossing \\
audio &:= crossing
\end{aligned}$$

Then, converting these assignments into propositional logic yields:

$$\begin{array}{lcl}
[& (crossing' \leftrightarrow \neg crossing \wedge req) & , \\
 & (req' \leftrightarrow pressed \wedge \neg crossing') & , \\
 & (tlight.g' \leftrightarrow \neg crossing' \wedge \neg req') & , \\
 & (tlight.r' \leftrightarrow crossing' \wedge req') & , \\
 & (plight.g' \leftrightarrow crossing') & , \\
 & (plight.r' \leftrightarrow \neg crossing') & , \\
 & (audio' \leftrightarrow crossing') &]
\end{array}$$

where the *primes* are assumed to be fresh.

Finally, let $\varphi_{\mathcal{L}}$ be a conjunction of these formulæ. $\varphi_{\mathcal{L}}$ is a complete model of the ladders execution in propositional logic.

1.2.2 Proof Formulæ

Given a safety condition to be proven for the system such as "*the traffic and pedestrians do not both see a green light at the same time*". Formulated into propositional logic as $\neg(tlight.g \wedge plight.g)$; assuming the initial state is when *crossing* and *req* are both false and let the invariance be $\psi_{Inv} := \neg(crossing \wedge req)$, the proof formulæ 5 and 6 would become

$$\neg(crossing \wedge req) \wedge \varphi_L \wedge \neg(crossing \wedge req) \rightarrow \neg(tlight.g' \wedge plight.g')$$

and

$$\neg(tlight.g \wedge plight.g) \wedge \varphi_{\mathcal{L}} \wedge \neg(crossing \wedge req) \rightarrow \neg(tlight.g' \wedge plight.g')$$

which are provable.

1.3 Problem ② – Signalling Principles

Problem ② can be described as verifying that an interlocking ratifies with signalling principles. A full discussion of signalling principles and the translation into safety conditions is in Sect. 6. An example signalling principle would be "*points in a rail yard should not be set to the normal and reverse positions simultaneously*". Typically first order logic can be used to formalise signalling principles, the above condition would be formalised as

$$\forall pt \in Points : \neg(normal(pt) \wedge reverse(pt))$$

To resolve this principle into safety conditions, it is necessary to construct a topology model of the rail yard the interlocking is being verified for. This model is then queried for relevant information to facilitate the construction of safety conditions. Prolog facts and terms were used to build such models.

All rail yards are finite, i.e. *there are a finite number of points*. Thus, it is possible to replace universal quantification by a finite conjunction and existential quantification by a finite disjunction. Canonical rules are used to remove predicates from the resulting formula.

The final safety condition will typically be a large conjunction of more specific safety conditions (*as a result of removing universal quantification*) and verify that the interlocking ratifies with the signalling principle. To more precisely identify the cause of a counter example, if any, the final safety condition is split up into conjuncts, testing smaller portions of the system at a time.

For example, suppose a simple rail yard with two points $pt1$ and $pt2$, the above signalling principle would be translated into two safety conditions,

$$\neg(normal(pt1) \wedge reverse(pt1))$$

and

$$\neg(normal(pt2) \wedge reverse(pt2))$$

1.4 Implementation

Both problems have had solutions implemented, the first problem was solved using Haskell as it allowed for rapid development and is ideal for manipulating inductively defined tree structures, i.e. *formula's*. The ladder translation and proof formulæ construction has been successful in terms of feasibility, for a real world ladder with 331 rungs and 599 variables. The second problem was solved using Prolog and Java, which was also successful.

The basic architectures for both of these problems are shown in *Fig. 1.6* and *Fig. 1.7* respectively.

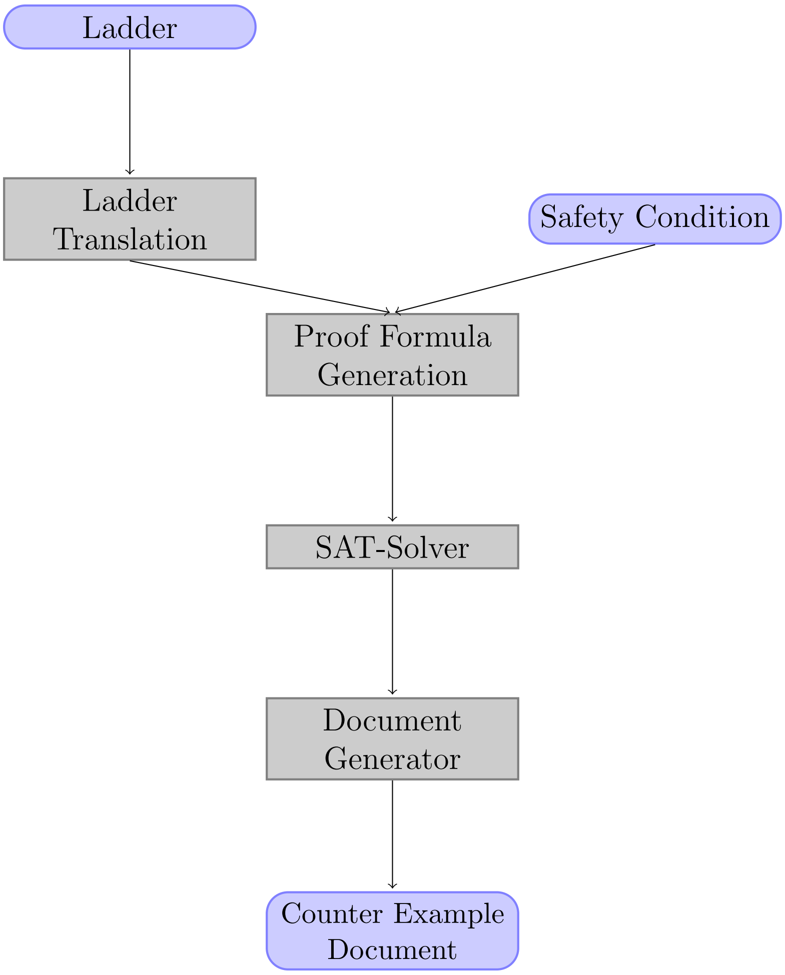


Figure 1.6: Problem 1 Software Architecture

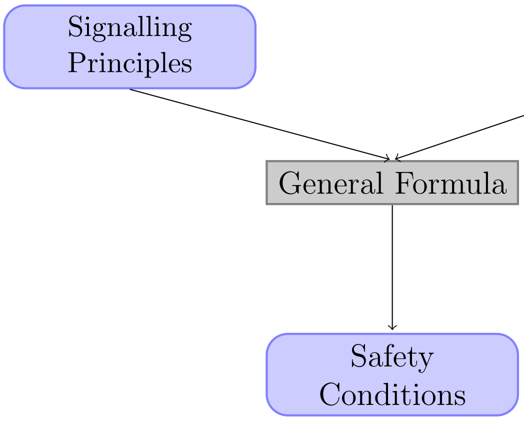


Figure 1.7: Problem 2 Software Architecture

Chapter 2

Background

2.1 History

There have been many attempts to formally verify railway interlocking systems, some have been successful as in the case of Banverket [Eri97b, Han94, FGHvV98] while others have not. In the cases where verification has not been successful, the failure is due to feasibility because verification is a complex problem and interlockings are complex systems.

Since the birth of British railways in 1826 there has been a tremendous effort directed toward controlling the trains that run on them in a careful way so that they do not collide, derail or deadlock. This was important because a single accident on the railway has the potential of killing many people. Trains and the railway infrastructure cost a lot of money and the reputation of a railway can easily be damaged when a train derails or worse. This control became known as signalling.

In the early days of signalling, before the 1840's policemen were responsible for ensuring safety by using a system of coloured hand held flags during the day and oil lit lanterns by night [KR01]. There was no communication between policemen, they simply relied upon a time interval system to prevent following trains from running into the train ahead. The policeman would show a red flag to following trains for 5 minutes after a train had passed him, and a green flag for a further 5 minutes. Only after more than 10 minutes had passed would a white flag be shown to approaching drivers. If a train stopped unexpectedly after passing a policeman and out of his sight, then the driver of a following train only had his own vision for warning.

During the 1840's mechanical signals were placed at stations and junctions, although, it was not clear where trains should stop when signalled to do so. Train drivers were responsible for selecting a good location to stop the train. These signals were initially controlled locally, but was soon discovered that groups of signals could be controlled from central locations by pulling leavers, *Fig. 2.1.* These central control locations created a new job of the signal operator. From these central locations, it was soon realised that specific combinations of these leavers should not be pulled simultaneously: i.e. *two opposing routes should not both be set.*

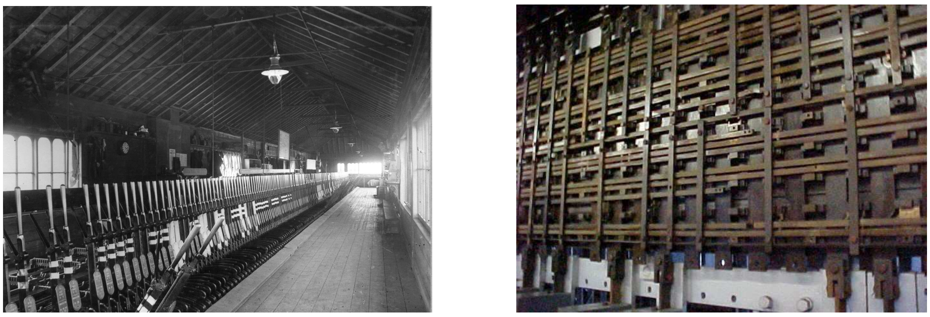


Figure 2.1: A traditional signal room (*left*) with the associated interlocking (*right*).

At Kentish Town[1] in 1860 a device which prevented specific combinations of leavers being pulled simultaneously was trialled. This device became known as an interlocking and would become the basis of railway signalling. During the same time period electronic communication between stations became possible. This allowed for the development of the *block section*, a signalling concept. Trains would travel along a train line; the next train was not allowed to follow the first until the signal operator gave it the all clear; the signal operator knew to give the all clear when the signal operator at the next signal sent a signal back saying that a train had passed. If this signal was not received in a reasonable amount of time, then it could be assumed that the train had broken down somewhere on the line.

Over the following years, much of the signalling hardware was upgraded to be electrical, notably during the 1920's mechanical signals were starting to be replaced by electronic versions and techniques were being tried which allowed for the position of a train on a line to be detected electronically.

It is clear that the mechanical interlockings were limited by the materials used. Coupled with the advent of valves and later during the 1920's relays, digital interlockings were being developed. Interlockings progressed through to microprocessors. It is worth noting that currently on the United Kingdom (*UK*) railways there are various types of interlockings in use. There are still

[1] London, UK

signal rooms such as the one in *Fig. 2.1* operating on smaller lines. The rational for keeping old interlocking systems in use comes from the idea that they have worked for many years and there is no real need to upgrade them as long as the railway yard they are operating on is not modified, such as adding new signals or lines.

The first digital interlocking built using microprocessors used in the UK was the “British Rail Solid State Interlocking” and this project was started in 1976, [Cri87]. The interlocking was first used (*piloted*) at Leamington Spa and was specified using the formal language Z, [Kin94]. This was a great success and paved the way for subsequent interlockings.

2.2 Railway Yards

Throughout this document the term railway yard is intended to mean a section of a railway such as a station or a depot. Railway yards are then connected by lines, *i.e. two adjacent stations are connected by a railway line.* The terms *railway yard* and *rail yard* are used interchangeably.

Typically a railway yard is made from components such as tracks, signals and points. The tracks are divided up into *track segments*, each track segment has an associated id and *track circuit* which is responsible for detecting if the track segment is occupied. Signals are also classified into subcategories with two important categories being main and distant. Distant signals display information about the next main signal on the line so that if the train needs to stop, it can slow down in advance and reduce the risk of a collision. There are other signalling schemes such as n-aspect signalling but these are ignored in this document. Each component is given an identifier which plays an important role in the verification, most railway operators use their own naming schemes for these identifiers which can complicate matters.

In *Fig. 2.2* there is an example station that is the terminus of a bidirectional line. Trains approach the station on the top line and enter one of the two platforms if vacant or they should wait at `ms1`. The train should then leave via the bottom line if `ms4` permits it. Track segment `ts2a` is a point, it comprises of two parts, the first connects `ts1a` to `ts3a` (*normal position*) and the second connects `ts3a` to `ts2b` (*reverse position*). The track segments `ts2a`, `ts2b`, `ts3a` and `ts3b` are points, `pt1`, `pt3`, `pt2` and `pt4` respectively. Typically, `pt1` and `pt3` should be treated as a single point, i.e. both points should be set to the same position, and move together when required.

Considering a map of the railway yard such as that in *Fig. 2.2*, it is clear that the concept of a train route can be defined as *“a sequence of adjacent track segments along with signalling information”*. This signalling

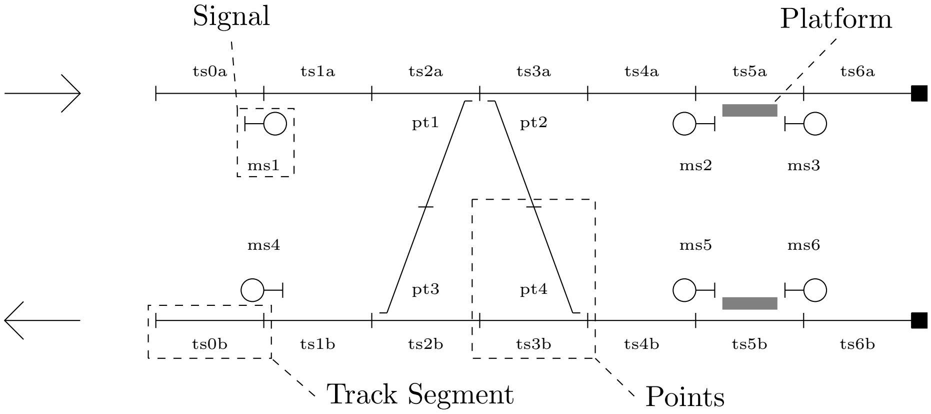


Figure 2.2: An example railway yard, all parts of the yard are named. The grey boxes on the right are platforms. The arrows on the left side indicate the direction trains are supposed to travel down the lines. The black boxes on the right are "end of line" markers. The "lollipops" named `ms1`, `ms2`, . . . , `ms6` are signals.

information specifies configurations for relevant signals and points. Train routes never extend through a main signal, but start and end at main signals.

When a railway yard is being designed not only are the topological aspects of how various components are connected, but also the operational protocols and safety properties are defined by means of *control tables*, see *Table 2.1* for an example. These tables contain the signalling information discussed above.

From *Table 2.1* it can be seen that route `A` takes trains from `ms1` to `ms3` and uses track segments `ts1a`, `ts2a`, `ts3a`, `ts4a` and `ts5a`; `ts6a` is required to be clear for overlap protection of the route. Overlap protection is where track segments following the route (nut not part of the route) are required to be unoccupied so that if the train does not stop in time, the risk of an accident occurring is mitigated. Both sets of points are set to the normal position. The signal aspect in the control table is not descriptive for this example, but with more complicated layouts where there are open lines or n aspect signalling[2] in operation, then the column describes the signal's behaviour. See *Fig. 2.3* for a graphical depiction of the four routes.

Not all possible routes have been listed but it could be assumed that there are at least two more routes; one of these routes takes trains from `ms4` to the next station and another route that terminates at `ms1` which brings in trains.

[2] n aspect signalling is not explained in this document as the actual implementation does not affect the verification in question.

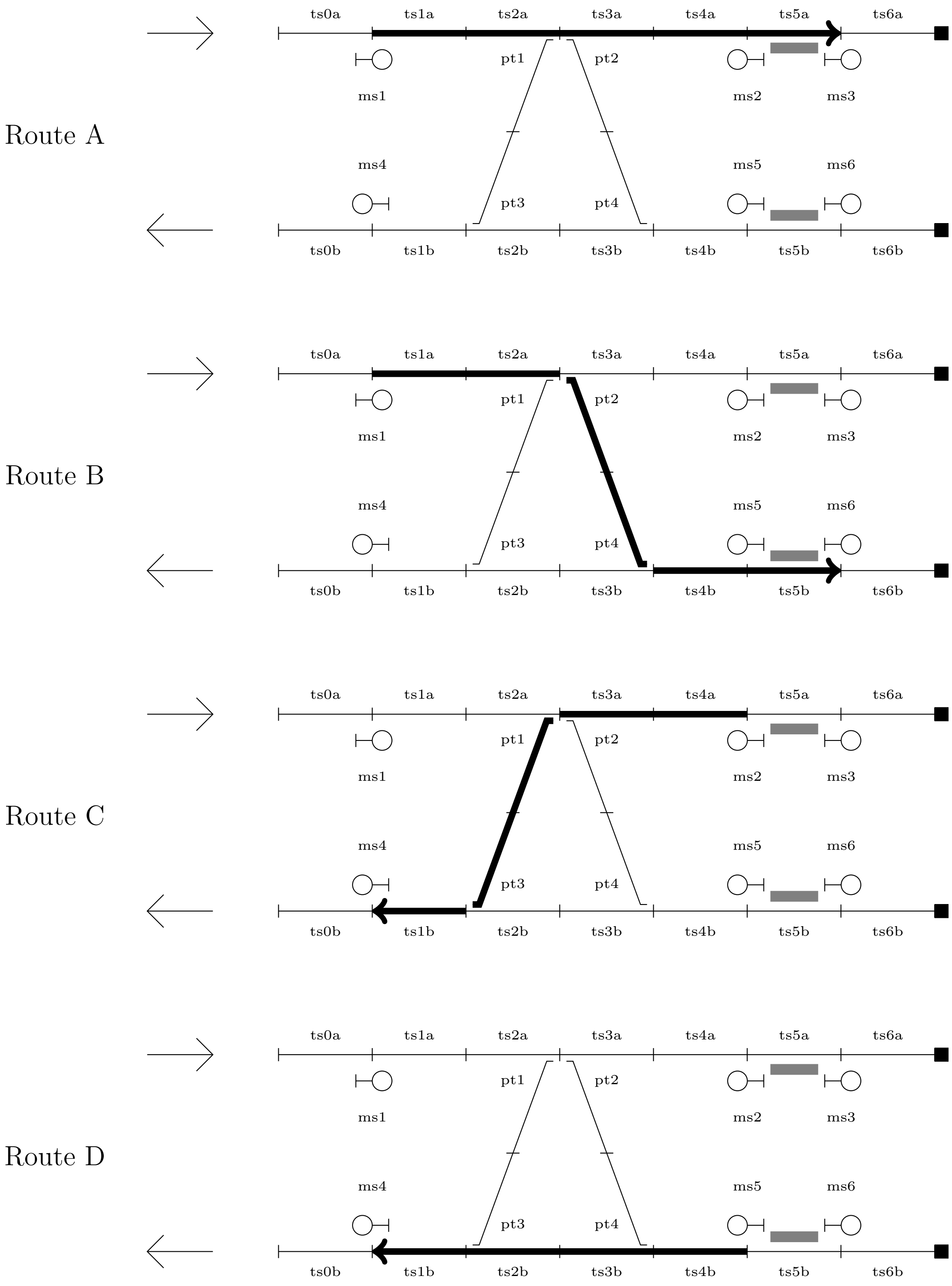

Route A
ts0a ts1a ts2a ts3a ts4a ts5a ts6a
pt1 pt2
ms1 ms2 ms3
ms4 ms5 ms6
pt3 pt4
ts0b ts1b ts2b ts3b ts4b ts5b ts6b
Route B
ts0a ts1a ts2a ts3a ts4a ts5a ts6a
pt1 pt2
ms1 ms2 ms3
ms4 ms5 ms6
pt3 pt4
ts0b ts1b ts2b ts3b ts4b ts5b ts6b
Route C
ts0a ts1a ts2a ts3a ts4a ts5a ts6a
pt1 pt2
ms1 ms2 ms3
ms4 ms5 ms6
pt3 pt4
ts0b ts1b ts2b ts3b ts4b ts5b ts6b
Route D
ts0a ts1a ts2a ts3a ts4a ts5a ts6a
pt1 pt2
ms1 ms2 ms3
ms4 ms5 ms6
pt3 pt4
ts0b ts1b ts2b ts3b ts4b ts5b ts6b

Figure 2.3: Graphical Representation of Routes from *Table 2.1*.

G = Green and R = Red

Route Name	Start	Exit	Signal Aspect	Condition	Track Segments	Points Normal	Points Reverse
A	ms1	ms3	G	Route Set	ts1a, ts2a, ts3a, ts4a, ts5a, ts6a	ts2*, ts3*	
			R	Route Unset			
B	ms1	ms6	G	Route Set	ts1a, ts2a, ts3a, ts3b, ts4b, ts5b, ts6b	ts2*	ts3*
			R	Route Unset			
C	ms2	ms4	G	Route Set	ts4a, ts3a, ts2a, ts2b, ts1b, ts0b	ts3*	ts2*
			R	Route Unset			
D	ms5	ms4	G	Route Set	ts4b, ts3b, ts2b, ts1b, ts0b	ts2*, ts3*	
			R	Route Unset			

Table 2.1: An example (incomplete) control table for the railway yard of *Fig. 2.2*. The 'start' and 'exit' columns indicate which signals the route starts and ends at; the 'track segments' column displays the track segments that must be unoccupied for a train to enter the route. The points columns together show the configuration which points must be in for a train to enter the route. ts$n$* stands for ts$n$a and ts$n$b.

2.3 Ladder Logic

Ladder logic[3] is a graphical language that is used to design integrated circuits; it is expressively equivalent to propositional logic, and as such there exists a canonical translation between the two languages [FHG+98]. Ladder logic consists of 3 operations: disjunction, conjunction and negation, and it can be shown that all other Boolean operations can be defined from these operations.

Typically, ladder logic consists of a sequence of rungs (*Boolean formulæ*); each rung can have one or more coils (*resultants*). These resultants can be fed into rungs lower down in the ladder as atomic propositions. For this project a restricted version of this ladder logic is considered; each rung has exactly one coil. This restriction helps to simplify the processing of the ladder but does not reduce the expressiveness as a rung with multiple coils can be split into multiple rungs with exactly one coil on each rung.

In order to define ladder logic, the set of propositional formulæ is defined as follows: atomic propositions p are propositional formulæ and if φ and ψ are propositional formulæ so are

[3] Ladder logic is defined in **IEC 61131-3** and **BSI EN 61131-3:2003**

- $\varphi \wedge \psi$,
- $\varphi \vee \psi$ and
- $\neg \varphi$.

An example ladder can be seen in *Fig. 2.4* and in *Fig. 1.5*. A strict partitioning of the propositional variables in the ladder can be defined, there are *inputs*, *outputs* and *latches*. *Latches* remember their value from one cycle to the next. *Outputs* are a special case of latches where their value is output to the real world at the end of every cycle. The data flow is depicted in *Fig. 3.2*.

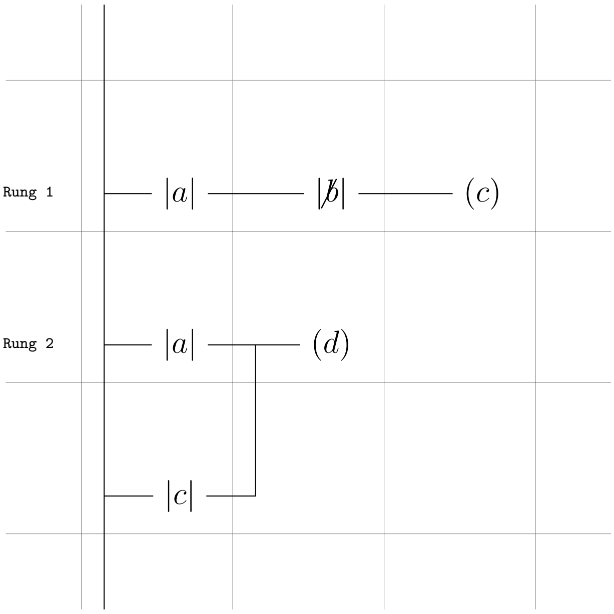


Figure 2.4: An example ladder logic diagram that depicts two rungs.

The ladder logic diagram depicted in *Fig. 2.4* is made from the following symbols:

Symbol	**Name**	**Meaning**
$\|x\|$	Normal	Represents the value of propositional variable x
$\|\not x\|$	Closed	Represents the negated value of propositional variable x
$\|x\|$——$\|y\|$		Represents conjunction $x \wedge y$, where x and y are propositional variables
$\|x\|$──┬─ $\|y\|$──┘		Represents disjunction $x \vee y$, where x and y are propositional variables
(x)	Coil	Represents the propositional variable that stores the result of the rung

where x and y are Boolean valued variables.

Table 2.2: Ladder Logic Symbols

Using the above translation rules, it can be derived that rung 1 is equivalent to $c := (a \wedge \neg b)$ and rung 2 is equivalent to $d := (a \vee c)$.

Rungs are represented as 2 arity tuples, ($\mathtt{a}$,φ), where the first element is the coil and the second element is a formula expressing the value semantics of the rung. Thus a ladder can be viewed as a list of tuples.

The example above is ladder logic in its simplest form; one complexity with ladder logic occurs because rung 1 is evaluated first, then rung 2. If rung 2 overwrites output c, then using standard propositional logic this can be represented by creating intermediary variables. This point is of great importance to a proof engine. The discussion of the translation will be reviewed in Sect. 5

Chapter 3

Techniques

Various mathematical techniques are applied to create a sound basis for the formal verification of the interlockings w.r.t. safety conditions, briefly, satisfying propositional formulæ along with the principle of induction. These concepts are introduced in the following sections.

3.1 Boolean Satisfaction Problem

Given a propositional formula built from propositions p_i and Boolean connectives, then in its simplest form the satisfaction problem is assigning to each of the p_i's a truth value such that the formula is satisfied. Suppose the following propositional formula:

$$(A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee B)$$

then there exists exactly one satisfying assignment, namely when $A \mapsto tt$ and $B \mapsto tt$.

S. Cook was the first person to explore the complexities of the Boolean Satisfaction Problem (*SAT-Problem*[1]); it was proven that the problem was in the complexity class *NP-Hard* [Coo71]. The implication of this proof is that the SAT-Problem is computationally equivalent to the hardest problems, therefore, by assumption[2] it will often be stated that the SAT-Problem is a hard problem. The reason for this is that every variable in a given formula has a choice of two potential values, so a formula with n variables will have an upper bound of 2^n assignments and there is no known decision algorithm that terminates in polynomial time for determining if a given propositional

[1] Also abbreviated to SAT.

[2] It is not known if there exists a polynomial space and time decision algorithm for SAT.

formula with n variables has a satisfying assignment. Every time a new unique variable is added to the formula, the search space is doubled. From this view it is simple to see that the complexity of solving a formula with $n-1$ variables is half that of solving a formula with n variables, *i.e. exponential complexity.* There are different, more intelligent methods of solving SAT but no polynomial algorithm has been found. There are classes of propositional formulæ that do have polynomial time decision algorithms [GK05, Kul08], but these are not of particular use for checking whether safety conditions hold in general.

3.2 Ladder Logic and Interlockings

The interlockings of interest are programmable using ladder logic; this program can be updated allowing for different operation. The ladder logic can be thought of as a Boolean program for the interlocking.

The execution of the Boolean program commences as follows [Wes06], formalised in *Fig. 3.1* and *Fig. 3.2*:

1. Variables are set to their initial values, usualy `false`.
2. Write outputs to the real world.
3. Read inputs from the real world.
4. Update the timers.
5. Sequentially evaluate each rung of the ladder, storing the result in the correct variable.
6. Goto step 2, there is no termination condition.

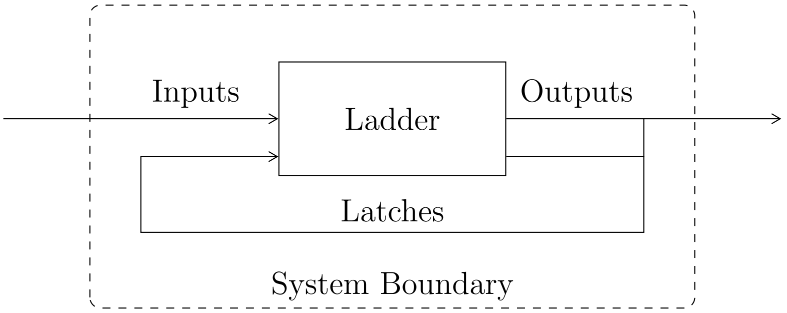


Figure 3.2: Model of the ladder execution as a *concurrent sequential process* style diagram.

```
/* entry point */
main() {
  initialise();
  executeladder();
}

initialise() {
  send outputs default values and
  setup initial state of variables
}

/*
 this is execution algorithm comes from the westrace, if it were to
 be designed by me then the output would be done after the loop,
 not before.
*/
executeladder() {
  while(true) {
    output();
    input();
    a0 := phi0;
    a1 := phi1;
    ...
    an := phin;
  }
}

output() {
  send outputs the logical values from the ladder
}

input() {
  read inputs from the world and store them to the ladder
}
```

Figure 3.1: Execution Strategy for Interlocking Ladders [Wes06]

This loop continues for ever unless a fault is detected at which time the interlocking will shut down, hence the loop is terminated violently.

To allow the logic to handle temporal aspects, timers are allowed; during the design stage a time period to time is specified. These timers are operated by two latches: the first latch $\texttt{timer}_a\texttt{In}$ starts and stops $\texttt{timer}_a$; the second latch $\texttt{timer}_a\texttt{Out}$ is read only and is set to true to indicate that $\texttt{timer}_a$'s time period has elapsed. See *Fig. 3.3* for an example where the timer's period is 2 cycles.

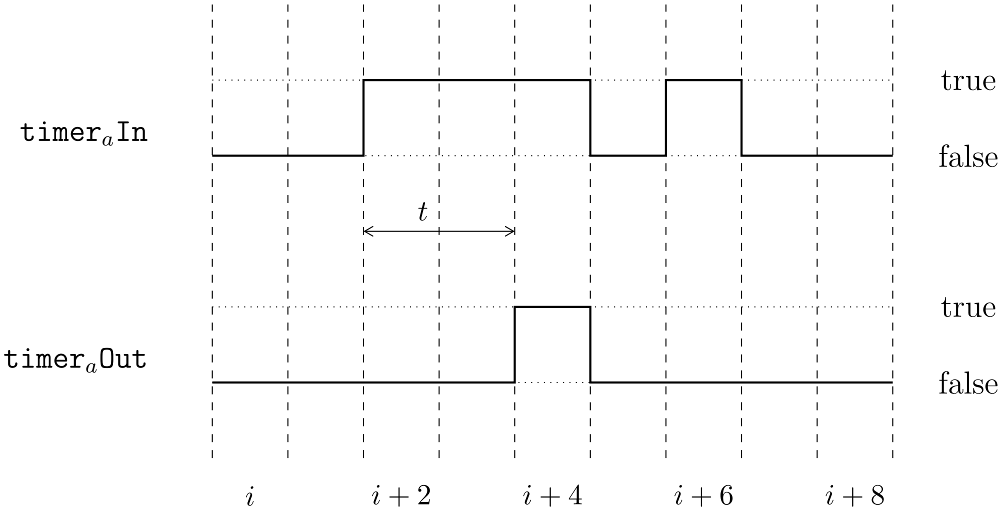


Figure 3.3: Timer interaction, t is the time period associated to $\texttt{timer}_a$.

When $\texttt{timer}_a\texttt{In}$ is set to true, $\texttt{timer}_a$ is started. If the time period t expires, then $\texttt{timer}_a\texttt{Out}$ is set to true as long as $\texttt{timer}_a\texttt{In}$ is true (as at time $i + 4$). But if $\texttt{timer}_a\texttt{In}$ is set to false before time period t expires, then $\texttt{timer}_a$ is stopped and reset.

All the resultants from the ladder are stored in a memory and are available for the next cycle. This is where values are placed when inputs are read from the real world and taken from when sent to the real world. A subtle note, each resultant of a rung and output is written to exactly once during each execution of the ladder. This completes the model of execution of interest to this project.

When attempting to formally verify that this Boolean program satisfies various safety conditions, there are two main paths, differing only in the granularity of the analysis: the first could be to use Hoare logic [Hoa69] directly; each rung would have a pre and post conditon and the rung would be the command. These pre and post conditons for all the rungs can then be linked together to provide a complete invariant of the ladder. The second, a more directly applicable method would be to translate the problem to that of the Boolean satisfaction problem, then apply Hoare logic to the whole ladder. The second approach is used for the project and Dr O. Kullmann is an expert in SAT who has provided much advice and guidance in this area of research [Kul08].

3.3 Ladder Invariants

The ladder consists of a finite set of atomic propositions L, and an ordered list of assignments for these propositions. Thus a ladder can be viewed as an imperative program P, where each program line of P is an assignment. The ladder shown in Sect. 2.3 would be equivalent to the following program fragment:

```
while(true) {
   /* do io operations */
   c $:= a \wedge \neg b$
   d $:= a \vee c$
}
```

More generally an arbitary ladder $\mathcal{L}$ with rungs (a_0,φ_0), ... , (a_n,φ_n) is equivelent to the following program imperative fragment:

```
while(true) {
   /* do io operations */
   $\mathsf{a}_0 := \varphi_0$
   $\vdots$
   $\mathsf{a}_n := \varphi_n$
}
```

Notation Assume a ladder with input variables

$$b_1, \ldots, b_m$$

Assume after the i^{th} iteration of the `while` loop the value of b_j is

$$b_j[i]$$

Then let the atomic propositional variables of the ladder be

$$a_1, \ldots, a_n$$

$a_k[i]$ is the value of the variable a_k after the i^{th} iteration. Strictly, $a_k[i]$ depends on the previous inputs to the ladder, written as

$$a_k^b[i]$$

where

$$b = \left(\begin{array}{c} (b_1[0], \ldots, b_1[i]), \\ (b_2[0], \ldots, b_2[i]), \\ \vdots \\ (b_m[0], \ldots, b_m[i]) \end{array} \right)$$

Assuming the ladder $\mathcal{L}$ has an initial configuration for the propositions in L. An invariant of the ladder is a property ψ expressed using propositional logic over the atomic propositions in L. Extending the notation above; semantics of $\psi[i]$ is defined as the same as that of ψ with each of the a_k substituted for $\mathsf{a}_k[i]$.

An invariant of the ladder is a formula that holds after arbitrarily > 0 many iterations of the while loop assuming arbitrary choices for the input variables b_1,…,b_m. Concluding the definition, if ψ is an invariant, then $\psi[i]$ must hold for all $i > 0$.

Formally an invariant ψ can be expressed as,

$$\forall b \; \forall i \; (i > 0) \wedge \psi(b_1[i], \ldots, b_m[i], a_0^b[i], \ldots, a_n^b[i])$$

Chapter 4

Literature Review

There have been many attempts to apply formal methods to railways and the interlockings. Banverket (Swedish National Rail Administration) has applied formal methods with great success; they have been working for over 10 years in this field [Eri97b, Eri97a, EF99]. The approach taken by L.H. Eriksson works by creating two models: the first is that of the interlocking and consists of rules and the second is of the railway yard for which the interlocking has been designed. For formal verification of these models, a proof engine produced by the company Prover[1] called NP-Tools was used [Eri97a]. NP-Tools is a collection of tools packaged with a proof engine; these tools translate various problems into an acceptable format for the proof engine to process them. NP-Tools has been used by many other companies for formal verification of critical systems such as ADTranz, Saab and Volvo.

Metra Transport International operates the Paris metro system. METEOR is a driverless metro line that is controlled by automatic train protection (*ATP*) and automatic train operation (*ATO*). The control software for this system is distributed over a large geographical region; some of the control is centralised, other parts are track side and on-board the trains. This was an extremely complicated system to develop, and the fact that it was a critical system means that there was no margin for error. Metra Transport decided to use the "B-Method" during development [BBFM99, Sab04]; B is a formal language, a successor to Z. B is a complex language to use partly because the specification and implementation are deeply intertwined, [Abr96]. Perhaps Metra Transport decided to use the "B-Method" as it was developed in France, so they had a good resource of local expertise. Using the "B-Method" Metra Transport were able to verify 100% of the safety and liveliness properties required of the ATP/ATO systems. It was claimed that

[1] Swedish based company started by Gunnar Stålmarck.

no bugs were found during the validation, in house testing, on site testing and since METEOR went live [BBFM99]. This is clearly an exceptionally good result for such a complicated critical system.

The Vienna Development Method (*VDM*) developed by International Business Machines (*IBM*) Vienna laboratory in the 1970's is one of the longest established formal methods for the development of computer based systems. Therefore, it is only fair to include a railway interlocking that has been formally verified using the VDM; K.Hansen has done just this [Han94]. The whole development cycle is very structured and clean, with clear conditions that should be proved to be correct. This development method was applied to the interlockings of a small number of Danish state owned stations, and the results were positive. The first attempt did not fulfill the requirements as there was no manual override control, *i.e. this made it very hard to shunt trains about as routes had to be allocated by the interlocking.* This was a simple matter to fix by adding an extra state to points which indicate if they are under control of the interlocking or manual control, [Han94].

In recent years, a grand challenge has been issued known as *TRain*, [Bjø04]. This challenge is to create *a domain theory for the railway infrastructure*. Although the grand challenge has a vastly wider scope than this project including topological, operational and tactical aspects some of the material is of interest. One such article providing a good overview of verification using temporal logic, interactive theorem proving and model checkers is by *Wolfgang Reif et al.* [ROTS04].

4.1 Topologies

There are many different data structures which can be used to model the topology of the railway yards. Below is a discussion of the main methods in use today. Using predicate logic, it is a simple matter to define n-ary relations between objects, a "*Prolog like*" syntax is suited very well to this. This is the approach taken by L.H. Eriksson when modelling the Swedish rail topology [Eri97b, Eri97a]. Simply put, consider two track segments, `ts1` and `ts2` then `connected_to(ts1,ts2)` and `connected_to(ts2,ts1)` are defined *iff* they are connected in the real world. Using objects and relations between them the real world can be modelled precisely. Assuming the objects and relations have applicable names; the model will be readable by humans.

Another approach taken by K. Hansen which originated from M. Monigel uses graph theory, [Han94, Mon92]. The basic idea is that each track segment is a vertex in the graph, each edge corresponds to two track segments being connected, *i.e. it is possible for a train to pass from one segment to the other.*

For technical reasons, generally the graphs are directed and doubly linked such that given two connected tack segments `ts1` and `ts2` which are vertices, then the edges `(ts1,ts2)` and `(ts2,ts1)` are defined. It is also possible to add signals to this definition by allowing each edge to have a label. If the label is set for a given edge, then it means "if the train is travelling along that edge in the given direction then the signal is visible". This is because signals can only be seen from one direction.

4.2 CNF Generation

Ladder logic can be canonically translated into propositional logic, but in general SAT solvers demand that the propositional logic be in Conjunctive Normal Form (*CNF*) as satisfiability is required. Using De-Morgan's laws a naïve translation can be defined but this has performance issues with a SAT-solver, because when the formula is translated, it explodes in size, although no intermediate variables are added. A secondary issue with naïve CNF translations relates to the loss of original structure in the formula, i.e. given two literals a and b, the following $a \leftrightarrow b$ would be translated into CNF as $(\neg a \vee b) \wedge (\neg b \vee a)$, whereas the optimal strategy would be for the SAT-Solver to constrain a and b to the same value [Kul08, FM07]. D. Sheridan and M.N. Velev have both shown alternate translations [She04, Vel04], Velev's translation is procedural and is intended for the formal verification of microprocessors. Sheridan's translation on the other hand uses graph theory and is intended for a more general translation of any propositional formula into a CNF formula that is intended for use with the SAT problem. Sheridan's translation, if provided with a list of identical sub-formulæ, can remove them and replace them by a variable for improved efficiency.

The *Tseitin* translation [Tse68] for converting an arbitrary propositional logic formula into CNF provides a good base algorithm, because it introduces new auxiliary variables for each node in the tree, and these are used to mirror the structure of the original formula so that SAT-Solvers can use this structure to help satisfy/falsify the formula. If logical equivalences are left in the formula base[2], particularly chains of equivalences, then the basic algorithm can be augmented to take advantage of this, mitigating the exponential clause explosion realised with the naïve method. Dr. Kullmann provides a detailed discussion of the exponential blow up concerned with problems with equivalence chains, also known as bi-conditional formulæ [Kul08].

[2] Formulas are built from the following base: $\wedge$, $\vee$, $\leftrightarrow$ and $\neg$.

4.3 SAT-Solvers

The following are informative on the current state of the SAT problem, [GOMS04, Kul08, PV04].

The criteria for choosing a SAT-Solver is not as simple as it might appear at first because different SAT-Solvers implement different algorithms and branching strategies. Thus, different encodings (*in the sense of adding auxiliary variables*) of a problem into CNF will require non-proportional amounts of time to solve on different solvers. Some solvers use the auxiliary variables if used correctly to speed up the solving.

The basic algorithm for the Boolean satisfaction problem is known as Davis-Putnam-Logemann-Loveland (*DPLL*), originally proposed in 1960 as DP, [DP60], then later as DLL, [DLL62], and finally combined into DPLL. The algorithm works by iteratively selecting a variable and satisfying/falsifying it until the clause set is satisfied or no more variables are left. In the second case, the algorithm will *backtrack* and try again with another new combination of satisfying/falsifying the literals. If this continues until there are no more combinations left, then the clause set is not satisfiable. With respect to the project, this last scenario where the clause set is not satisfiable translates to the fact that no counter examples found.

Generally there are three main classes of SAT-Solvers [Kul08]:

Backtracking The simplest class that attempts to assign values to variables; if the current (possibly partial) assignment yields a falsifying assignment, then backtracking is used. The DPLL satisfaction algorithm in its pure form is an example of a backtracking algorithm, and for obvious reasons there is an exponential search space. The following two methods are current *de facto* standards as they augment the basic backtracking approach.

Conflict-Driven Conflict driven algorithms add a conflict analysis to the basic backtracking such that clashes between literals are detectable, *i.e. when two constraints are identified, implying that x has to be satisfied and falsified at the same time.* When a conflict is identified, then the algorithm must backtrack as the current branch of the search tree is unsatisfiable. Also instead of having a fixed backtracking strategy, the algorithm can *jump* back to any branched literal in the search tree using a heuristic function. A final change that conflict driven algorithms have over basic DPLL is that they can restart randomly. The MinSat solver is an example of a conflict driven SAT-Solver. Conflict driven SAT-Solvers have been shown to be good at processing large clause sets which are simple in nature, and this type of data occurs in industry.

The rationale for random restarts comes from the principle that if the clause set is satisfiable, then the solution should be easy to find, so if the solver has been running down a pointless branch for a long time then restarting will hopefully reduce the total time taken to find a solution.

Look-Ahead The look ahead algorithms perform higher levels of reasoning by looking at the structure of the clause set, typically translating the CNF into a graph. These solvers are still based upon the DPLL. A look ahead SAT-Solver can tackle hard[3] and random clause sets but struggles on industrial problems in comparison to the conflict driven solvers. The OKSolver is an example of a look ahead SAT-Solver. The unit clause/boolean constraint propagation algorithm is an example of look-ahead. Research has shown that SAT solvers spend roughly 90% of the time performing unit clause propogation, [Alo06].

G. Pan and M.Y. Vardi discuss the implications of searching as described above or using symbolic model checking for the purpose of SAT solving [PV04].

The answer to the question about which criteria should be used to select a SAT-Solver is unclear. Whether there is a simple answer, or if lots of experiments with different SAT-Solvers with different clause set generation techniques is the only method. Conflict driven SAT-Solvers would be a natural choice due to their apparent applications to industrial problems. The ambiguity these questions raise reflect the lack of human knowledge of the underlying structure of the SAT problem.

4.4 Safety Conditions

Safety conditions are directly derivable from the control tables. Therefore, assuming the control table can be read automatically, these conditions can be generated automatically. W. Fokkink shows how these safety conditions can be produced mechanically, [FHG+98]. This type of verification verifies that the interlocking implements the control tables correctly, hence transitively safety conditions are met.

Signalling principles are also provable, L.H. Eriksson gives a detailed discussion about how this can be implemented by creating a formal model of the railway yard [Eri97a] and suggests a formal language to reason about the model [EF99].

[3] Hard in the sense that they contain many more constraints to satisfy not referring to the size of the clause set. Industrial problems are typically large but have fewer constraints.

4.5 Interlocking Specification Languages

There have been many attempts to produce an Interlocking Specification Language (*ISL*), and these languages are built on a formal language. One of the most notable languages is LARIS 1.0 developed at the CWI, Amsterdam [FGHvV98]. LARIS is an acronym for LAnguage for Railway Interlocking Specifications. LARIS is a typed version of the graphical language EUropean Railway Interlocking Specification (*EURIS*). In the latter language the whole interlocking is modularised; each module can send messages to other modules or the real world, and modules can also receive messages from the real world. All these modules are assumed to be asynchronous, therefore, the order the messages are processed is not fixed.

EURIS consists of a set of four sub-languages; three of these languages are used to model the railway yard and the routes. The final language models how the actual entities, such as a signal, operate. EURIS lacks a formal mathematical background, thus proving safety conditions is very difficult due to ambiguity. An attempt has been made to verify safety properties of a EURIS program by translating the program into a Petri net which has a precise mathematical meaning. There are many tools which can be used to simulate and show properties of Petri nets, one such tool is *ExSpect*. A thorough discussion of this technique can be found in [BBV95].

L.H. Eriksson has suggested that such a language should be proprietary free and designed independent of any specific interlocking [EF99]. This also translates to the problem of modelling concurrent sequential processes (*CSP*); the approach taken by LARIS.

This project does not require the use of an ISL although the syntax introduced by L.H. Eriksson is appropriate for formalising signalling principles, [EF99]. See *Fig. 4.1* for an example of this language. The language is very similar to first order logic (*FOL*) as it uses quantifiers and Boolean connectives. The predicates would be replaced by their definitions or atomic propositions to produce a proper FOL formula.

(A) `for all points p (part_of(p,rt) and route_locked(rt))` → `locked(pt)`

(B) $\forall p(\mathit{point}(p) \wedge \mathit{part\_of}(p, rt) \wedge \mathit{route\_locked}(rt)) \rightarrow \mathit{locked}(pt)$

Figure 4.1: An example signalling principle from [EF99], (B) is the first order formula of (A).

4.6 First Order Logic

The formal syntax of the language introduced in Sect. 4.5 is clearly an extension of propositional logic, where quantifiers have been added to the language. This is known as First Order Logic (*FOL*). Informally, the semantics of FOL can express that a property holds for *all numbers* or *some numbers*, [vD04].

It is a requirement to remove such judgements from a formula to construct a clause set; there are various methods. A common approach is to use *Herbrand* and *Skolem* functions, [Bus94, Her71]. The Herbrand function removes universal quantification from a FOL formula, whereas Skolem function removes existential quantification from a FOL formula. Combining both of these functions allows for all quantifiers to be systematically removed assuming that it is possible to select an x for the Skolem functions.

The removal of existential quantification causes problems, the only way to select such an x, if one exists, will, in general, require solving of formulæ (*signalling principle*) by "cloning" the formula for each possible selection of x and taking a disjunction of all such clones.

The variables in the FOL of safety conditions range over finite sets of rail yard objects, some real such as *Track Segments* and some abstract such as *Train Routes*. A simple translation is applied, one that does not use Herbrand and Skolem functions. Each universal quantification is substituted by a conjunction with a finite number of conjuncts, one for each possible value the variable can range over substituted into the matrix[4] of the formula. For example

$$\forall\ x \in \mathit{Points}\ \varphi(x)$$

could become,

$$\varphi(p_1) \wedge \cdots \wedge \varphi(p_n)$$

where *Points* is a finite set of n point id's $p_1, ..., p_n$.

Dually the existential quantification can be removed by a complementary process that substitutes the existential quantifier by a disjunction, one disjunct for each possible value that variable can range over substituted into the matrix of the quantification. For example:

$$\exists\ x \in \mathit{Points}\ \varphi(x)$$

could become,

$$\varphi(p_1) \vee \cdots \vee \varphi(p_n)$$

[4] Given $\forall x \psi(x)$, $\psi(x)$ is the matrix.

where *Points* is a finite set of n point id's $p_1, ..., p_n$.

From these translations it is apparent the different meanings of these two forms of quantification. The first requires that a property φ holds for all x. The second requires that a property φ holds for at least one x.

When a signalling principle is translated into propositional logic there is an issue of "cutting" up the formula such that each instance of the signalling principle can be tested independently. When given a signalling principle $\psi := \forall\ p \in \mathit{Points}\ \varphi(p)$, this would have the semantics

for all points p, the safety condition φ holds

In practice, if the safety condition does not hold for a given point p, then the signalling principle is falsified without indicating for which point it failed[5]. The following would be a better series of conditions:

the safety condition holds for point 1, and
the safety condition holds for point 2, ...
the safety condition holds for point n.

Although they are essentially equivalent, the second form allows for each safety condition to be tested by the solver individually. To produce the second form, the formula needs to be placed into prenex normal form[6]. Then the translation above is applied in order to remove quantifiers. Finally, all the top level conjunctions are broken down such that there is a list of conjuncts which do not have as a root, a conjunction. This is the list of safety conditions that are to be proven from the signalling principle.

4.7 Railway Signalling Principles

The following books were provided by Invensys, they detail the operations of the British railways. However not a full picture of all signalling principles is given as each different operator can use variations of these principles mainly due to historic reasons.

- Introduction to Railway Signalling - A simple to understand book which introduces many of the techniques and various hardware which is used on the railway. [KR01]

[5] Although deep analysis of the counterexample should yield this information.

[6] Let φ be a FOL formula, a prenex normal form of φ is a formula obtained from φ by first renaming bound variables in φ so that no variable is quantified more than once in φ and then using the axioms of the Prenex translation to move all quantifiers to the outside.

- Railway Signalling - A more technical and advanced book which covers many of the topological aspects of the railway and “philosophical” aspects of the signalling principles. [Noc02]

- Railway Control Systems - A supplement to the above book, explains techniques such as level crossing and train detection, and also covers “philosophical” aspects of signalling. [Lea03]

Chapter 5

Ladder Logic Translation

To produce a propositional model of the ladder logic, a translation is required. This section describes how to translate the Graphical Configuration Sub-System (*GCSS*) file format into a propositional formula.

The ladder logic when read from the GCSS file can be described as a list of rungs, where each rung is a list of cells. A cell has the following information: coordinates x, y, what type of cell is it, which of the adjacent cells it is connected to and an identifier. This information can be represented as a 5-arity tuple.

$$(x, y, \mathit{type}, \mathit{id}, \mathit{links})$$

where x, $y \in \mathbb{N}$, *type* ranges over types in *Table 5.1*, $\mathit{links} \subseteq \mathit{Links} := \{Top, Bottom, Left\}$ and *id* is a possibly empty String, which in the case of *Normal*, *Closed* and *Coil* cell types refers to the literal name.

Links indicate where the cell gets its inputs from, making the simplification that there are only $2^{|\mathit{Links}|} = 8$ different combinations which *links* can be, so $\mathit{links} \in \mathbb{N}_8$. These are depicted graphically in *Fig. 5.1*, the right link is implied by the cell type, see *Table 5.1.*

The following equivalences can be observed between types:

$$\mathit{Ladder} \equiv \mathit{list}(\mathit{Rung}) \equiv \mathit{list}(\mathit{list}(\mathit{Cell}))$$

and

$$\mathit{Cell} \equiv \mathbb{N} \times \mathbb{N} \times \mathit{Type} \times \mathcal{L} \times \mathbb{N}_8$$

where $\mathcal{L}$ is a finite, non-empty set of atomic propositions such that $\mathcal{L} \subset \Sigma^*$, where Σ is a given alphabet.

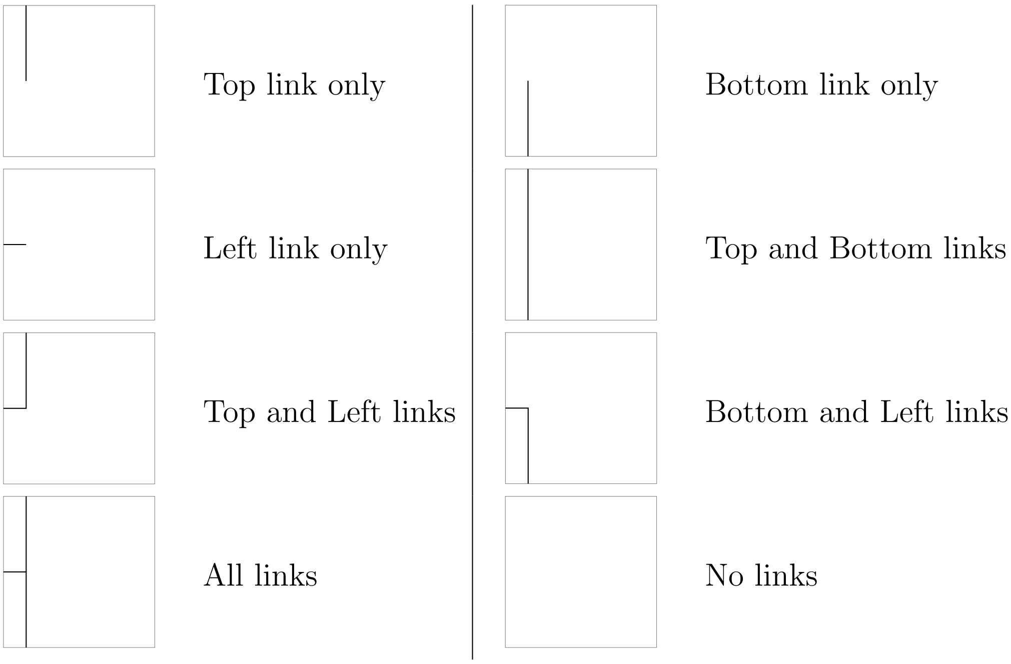


Figure 5.1: Cell Link Graphical Representations

The rung in *Fig. 5.2* can be encoded as

$$\begin{array}{lcl}
[& (1,1,Normal,a,[top,bottom]) & , \\
 & (2,1,Normal,c,[bottom,left]) & , \\
 & (3,1,Coil,e,[bottom,left]) & , \\
 & (1,2,Normal,b,[top,bottom]) & , \\
 & (2,2,Empty,,[top,left]) & , \\
 & (3,2,Empty,,[top,bottom]) & , \\
 & (1,3,Empty,,[top,bottom]) & , \\
 & (3,3,Empty,,[top,bottom]) & , \\
 & (1,4,Horizontal,,[top,bottom]) & , \\
 & (2,4,Closed,d,[left]) & , \\
 & (3,4,Empty,,[top,left]) &]
\end{array}$$

Outline The translation works as follows: starting with the first rung, find the coil; working backwards from the coil, follow all the different pathways back to where the x coordinate is one yields the correct propositional formula, Sect. 5.1; move on to the next rung and continue to the last rung; rename the literals in the rungs to force a model of sequential execution, Sect. 5.3,

Type	Symbol	Description
Normal	$— \lvert x \rvert —$	A cell with a single propositional variable.
Closed	$— \lvert \not x \rvert —$	A cell with a single negated propositional variable.
Horizontal	$———$	Horizontal link from cell on right, does not need to link to left cell.
Coil	$— (x)$	Resultant of the rung, a propositional variable. Invensys only allow there to be one per rung.
Empty		An empty cell, has no semantical meaning during the translation. Only provides placeholder for links, used primarily for *corners* and *vertical links*.

Table 5.1: Cell Types. The cells not link the left side of the cell. Coil and Empty do not link to the right cell. Combines with the links in *Fig. 5.1* to provide a complete graphical model of various cell configurations.

finalise the translation by taking a conjunction of all these rungs. The result is a model of the ladder in propositional logic.

5.1 Rungs

The goal of this section is to show the translation from a list of elements of *Cell* to a 2-arity tuple where the first element is the literal from the coil and the second is a propositional formula, the value of which is assigned to the coil.

The *Cell* type is ambiguous; it is possible to have repeated occurrences of different cells at the same coordinates. Ladder logic adds the constraint that for any coordinate there is at most one cell. Thus defining a partial function

$$rung_i : \mathbb{N} \times \mathbb{N} \overset{\sim}{\rightarrow} Type \times \mathcal{L} \times \mathbb{N}_8$$

to constrain the input, where i is the rung index.

The software used to produce these diagrams enforces other constraints upon each rung; a *coil* must be placed in the top row and can not be “shorted out”, *i.e.* can not be directly connected to the left hand column without going through a *normal* or *closed* cell type.

The data type $\mathbb{N}_8$ is an encoding of the links possible for the cells; to help with subsequent definitions, a new relation can be defined,

$$linked_i : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$$

$$linked_i(x_1, y_1, x_2, y_2) := \begin{cases} tt & \text{if first cell } (x_1, y_1) \text{ is immediately linked to} \\ & \quad \text{second cell } (x_2, y_2) \text{ in rung } i, \\ ff & \text{otherwise} \end{cases}$$

where i is the rung identifier. *linked* is not symmetric as there are no right links. I.e.

$$linked_i(x_1, y_1, x_2, y_2) \not\Leftrightarrow linked_i(x_2, y_2, x_1, y_1)$$

The next step is to select what to do for each type of cell, only left, top and bottom links are considered branches from the current cell.

Normal This is a conjunction of the literal with a disjunction of all the branches.

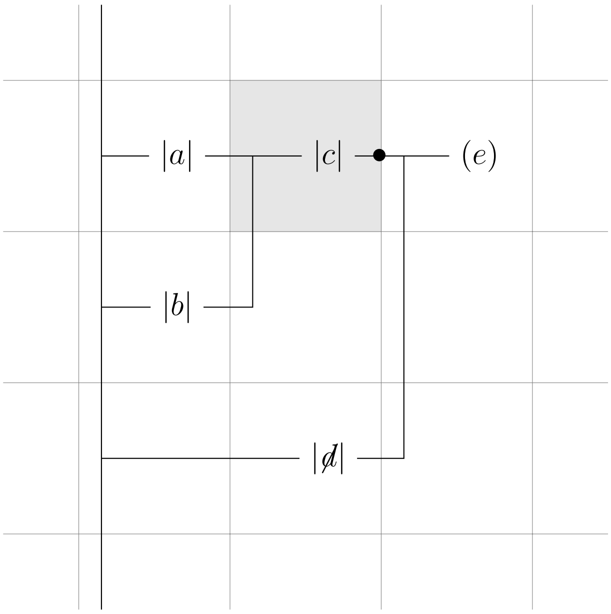
Closed This is a conjunction of the negated literal with a disjunction of all the branches.

Horizontal Disjunction of all branches from the cell (always travelling left).

Empty These cells should never be reached when translating into propositional logic, thus have no logical meaning.

There are two strategies that can be applied to yield the translation. These are graphically represented in *Fig. 5.2* and *Fig. 5.3*. The optimised strategy, is better because the result is *part way* to being a CNF.

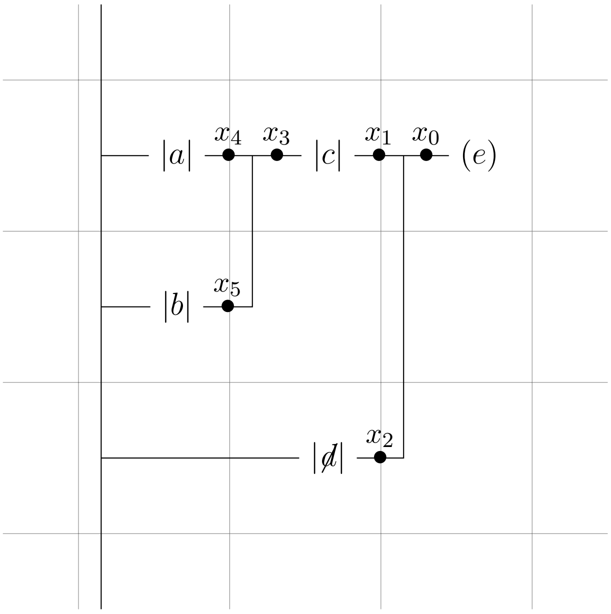
(A) *Naïve*



$$e := \neg d \vee (c \wedge (a \vee b))$$

Figure 5.2: Naïve Translation Strategy

(B) *Optimised*



$$\begin{array}{rcl} x_5 & := & b \\ x_4 & := & a \\ x_3 & := & x_4 \vee x_5 \\ x_1 & := & c \vee x_3 \\ x_2 & := & \neg d \\ x_0 & := & x_1 \vee x_2 \\ e & := & x_0 \end{array}$$

Figure 5.3: Optimised Translation Strategy

The *naïve* translation works in reverse, starting at the right most column where the coil is and working back to the start column. Each cell which is not empty, *such as a coil*, has a formula ϕ that is a disjunction of all cells in the column immediately right of the current column which are reachable from the current cell without travelling right, i.e. *travelling up or down a number of times and then once left.* If the current cell is not a *coil* and contains a non-negated propositional variable x, then the formula assigned to that cell would be $x \wedge \phi$ and if it is a negated propositional variable x, then the formula assigned to that cell would be $\neg x \wedge \phi$, otherwise ϕ is assigned to the cell. The result is similar to the *optimised* translation without introducing auxiliary variables.

For example, in *Fig. 5.2*, at the bullet mark the formula assigned to the shaded cell would be $(a \vee b) \wedge c$. If the propositional variable c in the shaded cell was negated, then the formula assigned to the shaded cell would be $(a \vee b) \wedge \neg c$

The *optimised* translation, see *Fig. 5.3*, keeps the structure of the ladder intact but adds auxiliary variables. These variables are used to mimic the structure, while at the same time simplifying the rung by splitting it up into smaller rungs. A conjunction of the smaller rungs can be used to start producing the CNF of the rung which is required by the solvers; all that is needed is to convert the smaller rungs into CNF formulæ. Formally, the optimised translation starts on the left hand column and progresses through to the right hand column where the coil is, building sub-formulæ up for each required cell. The required cells are cells which contain normal latches, negated latches or vertical links (a disjunction). Each sub-formula is assigned a fresh variable x_i to store its result. These variables are placed immediately after each normal or negated latch and immediately after a disjunction. The x_i's are used in place of the actual formulæ, thus each sub-formula consists of at most one operation and two variables.

Both translation strategies will terminate assuming the rung is finite as at every step the column index is incremented or decremented depending on the strategy. The *naïve* translation will terminate; eventually the algorithm ends up at column 1, similarly for the *optimised* strategy. If the rung is infinite, then the *optimised* translation will obviously never terminate and the *naïve* version will never start as it will never find the coil.

5.2 Naïve Translation

The remainder of this section will focus on the *Naïve* translation as this was the technique which was implemented in this research as the optimised translation was not discovered until after the implementation was done.

Translation works back from the coil to the left hand column, constructing the formula as it goes. A critical part of the translation is deciding which cells to visit from the current cell. Typically a given cell (x, y) in the rung has its formula built up by examining the connected cells in the $x-1$ column. This gives rise to a new function,

$$next_i : \mathbb{N} \times \mathbb{N} \rightarrow set(\mathbb{N})$$

which returns a set of y coordinates indicating cells in the $x-1$ column that are connected to the current cell, where i is the rung identifier.

A cell (x_1, y_1) is *connected* with another cell (x_2, y_2) if cell (x_2, y_2) is reachable from the first cell (x_1, y_1) by travelling up or down an arbitrary (possibly 0) many times then left once. The notion of travelling as used here requires that there are cell links to travel along, i.e. if the current cell does not have a *top* link, then it is not possible to travel *up* an arbitrary number of cells. Formally, *conn* and can be defined as,

$$conn_i : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$$

Where i is the identifier of the rung and takes two pairs of cell coordinates, the first is the cell in the x^{th} column and the second is the cell in the $x^{th} - 1$ column.

$$conn_i(x_1, y_1, x_2, y_2) := \begin{cases} linked_i(x_1, y_1, x_2, y_2) & \text{if } y_1 = y_2, \\ conn_i(x_1, y_1 + 1, x_2, y_2) & \text{if } y_1 < y_2 \land \\ & \quad linked_i(x_1, y_1, x_1, y_1 + 1), \\ conn_i(x_1, y_1 - 1, x_2, y_2) & \text{if } y_1 > y_2 \land \\ & \quad linked_i(x_1, y_1, x_1, y_1 - 1), \\ ff & \text{otherwise} \end{cases}$$

The constraint of $x_1 = x_2 + 1$ is not explicitly defined but is implied by the first clause, the second and third clauses are used for vertical traversal.

Concluding the definition of $next_i$,

$$next_i(x, y) := \{z \mid z \in \mathbb{N}, conn_i(x, y, x - 1, z)\}$$

The resultant set is finite, as a given rung has a finite number of cells.

To define the *naïve* translation, the use of an auxiliary function is required which searches for the *coil* in a given rung i. The type signature is,

$$findCoil : \mathbb{N} \overset{\sim}{\rightarrow} \mathbb{N} \times \mathbb{N}$$

such that $findCoil(n)$ is a pair (x, y) indicating the coordinates of the coil in rung n if one exists, otherwise undefined. The actual *naïve* translation has the following type,

$$\textit{naïve} : \mathbb{N} \rightarrow \mathcal{L} \times Formula$$

and implementation,

$$\textit{naïve}(i) := (\pi_2(rung_i \circ findCoil(i)), \textit{naïve}' \circ findCoil(i))$$

where π_2 projects the second argument from a tuple and *naïve*$'$ performs the actual translation by taking a disjunction of all the connected cells along with the current cell operation.

$$\textit{naïve}'_i : \mathbb{N} \times \mathbb{N} \rightarrow Formula$$

$$\textit{naïve}'_i(x, y) := \left(\begin{array}{l} (\bigvee \{\textit{naïve}'_i(x', y') \mid (x', y') \in (\{x - 1\} \times next_i(x, y))\}) \\ \qquad \wedge\ cellop_i(x, y) \end{array} \right)$$

where *cellop* is the operation of the current cell and can be defined as,

$$cellop_i : \mathbb{N} \times \mathbb{N} \rightarrow Formula$$

$$cellop_i(x, y) := \begin{cases} tt & \text{if cell } (x, y) \text{ has type is } \textit{coil} \text{ or} \\ & \quad \textit{horizontal} \\ \pi_2(rung_i(x, y)) & \text{if cell } (x, y) \text{ has type is } \textit{normal} \\ \neg\pi_2(rung_i(x, y)) & \text{if cell } (x, y) \text{ has type is } \textit{closed} \end{cases}$$

The first case has no effect on the generated formula as *cellop* is only used to build up a conjunction in *naïve*$'$. The other two cases project the literal name out from the underlying data structure and negate it accordingly. This completes the *naïve* translation of each rung in the ladder to propositional logic.

5.3 Renaming

Renaming of variables in the rungs is of great importance; it finalises the translation into propositional logic such that the result is a model of a Boolean program with an execution strategy as described in Sect. 3.2 and [Wes06].

Assuming the ladder has been translated into a list of pairs, *i.e.* by applying *naïve* to each rung of the ladder, preserving the original order of the rungs. The goal is to produce a single propositional formula that models the whole ladder. For example, assume that the ladder is as follows:

$$\begin{array}{lcl} [& (a, b \wedge c) & , \\ & (b, \neg b) & , \\ & (c, a \vee b) & , \\ & (a, c) &] \end{array}$$

This should be translated to

$$\begin{array}{lrcl} & (a' & \leftrightarrow & b \wedge c) \\ \wedge & (b' & \leftrightarrow & \neg b) \\ \wedge & (c' & \leftrightarrow & a' \vee b') \\ \wedge & (a'' & \leftrightarrow & c') \end{array}$$

where the *primes* and *double primes* are fresh variables.

Before formalising the translation, the following definitions are required. Let $\mathcal{L}$ be a finite set of literals and the type *Sub* be the type of substitutions, i.e. $\mathcal{L} \to \mathcal{L}$. The function

$$(a \mapsto a') : Sub$$

$$(a \mapsto a')(x) = \begin{cases} a' & \text{if } x = a \\ x & \text{otherwise} \end{cases}$$

creates substitutions, i.e. *replaces propositional variable a by a'*. $subst(s, \varphi)$ applies substitution s to the formula φ, formally $subst(s, \varphi)$ is the result of replacing all atomic propositions p in φ by $s(p)$. The identity function is required for technical reasons, defined as

$$id : Sub$$

$$id(p) := p$$

The algorithm used for the translation is inspired by Robert Milner's algorithm for deriving types in polymorphic programs, [Mil78]. The translation is formalised by defining a function *rename*, the function recursively builds up a substitution. Before defining *rename*, *rename'* is defined as

$$rename' : list(\mathcal{L} \times Formula) \to Sub \to list(Formula) \times Sub$$

The result of *rename'* is a pair; the first element is a list of the rungs renamed and the second is a substitution that specifies the renaming. The substitution is reused for producing the safety conditions in Sect. 6.3. The function works as follows: take the first rung and produce a new variable a'

that does not occur in the ladder for the rungs resultant a. Substitute in all subsequent rungs and the resultant of the current rung a for a', written as $[a := a']$; repeat the process in order on the subsequent rungs.

Given a fresh variable a', and let $(\mathit{fs}', s') = \mathit{rename}'(\mathit{fs}, s \circ (a \mapsto a'))$, the implementation of rename' is

$$\mathit{rename}'(\mathit{cons}\big((a, \varphi), \mathit{fs}\big), s) := (\mathit{cons}\big(a' \leftrightarrow \mathit{subst}(s, \varphi), \mathit{fs}'\big), s')$$

To utilise the rename' function, the top level call should be initialised with the identity substitution. While stepping through the rungs each atomic proposition a is added to the substitution chain.

Finally, the rename function is defined as

$$\mathit{rename} : \mathit{list}(\mathcal{L} \times \mathit{Formula}) \to \mathit{list}(\mathit{Formula}) \times \mathit{Sub}$$

$$\mathit{rename}(\mathit{ladder}) := \mathit{rename}'(\mathit{ladder}, \mathit{id})$$

Chapter 6

Safety Conditions

6.1 Signalling Principles

Signalling principles are a method of defining safety conditions between abstract *entities*. All entities have types[1] associated with them. These types can be arranged hierarchically, as shown by Eriksson in [Eri97b]. This technique is known as *sub-typing* which is used in object oriented (*OO*) programming where all classes (*types*) derive directly or indirectly from the base `Object` type. Each different type of entity has different properties and relations to other entities, and this information along with the parent entity type makes up the entity type.

The basic entity type *yardobject*[2] is extremely general, and all other entities are related directly or indirectly to this one. The exact hierarchical type structure depends on the real system being modelled and has implications regarding the semantics of the signalling principles. Typically all railway yards can consist of track segments, points and signals. There are many different types of signals, thus, the *signals* entity type consists of all the different kinds of signals. *Signals* can also be split up into sub types[3] *main signal* and *distant signal*, where a distant signal is one which provides information about the next main signal on the route so that a train has time to stop if required. These can also be split up into more types that represent actual types of signals, i.e. *stop*, *3 aspect*, *2 aspect* or *n aspect* signal types. The correct type hierarchy of signals is not clear, should the *stop signal* entity type be a sub type of *main signal* or should *main signal* entity type be a sub type of *stop signal* as all signals have a *red* aspect. The approach taken by Eriksson was

[1] Similar to normal programming, where variables have types, i.e. `int` or `bool`.

[2] Equivalent to `Object` type in traditional OO programming.

[3] Can also be thought of as a specialisation of the *signals* type.

to make *stop signal* a super type of *main signal*. See *Fig. 6.1* for the basic hierarchical structure of the entity types.

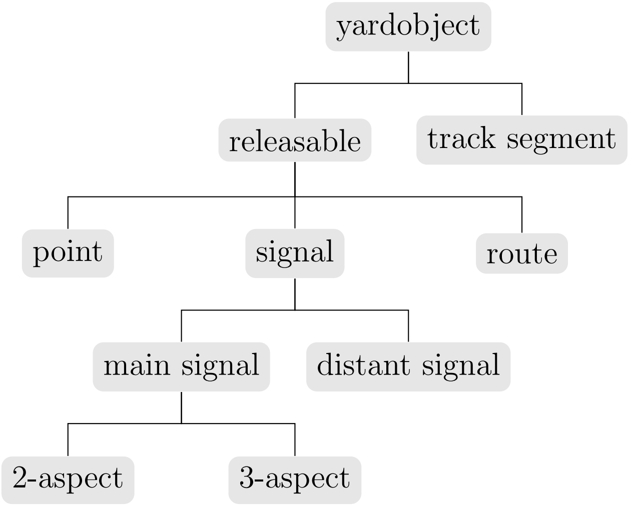


Figure 6.1: Basic Hierarchy of Entity Types

Let pt be a point, formally $pt \in point$ then by a transitive relationship $pt \in releasable$ and $pt \in yardobject$. These relationships are often called *isA*[4], i.e. *pt isA point* and *pt isA yardobject*.

Signalling principles can be viewed as specifying a single safety condition that applies for different entities of the given types. The restriction of an entity type is required to ensure that only compatible entities have to satisfy the safety condition. Suppose that φ is a signalling principle that has the semantics, *when a route is set then all points in the routes should be locked*, without the constraint that the first entity is a route and the rest are points, absurd safety conditions could be conceived. The first entity requires that it has the property of being *set*, while the remaining entities requires that they can be *locked*; these types can be added to the hierarchy if needed.

Thus a signalling principle has two parts: the first is the actual abstract condition expressed as a formula and the second part details which entity types the condition is applicable to. Entity types are essentially finite *sets* of real entities hierarchically organised using the $\supseteq$ operator. *e.g. releasable entities include points and signals* and *point entities only include points*. The choice of entity types and the hierarchy is critical while authoring signalling principles.

The actual entities model real world entities or real world concepts as in the case of train routes. An entity consists of n-ary relations to other entities

[4] Pronounced ‘is a’.

and a current state, *i.e. each track segment is related to other track segments; typically each track segment is connected to a predecessor and successor in the case of a straight line; but terminal segments and points do not follow this scheme. Sect. 6.1.1 discusses this in detail.* The state of an entity, in the case of a signal indicates which aspects are currently presented to trains, in the case of points whether they are in the normal, reverse or neither positions.

6.1.1 Railway Yard Modelling

The concept of an entity type hierarchy can be modelled in many different ways. Regardless of the methodology, the following two points need to be considered; firstly, the actual type hierarchy needs to be defined including the actual relations and possible states of the various entity types. Secondly, the actual entities need to be defined formally and given types.

There are many different formal languages which these concepts can be encoded into. Propositional predicate calculus is a natural choice as it is built around the concept of relations on arbitrary objects. These definitions can then be used in FOL predicate calculus to query information about the railway yard.

Given the track plans and the signalling information, a model for a railway yard can been built according to a chosen entity type hierarchy. The following sections use GNU Prolog syntax for defining the railway yard model as it seems a natural choice for this task given that it is based on predicate calculus.

Prolog does not know anything about the entities; they are treated as literals. Thus a relationship `rel` between two entities `e1` and `e2` can be represented by:

```
rel(e1,e2).
```

Where `rel` has an arity of two, and written as `rel/2`; '.' indicates the end of the term. To specify that an entity `e` is of a given type `type` a unary arity predicate can be used,

```
type(e).
```

unary predicates can be thought of as adding an attribute to an entity. In this case the attribute is the fact that entity `e` is of type `type`. Unary predicates are used extensively to define the finite sets of entities as described above.

6.1.1.1 Track Segments

The basic structure of the railway yard is the track segment, and each track segment has an identifier, e.g. `ts1`, `ts2`, . . . , `tsn`. This is defined by:

```
tracksegment(ts1).
tracksegment(ts2).
.
.
.
tracksegment(tsn).
```

To state that all track segments are also of type `yardobject` the following line defines this relationship

```
yardobject(X) :- tracksegment(X).
```

Track segments are connected to each other, usually a track segment will at most connect to two other track segments, but points can connect to three other track segments and terminus segments connect to one other segment. Using a predicate `connects_to/2`[5], which track segments are connected to other track segments can be defined.

```
connects_to(ts1,ts2).
connects_to(ts2,ts3).
connects_to(ts2,ts4).
```

The above description would indicate that `ts2` is a point. When two track segments are connected there should be no direction for this connection such that trains can go both directions over these track segments. For this there are two possibilities: the first, is to duplicate the data and swap over the operands to the predicates; and the second, is to introduce another predicate `connected/2`:

```
connected(X,Y) :- connects_to(X,Y) | connects_to(Y,X) .
```

The vertical bar indicates a disjunction.

The example railway yard from Sect. 2.2 would have the following track segments defined

```
tracksegment(ts0a).
tracksegment(ts0b).
tracksegment(ts1a).
tracksegment(ts1b).
tracksegment(ts2a).
tracksegment(ts2b).
```

[5] The '/2' indicates that the predicate takes two arguments, namely, the two track segments which are connected.

```
tracksegment(ts3a).
tracksegment(ts3b).
tracksegment(ts4a).
tracksegment(ts4b).
tracksegment(ts5a).
tracksegment(ts5b).
tracksegment(ts6a).
tracksegment(ts6b).
```

with the following `connects_to/2` relations defining the topology of the rail yard.

```
connects_to(ts0a,ts1a).
connects_to(ts1a,ts2a).
connects_to(ts2a,ts3a).
connects_to(ts3a,ts4a).
connects_to(ts4a,ts5a).
connects_to(ts5a,ts6a).

connects_to(ts0b,ts1b).
connects_to(ts1b,ts2b).
connects_to(ts2b,ts3b).
connects_to(ts3b,ts4b).
connects_to(ts4b,ts5b).
connects_to(ts5b,ts6b).

connects_to(ts2a,ts2b).
connects_to(ts3a,ts3b).
```

6.1.1.2 Releasable

The `releasable` entity type is a super set of all entity types that can released to the maintainer for local control. This includes rail yard objects such as points and signals.

Extending the definition of `yardobject` to be the super type of all releasable entities and track segments the following definition can be given as

```
yardobject(X) :- tracksegment(X) | releasable(X).
```

where `releasable` is defined progressively in subsequent sections. This definition has the semantics *a yard object is a track segment or is a releasable entity.*

6.1.1.3 Signals

There are different types of signal: main signal and distant signal but each has an identifier so we can define them by:

```
mainsignal(ms1).
mainsignal(ms2).

distantsignal(ds1,ms1).
distantsignal(ds2,ms2).

signal(X) :- mainsignal(X) | distantsignal(X,_).
```

From these definitions it can be seen that `ms1`, `ms2`, `ds1` and `ds2` are all signals, where `ms1` and `ms2` are main signals, and `ds1` and `ds2` are distant signals. Where `ds1` is the distant signal of `ms1` and likewise for `ds2`.

To show that signals are also of type `releasable` and indirectly of type `yardobject` the definition of `releasable` is

```
releasable(X) :- signal(X).
```

A signal can only be seen from one direction and can only occur between two track segments. To define this, two more predicates are introduced, `infrontof/2` and `inrearof/2`. A signal can be seen from trains approaching from the track segment indicated by `infrontof` but not by trains approaching from track segment indicated by `inrearof`.

```
infrontof(ts1,ms1).
inrearof(ts2,ms1).
```

Defines that `ms1` is between `ts1` and `ts2` and can be seen from trains approaching on `ts1`.

The example railway yard from Sect. 2.2 does not have any distant signals, thus has the following entity type definitions

```
mainsignal(ms1).
mainsignal(ms2).
mainsignal(ms3).
mainsignal(ms4).
mainsignal(ms5).
mainsignal(ms6).
```

with the `infrontof/2` relations

```
infrontof(ts0a,ms1).
infrontof(ts5a,ms2).
infrontof(ts5a,ms3).
infrontof(ts1b,ms4).
infrontof(ts5b,ms5).
infrontof(ts5b,ms6).
```

and the `inrearof/2` relations

```
inrearof(ts1a,ms1).
inrearof(ts4a,ms2).
inrearof(ts6a,ms3).
inrearof(ts0b,ms4).
inrearof(ts4b,ms5).
inrearof(ts6b,ms6).
```

NOTE: for pragmatic reasons of the GNU Prolog engine, the predicates are grouped together.

6.1.1.4 Points

Points are track segments and also have their own identifier so a simple predicate `point/2` is introduced that relates a point identifier to a track segment identifier. The connected track segments do not need to be specified here because they have already been defined by the `connects_to` predicate.

```
point(pt1,ts2).
```

Points are also releasable yard objects, to reflect this the definition of `releasable` is updated to

```
releasable(X) :- signal(X) | point(X).
```

which has the semantics *a releasable entity is a signal or a point.*

The model needs to know which is the normal and which is the reverse branch of the point. This is done by the introduction of two more predicates `normal_branch/2` and `reverse_branch/2`.

```
normal_branch(ts3,pt1).
reverse_branch(ts4,pt1).
```

The example railway yard from Sect. 2.2 has four points,

```
point(pt1,ts2a).
point(pt2,ts3a).
point(pt3,ts2b).
point(pt4,ts3b).
```

with the normal branches defined as

```
normal_branch(ts1a,pt1).
normal_branch(ts4a,pt2).
normal_branch(ts3b,pt3).
normal_branch(ts2b,pt4).
```

and the reverse branches defined as

```
reverse_branch(ts2b,pt1).
reverse_branch(ts3b,pt2).
reverse_branch(ts2a,pt3).
reverse_branch(ts3a,pt4).
```

6.1.1.5 Routes

Routes consist of a list of connected track segments through a railway yard that start and stop at a main signal. These main signals should be 'in front of' the direction of the route, i.e. such that the train can see them. There should be no other main signal on the route other than the start and stop signals, introducing a predicate `route/1` that defines routes and a predicate `part_of/2` that relates a track segment to a route.

```
route(rt1).
route(rt2).

part_of(ts2,rt1).
part_of(ts3,rt1).

part_of(ts2,rt2).
part_of(ts4,rt2).
```

Routes are also releasable yard objects, to reflect this the definition of `releasable` is updated to

```
releasable(X) :- signal(X) | point(X) | route(X).
```

which has the semantics *a releasable entity is a signal, a point or a route.*

Each route has a direction that can not always be derived from the location of the main signals. This can be specified by choosing a track segment directly before the start of the route:

```
before(ts1,rt1).
before(ts1,rt2).
```

This scheme is limited in the sense that it does not consider routes that start on a terminal track segment such as in a rail depot.

The example railway yard from Sect. 2.2 has four routes as defined in *Table 2.1* and depicted in *Fig. 6.2.* These can be formally represented in the model as

```
route(A).
route(B).
route(C).
route(D).
```

where `A`, `B`, `C` and `D` are identifiers not variables, capital letters in Prolog denote variables; they should be encapsulated within single quotes but have not been for readability. The `part_of/2` relations are as follows

```
part_of(ts1a,A).
part_of(ts2a,A).
part_of(ts3a,A).
part_of(ts4a,A).
part_of(ts5a,A).

part_of(ts1a,B).
part_of(ts2a,B).
part_of(ts3a,B).
part_of(ts3b,B).
part_of(ts4b,B).
part_of(ts5b,B).

part_of(ts4a,C).
part_of(ts3a,C).
part_of(ts2a,C).
part_of(ts2b,C).
part_of(ts1b,C).

part_of(ts4b,D).
part_of(ts3b,D).
part_of(ts2b,D).
part_of(ts1b,D).
```

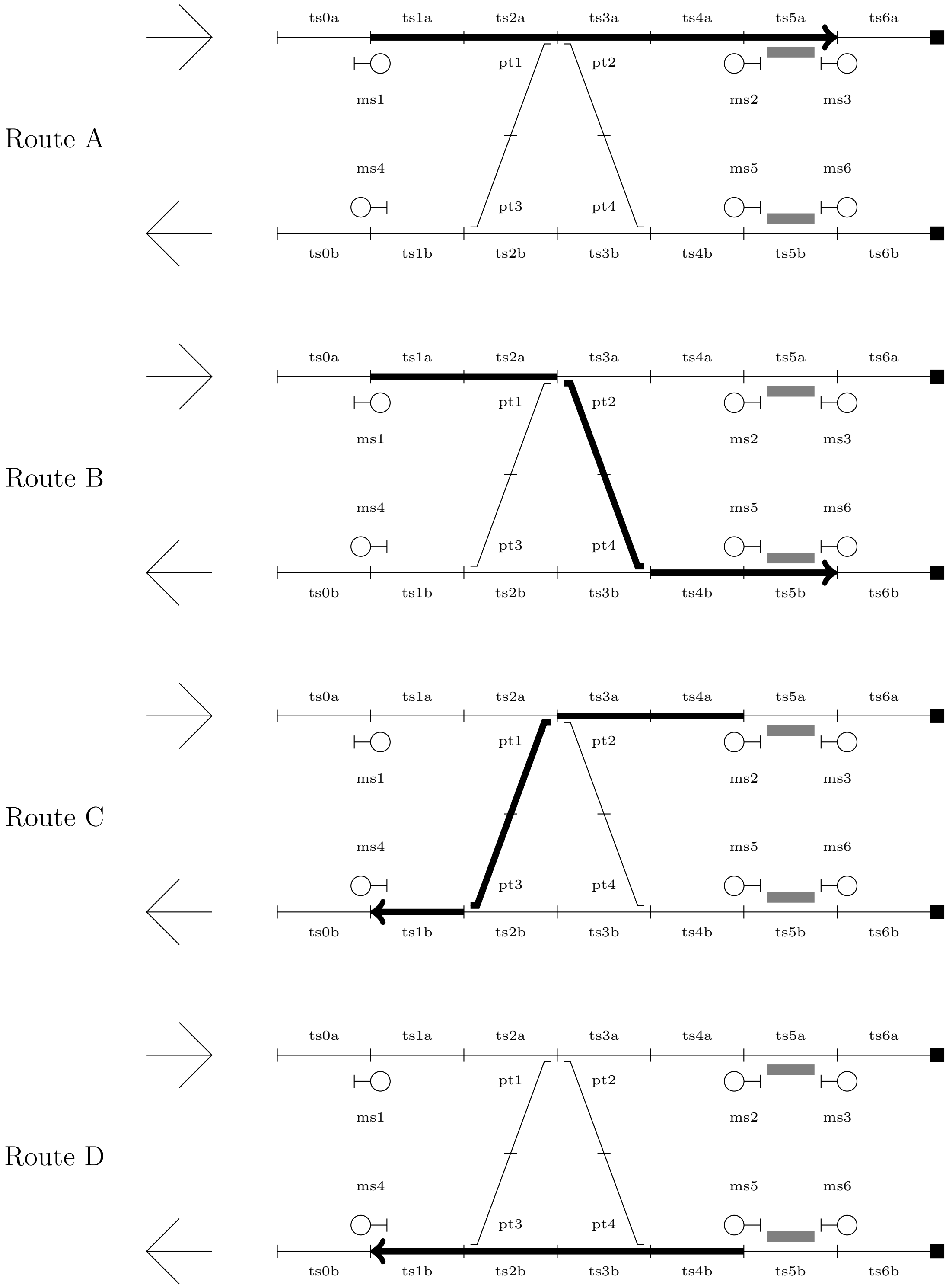

Route A
ts0a ts1a ts2a ts3a ts4a ts5a ts6a
pt1 pt2
ms1 ms2 ms3
ms4 ms5 ms6
pt3 pt4
ts0b ts1b ts2b ts3b ts4b ts5b ts6b
Route B
ts0a ts1a ts2a ts3a ts4a ts5a ts6a
pt1 pt2
ms1 ms2 ms3
ms4 ms5 ms6
pt3 pt4
ts0b ts1b ts2b ts3b ts4b ts5b ts6b
Route C
ts0a ts1a ts2a ts3a ts4a ts5a ts6a
pt1 pt2
ms1 ms2 ms3
ms4 ms5 ms6
pt3 pt4
ts0b ts1b ts2b ts3b ts4b ts5b ts6b
Route D
ts0a ts1a ts2a ts3a ts4a ts5a ts6a
pt1 pt2
ms1 ms2 ms3
ms4 ms5 ms6
pt3 pt4
ts0b ts1b ts2b ts3b ts4b ts5b ts6b

Figure 6.2: Graphical Representation of Routes from *Table 2.1*.

Routes can also have *overlap* and *flank* protection; these are specified by two more predicates: `overlap/2` that indicates that a given track segment is an overlap of a given route, and `flank/3` that indicates that a given point provides flank protection for a given route.

Overlap Protection It is possible that a train does not stop in time when shown a stop aspect, thus for safety reasons a number of track segments are allocated immediately following a route. The actual number of track segments depends on the maximal stopping distance of train using the route. Assuming that the required overlap protection is one track segment, the following defines the overlap using `overlap/2`

```
overlap(ts6a,A).
overlap(ts6b,B).
overlap(ts0b,C).
overlap(ts0b,D).
```

Flank Protection A *flank point* is a point which, if traversed by an overrunning train in the facing direction, could direct that train towards a route and overlap that has been allocated for an authorised train movement. Flank protection is defined as the setting of flank points to the position whereby an overrunning train will be diverted away from a route and overlap that has been allocated for an authorised train movement, i.e. *a set route.* The points providing flank protection for the route can be related to the route using `flank/3` which takes as operands a point identifier, a configuration and a route identifier. For route A, in the running example

```
flank(pt3,normal,A).
flank(pt4,normal,A).
```

6.1.1.6 Auxiliary Predicates

While writing signalling principles, it is often convenient to create auxiliary predicates in the model which simplify the process of writing the conditions. *i.e. determine whether a point is part of a route.*

equality It is often useful to test equality between entities while writing the signalling principles, this can be done by wrapping the prolog equality operator into a predicate,

```
equal(X,Y) :- X == Y.
```

negation While writing auxiliary predicates, negation is a useful feature and can be defined by,

```
not(G) :- G, !,
          fail.
not(_).
```

where `!` is the *cut* operator to stop matching the second case of `not` in the case that `G` is true. The second case accepts anything and is only matched in the case that `G` does not hold. `fail` is a built in predicate that will never hold.

route_main_signal Determining if a signal is the main signal of a route can be done as follows in prolog code,

```
route_main_signal(MSI,RT) :- before(TS,RT),
                             connected(TS,TSS),
                             part_of(TSS,RT),
                             infrontof(TS,MSI),
                             inrearof(TSS,MSI).
```

point_part_of Determining if a point is part of a route can be done as follows in prolog code,

```
point_part_of(PT,RT) :- route(RT), point(PT,TS), part_of(TS,RT).
```

pointnormal Determining if a point should be normal in a route can be done as follows in prolog code,

```
pointnormal(PT,RT) :- point_part_of(PT,RT),
                      normal_branch(TS,PT),
                      part_of(TS,RT).
```

This works by first checking that the point is part of the route then checks that the normal branch of the point is part of the route.

pointreverse Determining if a point should be reverse in a route can be done as follows in prolog code,

```
pointreverse(PT,RT) :- point_part_of(PT,RT),
                       point(PT,_),
                       not(pointnormal(PT,RT)).
```

It works by first checking that the point is part of the route then negates the validity of `pointnormal/2`.

6.1.2 Safety Condition Generation

The reduction of a signalling principle to a safety condition produces many safety conditions. Given a signalling principle φ with quantifiers of the form

$$\forall x_i : \tau_i$$

or

$$\exists x_i : \tau_i$$

where each τ_i is a type defined in the entity hierarchy. All different entity types must be finite as all rail yards only have finitely many components, thus conversion of ψ can proceed as follows:

- Construct the prenex normal form of ψ, called ψ_{prenex}.
- Using structural induction recurse through ψ_{prenex} replacing universal and existential quantifiers for the appropriate conjunctions and disjunctions, respectively.

6.1.2.1 Prenex Normal Form

A prenex normal form of a FOL formula ψ is where all the quantifiers occurring in ψ are moved to the front of the formula so it is of the form

$$(\forall x_1 \ldots x_{n_1})(\exists y_1)(\forall x_{n_1+1} \ldots x_{n_2})(\exists y_2) \ldots (\forall x_{n_r+1} \ldots x_{n_{r+1}})(\exists y_{r+1})\varphi$$

where φ is quantifier free, also known as the matrix.

The reason that the signalling principles are placed into prenex normal form is that separation of the safety conditions becomes easier. Typically a signalling principle will use universal quantification $\forall x : \tau$, each possibility that the quantification can assign to x will yield a different safety condition. The next section shows how to remove quantifiers, where universal quantification is replaced by a finite conjunction, *i.e. each conjunct is a safety condition.* Thus, the matrix becomes a safety condition to prove after being instantiated with entities defined by the quantifiers.

If the signalling principle was not placed into prenex normal form, then, if the author of the conditions does not fully understand the logic, unexpected results could occur, i.e. *instead of generating lots of small safety conditions, they generate one very large safety condition, being a conjunction of the smaller ones.* Although the large safety condition and a conjunction of the small ones have the same semantics, if one of the smaller ones does not hold, then the whole of the large condition does not hold making it harder to determine the reason the signalling principle does not hold.

The first step of the translation is to make sure that no two quantifiers in ψ use the same variable, a simple rename operation can be applied. If this was not the case, then in prenex normal form there might be ambiguity in the matrix with the variable quantified more than once.

The following function can be defined using case distinctions and structural induction, [vD04]:

$$\mathit{prenex} : \mathit{FOLFormula} \rightarrow \mathit{PrenexFOLFormula}$$

Let x' be a fresh variable not occurring in either φ and ψ.

$$\begin{array}{lcl}
\mathit{prenex}((\boxast x : \tau\ \varphi) \Box \psi) & := & \boxast x' : \tau\ \mathit{prenex}((\varphi[x := x']) \Box \psi) \\
\mathit{prenex}(\varphi \Box (\boxast x : \tau\ \psi)) & := & \boxast x' : \tau\ \mathit{prenex}(\varphi \Box (\psi[x := x'])) \\
\mathit{prenex}(\neg \forall x : \tau\ \varphi) & := & \exists x : \tau\ \mathit{prenex}(\neg \varphi) \\
\mathit{prenex}(\neg \exists x : \tau\ \varphi) & := & \forall x : \tau\ \mathit{prenex}(\neg \varphi)
\end{array}$$

where $\boxast \in \{\forall, \exists\}$ and $\Box \in \{\wedge, \vee\}$. The first two cases rename the variable x such that there are no clashes with other free variables.

The trivial cases (*inc.* base cases) that recurse the formula's structure have been omitted for simplicity. The above translation is only valid in classical logic, conjunction and negation cause problems in intuitionistic logic. *prenex* can be extended to implications $a \rightarrow b$ by translating to $\neg a \vee b$ and applying the rules above. The result is:

$$\begin{array}{lcl}
\mathit{prenex}((\forall x : \tau\ \varphi) \rightarrow \psi) & := & \exists x : \tau\ \mathit{prenex}(\varphi \rightarrow \psi[x := x']) \\
\mathit{prenex}((\exists x : \tau\ \varphi) \rightarrow \psi) & := & \forall x : \tau\ \mathit{prenex}(\varphi \rightarrow \psi[x := x']) \\
\mathit{prenex}(\varphi \rightarrow (\boxast x : \tau\ \psi)) & := & \boxast x : \tau\ \mathit{prenex}(\varphi[x := x'] \rightarrow \psi)
\end{array}$$

Through a similar reasoning it is possible to extend *prenex* to equivalences although care must be taken as quantifiers are duplicated so a renaming strategy is required.

6.1.2.2 Quantifier Removal

The basic strategy for removal of quantifiers was introduced in Sect. 4.6. Suppose a signalling principle φ_A in prenex normal form

$$\forall pt : \mathit{Point}\ \varphi'_A(pt)$$

which has the semantics, *for all points* φ'_A *holds*. There are only a finitely number of many points in any given rail yard, let pt_1, pt_2, …, pt_n define the set *Point* i.e. $pt_i \in \mathit{Point}$. So each pt_i substituted into φ'_A produces a new safety condition.

Taking a conjunction of all such safety conditions has the same semantics as φ_A, i.e.

$$\varphi_A \Leftrightarrow \varphi'_A(pt_1) \wedge \varphi'_A(pt_2) \wedge \cdots \wedge \varphi'_A(pt_n)$$

Following a similar argument for existential quantification, suppose a signalling principle φ_E in prenex normal form

$$\exists pt : Point\ \varphi'_E(pt)$$

which has the semantics, *there exists at least one point where φ'_E holds.* This has an identical meaning to a list of disjunctions, i.e. let p, q and r be atomic propositions.

$$p \vee q \vee r$$

holds exactly when one or more p, q or r holds. Thus

$$\varphi_E \Leftrightarrow \varphi'_E(pt_1) \vee \varphi'_E(pt_2) \vee \cdots \vee \varphi'_E(pt_n)$$

6.1.2.3 Predicate Removal

The final phase of the translation from signalling principles into safety conditions requires that the predicates used to reason about the rail yard and its entities are resolved into literals.

For simplicity here, *tt* and *ff* are assumed to be literals constrained to true and false values respectively.

Predicates that exist within the model, such as

```
connected(ts1,ts2).
```

can be looked up in the model and replaced with *tt* and *ff* accordingly.

Example Let ψ be a safety condition that must hold for all connected track segments, formulated into a signalling principle as

$$\forall t_1 : TrackSegment\ \forall t_2 : TrackSegment\ (\texttt{connected\_to}(t_1, t_2) \rightarrow \psi(t_1, t_2))$$

which has the semantics if t_1 is connected to t_2, then ψ must hold where t_1 and t_2 are track segment identifiers. As universal quantification over an entity type τ is replaced by a finite conjunction with $|\tau|$ (the number of entities with type τ in the rail yard) conjuncts, this technique produces a large number of conjuncts[6], in the example above $|TrackSegment|^2$ conjuncts are produced. Resolving `connected(ts1,ts2)` into a constant Boolean value using the topology model in each of the $|TrackSegment|^2$ conjuncts and applying standard Boolean rules to remove constants from the formula leaves only the conjuncts where t_1 is connected to t_2; conjuncts where t_1 is not connected to t_2 are removed.

[6] Safety conditions in this case.

Literal Predicates A second class of predicates that exists within the context of signalling principles relates to the encoding of literals in the ladder. Typically literals within the ladder follow a naming convention such as given a point identifier, appending various strings to this identifier will result in properties about the point. This notion can be extended to all entities on the railway yard. The actual strings that can be appended change from railway yard to railway yard and should be specified in the naming convention for a given railway yard.

Example, let $Points := \{pt_1, pt_2, \ldots, pt_n\}$. If $pt \in Points$, then $pt.Normal$, $pt.Reverse$ and $pt.Locked$ are valid literals. **Note:** This is a fictional naming convention not based on any convention in particular.

Introducing a new function $custom : String^2 \rightarrow String$ which concatenates two strings together. Thus assuming a canonical mapping f from elements of $Points$ to textual representations, $f_{Points} : Points \rightarrow String$, f is the equivalent of `toString` in a conventional programming language. Continuing the above example,

$$\forall pt \in Points \Rightarrow custom(f_{Points}(pt), \text{".Normal"}) \quad \text{is a valid literal.}$$

The *custom* function has a simple reduction strategy identical to the 'concatenation of two lists'. The only constraint is that the variables in the operands must be instantiated, otherwise they yield ambiguous results.

6.1.2.4 Example Reduction

The example introduced in Sect. 1.3 demonstrates the reduction of a signalling principle into a safety condition. Repeated here for simplicity.

Suppose a signalling principle such as "*points in a rail yard should not be set to the normal and reverse positions simultaneously*", which when formulated into FOL becomes

$$\forall pt \in Points : \neg(normal(pt) \land reverse(pt))$$

Also, suppose that the rail yard to be ratified against this signalling principle has two points, namely *pta* and *ptb*. Note that the formula is already in prenex normal form so this step is skipped. Removal of quantifiers would yield

$$\neg(normal(pta) \land reverse(pta)) \land \neg(normal(ptb) \land reverse(ptb))$$

and, removal of predicates would yield

$$\neg(pta.Normal \land pta.Reverse) \land \neg(ptb.Normal \land ptb.Reverse)$$

As there are no constant Boolean values introduced while reducing the predicates, the removal of the constants is skipped. Finally, the conjuncts are extracted, these conjuncts are the safety conditions to be verified.

$$\neg(pta.Normal \wedge pta.Reverse)$$

and

$$\neg(ptb.Normal \wedge ptb.Reverse)$$

This completes the example.

6.2 Safety Conditions

The difference between signalling principles and safety conditions is that, safety conditions can not use quantification or predicates. Mathematically, this is the difference between first order logic and propositional logic.

Safety conditions are propositional formulæ which range over literals in the ladder. I.e. *safety conditions have no quantification and no predicates, only literals from the ladder are used as atomic propositions.*

6.3 Proving

Only safety conditions are used while attempting to prove safety properties as they relate directly to the safety conditions used in the proof formulæ. The following sections describe induction as used for this project. This section describes the proof formulæ chosen to be entered into the SAT-Solver and the reasoning behind the chosen formulæ.

6.3.1 Proof of Safety Properties using Induction

As already discussed in Sect. 3.2, ladder logic uses a sequential execution strategy where an undefined n number of cycles are consecutively executed performing *input* and *output* operations between these cycles. The only distinguishing property of these cycles is the state of the memory between executing cycles. This is known as the state of the program, and in the case of the ladder it consists of truth values associated to the literals that consist within the ladder.

To prove that a given safety property ψ holds for a given $\varphi_{\mathcal{L}}$, it suffices to verify that in all states (*excluding the initial state*) of the program ψ holds.

The goal of the verification is to prove that a safety condition ψ holds after n execution cycles of the ladder. Formally,

$$\forall n \in \mathbb{N} \setminus \{0\} \quad \psi(n)$$

There are different methods of performing this verification; a simple method is to enumerate all possible states of the program and check that for each state ψ holds; ignoring reachability of the counter example. A different technique is to use induction.

To show that a safety condition holds in each n^{th} cycle of the ladder the induction principle can be applied. The base case requires that the safety condition ψ holds after the first cycle of the ladder. The reason that the condition is only proven to hold after the first execution of the ladder is that the initial state of the ladder could invalidate the condition, but after the ladder has executed one cycle the system is deemed to be in a safe state. Thus the base case formula is

$$\psi_I \wedge \varphi_{\mathcal{L}} \rightarrow \psi'$$

where ψ_I is the initial configuration of the variables and ψ' is a renamed version of ψ using the substitution constructed while translating the ladder in Sect. 5.3.

The inductive step of the proof states that **if** ψ holds in the n^{th} cycle, **then** it holds in the $n^{th} + 1$ cycle.

$$\psi \wedge \varphi_{\mathcal{L}} \rightarrow \psi'$$

where ψ' is as above.

6.3.1.1 Limitations

The inductive proof system described is limited in the sense that if a counter example is found, then finding a reachability trace is not trivial in the sense that a counter example might be in an unreachable state. Counter example traces are of interest to Invensys. Although, in the case no counter example is found, then it is proven the safety conditions hold after the n^{th} iteration for an arbitrary n.

Similarly another major weakness is that if a counter example is found, would this be a *real* counter example? i.e. *whether from the initial state it is possible to end up in a state where the safety condition does not hold.* This is mitigated by appending invariance to the formulæ. These invariances restrict the possible states considered; invariance is discussed at length in the next section.

6.3.2 Invariance & Reachable States

After each execution cycle of the ladder the system can be in exactly one unique state; each state is characterised by the configuration of the propositional variables within the ladder. Some of these configurations are invalid in the sense that the ladder will never reach these states under any circumstances, i.e. *regardless of the number of cycles of the ladder and the inputs given to the ladder throughout these cycles.* There are two reasons for this restriction, one is the hardware forbids certain input combinations and the second reason is that the ladder logic forbids certain states. For example, let a and b be atomic propositions in the ladder, it could be the case that the ladder logic enforces $a \leftrightarrow \neg b$.

A propositional formula ψ built from the atomic propositions in the ladder that holds for all states that are reachable (*valid*) is called an invariance of the ladder. See *Fig. 6.3* for an example. In practice, many invariants hold not only in the reachable states but in some unreachable states. For example, the invariant $a \leftrightarrow \neg b$ would hold in all reachable states and many unreachable states in all but the most trivial ladders.

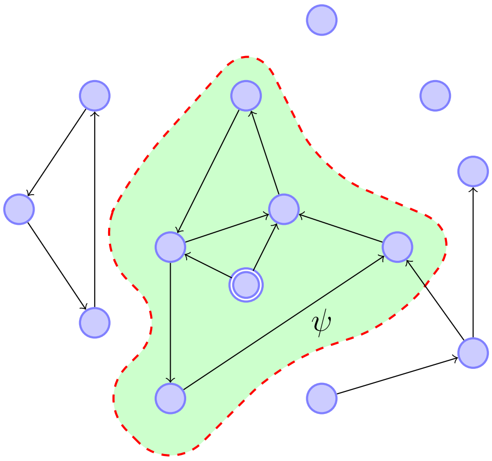


Figure 6.3: Each circle indicates a state the of the system, a state comprises of the configuration of the variables. The double lined state is the initial state and arrows define the transitions between the states. The shaded area indicates states where a formula ψ holds.

Construction of an invariance ψ of the ladder such that it holds exactly in the reachable states is a complex problem. There are two methods that are orthogonal, the first constructs an invariance based on the actual inputs to the ladder enforcing the operational semantics of the hardware, thus removing states that are assumed by the system's designers to be invalid even if the logic permits such a state. This type of invariance is not provable, but follows from the system design, see Sect. 6.3.2.1. The second method produces provable invariances of the ladder, and can be proven to be invariances using the same method as used for proving safety conditions, i.e. *a logical consequence of executing the ladder*, see Sect. 6.3.2.2. Hence, safety conditions are also safety invariants.

Given a safety condition ψ_S and an invariance ψ_{Inv} that defines reachable states of a given ladder $\varphi_{\mathcal{L}}$, it is required that if ψ_{Inv} holds in a given state, then so does the safety condition ψ_S. States in which ψ_{Inv} does not hold are not of interest as the system can never reach them so it does not matter if the safety condition ψ_S fails to hold in these states.

In practice it is hard to construct a strong[7] invariance ψ_{Inv}, as shown in *Fig. 6.3*, that only accepts reachable states. It is easier to construct a slightly weaker version which accepts all reachable states and a few unreachable states. Also to prove that a given invariance accepts exactly the reachable states is not trivial for non-trivial ladders, ideally all states could be enumerated as in *Fig. 6.3* with the transitions between these states defined. When such a state diagram has been built, it is a trivial matter to extract the reachable states by starting with the initial state and following all transitions. Typically a ladder with 600 atomic propositions would have $2^{600} \approx 10^{180}$ states making a careful analysis of all the states and the transitions between them not feasible. In practice, ladders can have thousands of atomic propositions with the number only set to increase in the future; currently Invensys are producing ladders with > 3000 rungs. I.e *a ladder with 6000 atomic propositions can have* $\approx 10^{1806}$ *states.*

If a counter example is found, a question can be raised *is this a real counter example?* The reason this question is raised directly relates to whether the invariance used is strong enough. Suppose while verifying a safety condition a counter example is found, is this counter example in a reachable state? Is a stronger invariance required?

6.3.2.1 Physical Invariances

Physical invariances are states that can not occur because the real world hardware does not allow them. Suppose a switch with three physical contacts representing positions `A`, `B` and `C`, these contacts are closed by the switch so it should not be possible for a switch to be in more than one position simultaneously[8], although it could be possible for neither contact to be closed while the switch is moving between contacts. *Fig. 6.4* shows a three way

[7] An invariance that accepts exactly the reachable states.

[8] Assuming that the hardware is working correctly with no short circuits.

switch, the switch is between positions A and B. The switch is represented within the ladder by three atomic propositional variables, one for each physical contact inside the switch. When moved left the contacts for A are closed and an atomic proposition in the ladder representing this action goes high, then if moved right the atomic proposition goes low. When turned further to the right, the switch is in the B position and an atomic proposition in the ladder representing this action goes high. Likewise for position C.

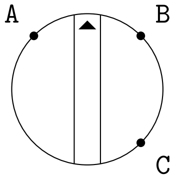


Figure 6.4: 3-Way Switch

These invariances require a careful analysis of a railway yard and signalling principles to construct.

It has been suggested that a correctly implemented ladder logic program should not require physical invariance to be specified during the proof as the program should enforce these constrains internally, i.e. *a switch should be in at most one position.* If there is a conflict, then the safest available option should be chosen. This design approach makes the system resilient against short circuits. Although enough short circuits will always result in ambiguous behaviour of the interlocking, especially if they are maliciously instigated.

6.3.2.2 Mathematical Invariances

A mathematical invariance is a formula ψ which is an invariant of the ladder, i.e. *holds in all the reachable states, excluding the initial state.*

Let ψ_{Inv} be a formula which rejects unreachable states for a given ladder $\varphi_{\mathcal{L}}$. The formulæ defined in Sect. 6.3.1 can be refined to filter out unreachable states. The base case becomes

$$\psi_I \wedge \varphi_{\mathcal{L}} \wedge \psi_{Inv} \rightarrow \psi'$$

where ψ_I is the initial configuration of the variables and ψ' is a renamed version of ψ using the substitution constructed while translating the ladder in Sect. 5.3.

The base case does not strictly require the reachable condition as it is trivially reachable by definition. It starts with the initial configuration of the variables as the ladder would in reality. The reachable condition is of great importance to the inductive step.

The inductive step of the proof states that **if** the safety condition ψ holds in the n^{th} cycle and the cycle is reachable, **then** it holds in the $n^{th}+1$ cycle.

$$\psi \wedge \varphi_{\mathcal{L}} \wedge \psi_{Inv} \rightarrow \psi'$$

where ψ' is as above.

Invariance formulæ can be built by hand with an intimate knowledge of the ladder and the signalling principles for simple systems. Hand built invariance for large ladders is also useful, but can not be expected to be comprehensive in terms of defining the reachable states.

An approach suggested by Dr A. Setzer to aid in automating the generation of invariance is to compute all satisfying assignments of the ladder and look for pairs (a, b) of atomic propositions such that

- $a \leftrightarrow b$ or
- $a \leftrightarrow \neg b$

is an invariance of the ladder. When these pairs are identified, they are fed back into the proof formulæ as an invariance (*see below*) and starts again, and is continued until no more pairs can be identified which fulfil the above conditions. This is obviously a terminating procedure as there are only a finite number of variables in a given ladder but the number of cycles required before termination is not clear. Although this will pick all dependencies between pairs of variables, the approach does not directly scale to finding an invariance with an arbitrary number ≥ 3 of atomic propositions built from arbitrary operations.

6.3.3 Proof Formulæ

While performing the verification, it is necessary to show that the formulæ introduced in Sect. 6.3.2.2 always hold. This is done by negating the two formula's, i.e.

$$\neg(\psi_I \wedge \varphi_{\mathcal{L}} \wedge \psi_{Inv} \rightarrow \psi')$$

and

$$\neg(\psi \wedge \varphi_{\mathcal{L}} \wedge \psi_{Inv} \rightarrow \psi')$$

Thus, in the case both of these formulæ are unsatisfiable, the proof is complete that safety condition ψ holds in all reachable states except the initial. If it is possible to satisfy one of these formula's, then a counter example has been found.

When these formulæ are entered into a SAT-Solver, the solver will search for a satisfying assignment ⇒ counter example.

Chapter 7

Results Obtained

There were many different safety conditions that were made available, some of these were given by *Invensys* others were produced during the project for testing purposes.

This chapter discusses the results of the experimental verification of signalling principle, for legal reasons detailed analysis of counter examples, if they occurred, have been omitted, along with the proof formulæ because the information could be used to detrimental effects. The signalling principle to be verified is given in each case, but without the topology model it is not possible to construct the actual proof formulæ.

7.1 Software Architecture

The software used to produce safety conditions from signalling principles differs from the software used to verify the ladder logic. A top level architecture is shown in *Fig. 7.1.* The two boxes indicate the two parts of the program, *general formula* takes a signalling principle and produces safety conditions (`*.cond` files). *Rail Verifier* reads safety conditions and produces clause sets, which are entered automatically into `OKSolver`. The output of OKSolver is processed to produce counter example documentation, if a counter example was found. The user guide in Appendix A has a comprehensive description of both parts of the software.

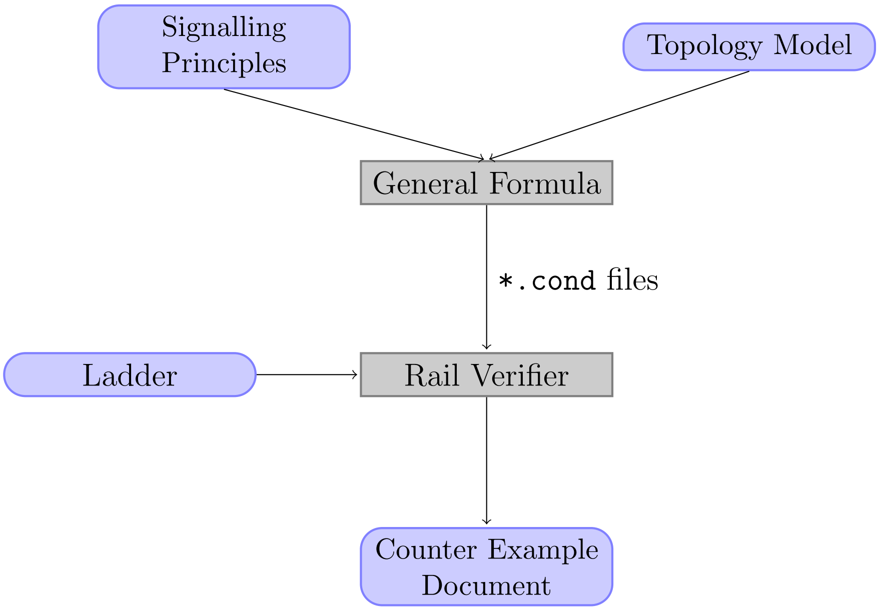


Figure 7.1: Top Level Dataflow

7.2 Variables of Configuration

All benchmarks were carried out using a Dell® Precision WorkStation™ T3400 running Ubuntu® 7.10 Gutsy 64-bit. The machine is configured with two 1GB DDR2 memory modules running at 800MHz. There is one memory module for each CPU core. The CPU is a Intel® Core 2 Duo™ E6850, running at 3GHz.

The proof engine used is `OKSolver`, using an optimised[1] 64-bit build.

The project has been concerned with a specific ladder logic program with 331 rungs and 599 variables. The actual station the ladder logic is for can not be named for legal reasons but has a similar design to the example station shown in the introduction, Sect. 2.2. The names of railway yard entities differ and the topology is more complex.

7.3 Experiments

The list below shows some of the signalling principles that have been experimented with. The signalling principles are all written using first order logic

[1] Built using `GCC` with the `-O3` compiler option.

with general predicates, and the generated safety conditions are written in propositional logic.

Experiments were commenced as follows,

1. Create a root directory, and 6 subdirectories; one for each signalling principle. In practice all the signalling principles can be placed in one directory but for benchmarking purposes there is only one in each directory. These directories are numbered $1 \dots 6$.

2. Enter the signalling principles into text files and save into the subdirectories. These text files have the extension `gen`, although this does not matter.

3. Run the `gen` files through the software to produce safety conditions using the topological model of the station. The resulting files have the extension `cond`, and contain many safety conditions. Each `gen` file produces exactly one `cond` file.

4. Using the software produced during the project, generate clause sets which encode the proof formulæ introduced previously. Each safety condition produces two clause sets required for the induction hypothesis to be valid.

5. Enter the clause sets into a proof engine, saving the output into files in the case a counterexample is found. These files are then processed to help with building documentation.

6. When counterexamples are discovered, generate the correct documentation and concatenate all counterexample documentation into a single file.

7. Save the output from `/usr/bin/time` program for latter analysis. The items 4 to 6 are an atomic command and passed into `/usr/bin/time` for benchmarking purposes. The `time` program provides a detailed analysis of what the process spent its time doing, along with page faults and other information. `bash` comes with a built in `time` command but this caused segmentation faults when used with developed proof system, so should not be used.

There are 6 signalling principle selected for experimentation, which will be discussed below; these are of varying complexity to demonstrate the flexibility of the system.

Variable Name	Type
cpt	$Combinedpoint$
pt	$Point$
rt	$Route$
s	$Signal$
ts	$TrackSegment$

Table 7.1: Type Shorthands

To aid in condensing the signalling principles below, the types of variables are omitted but instead specific letters are used to indicate elements of a given type. See *Table 7.1* for a list of these shorthands.

In practice, there are two types of point identifiers *Combined* and *Normal*, because points appear in pairs. Sometimes each point in a pair has its own identifier and other times the combined pair has an identifier and is treated as an atomic railway yard entity. There are also cases where hybrids of the two methods are applied, i.e. *certain literals only refer to a single point and others refer to both points in the pair.* The *combined* points were not introduced in Sect. 6.1.1 for simplicity.

Signalling principles and safety conditions often make use of logical implications, '→', following the usual standard, implications are right associative. Thus,

$$a \rightarrow (b \rightarrow (c \rightarrow d))$$

can be written as,

$$\begin{aligned} a \quad & \rightarrow b \\ & \quad \rightarrow c \\ & \quad\quad \rightarrow d \end{aligned}$$

7.3.1 Red Aspect with Occupied Trailing Segment

A signal faces in a given direction, such that only trains approaching from one side should be able to see its aspect. Thus, if a train passes a given signal, the next train to approach the signal should see a red aspect unless the first train has travelled a sufficient distance down the line. If the first train is still occupying the track segment directly behind the signal, then the second train should see red aspect on the signal. The formula below encodes this principle

$$\begin{aligned} \forall s, ts \quad inrearof(ts, s) \ & \rightarrow occupied(ts) \\ & \quad \rightarrow redaspect(s) \end{aligned}$$

The predicate $inrearof(ts, s)$ holds exactly when track segment ts is behind the signal, validity of the predicate is resolved from the topology model. The second two predicates are used to build the safety condition, and they can be seen as describing the state of the railway yard entities rather than topological relationships.

When this signalling principle is translated to safety conditions the $inrearof(ts, s)$ predicate is removed using the topological model and replaced by the correct logical value, this constant can then be removed using standard Boolean laws. The resultant safety conditions are of the form

$$occupied(ts) \rightarrow redaspect(s)$$

where these predicates are reduced to literals representing the correct state assuming ts and s satisfy $inrearof(ts, s)$. The final reduction of these predicates into literals is not presented as it does not yield further insight into the process. The above signalling principle was expanded into 4 safety conditions, one for each signal in the rail yard.

The total time taken was 3 minutes and 10 seconds to find that there were no counter examples.

7.3.2 Conflicting Routes can not be Set

If two or more routes require the same track segments, then these routes conflict. This can be formalised into a signalling principle as

$$\begin{aligned} \forall rt, rt', ts \quad rt \neq rt' \quad &\rightarrow (part\_of(ts, rt) \wedge part\_of(ts, rt')) \\ &\rightarrow \neg(routeset(rt) \wedge routeset(rt')) \end{aligned}$$

which has the semantics: if track segment ts is part of two distinct routes rt and rt', then at most one of these routes can be selected at any one time.

The $part\_of(ts, rt)$ predicates are resolved using the topology model, yielding safety conditions of the form

$$\neg(routeset(rt) \wedge routeset(rt'))$$

where routes rt and rt' conflict with each other. The signalling principle was expanded to 5 safety conditions, but in fact there are only two routes which are compatible within the example railway yard.

The total time taken was 4 minutes and 9 seconds, no counter examples were found.

Side Note: Non conflicting routes has been proven valid and has been added to the list of invariants the program uses to help reduce the number of reachable states.

7.3.3 Points not Normal and Reverse

A fairly important signalling principle is that the interlocking does not try to move a point into the normal and reverse positions simultaneously. This could have unpredictable results, more importantly there should never be a scenario according to the signalling rules where such a situation is required. Formally, this condition can be written as

$$\forall cpt \quad \neg(normal(cpt) \wedge reverse(cpt))$$

There are no topological predicates here so removal of the quantifier yields the structure of the produced safety conditions.

$$\neg(normal(cpt) \wedge reverse(cpt))$$

A safety condition of the above form will be produced for each combined point in the railway yard. There are 2 combined points in the railway yard that is being tested, thus, there are 2 produced safety conditions.

The benchmarking shows that these two safety conditions are provable in 1 minute and 20 seconds.

Side Note: The condition that points can not be moved normal and reverse simultaneously has been added to the invariance the program uses after being proved as a valid consequence of the ladder.

7.3.4 Occupied Points Locked

When a train is on the same track segment as a point, then the point should be locked. *i.e. the interlocking should not allow a request to move it.* The signalling principle is below, note that it makes use of both normal and combined point identifiers because the logic only allows combined points to be locked but individual points are related to track segments in the topology model.

$$\begin{aligned} \forall cpt, pt, ts \quad part\_of(pt, cpt) \;&\rightarrow point(pt, ts) \\ &\quad\rightarrow occupied(ts) \\ &\qquad\rightarrow (normal(cpt) \vee reverse(cpt)) \end{aligned}$$

The $part\_of(pt, cpt)$ and $point(pt, ts)$ predicates are solved by the topology model, the other 3 are used to construct the safety conditions. The structure of the safety conditions, after removing the topology predicates becomes:

$$occupied(ts) \rightarrow (normal(cpt) \vee reverse(cpt))$$

There are 4 points in the example railway yard, thus 4 safety condition are generated.

The benchmark took 3 minutes and 20 seconds and found counter examples in all cases, including the base cases. These counter examples are attributed to the fact that the proof system allows for trains to *magically* disappear and/or reappear. Thus, supposing a point was not locked, then suddenly a train appears on the point. As the point is not locked, the interlocking would need to know if it should be locked in the normal or reverse positions. Perhaps this signalling principle was not chosen well for the above reason, although it demonstrates the difficulty with writing good signalling principles.

It is possible to adapt the proof system, see Sect. 8.5.1, such that the proof formula verifies from a state where the safety condition ψ does not hold, then after executing the ladder $\varphi_{\mathcal{L}}$ the safety condition holds. Omitting invariance, the proof formula would become,

$$\neg\psi \wedge \varphi_{\mathcal{L}} \rightarrow \psi'$$

7.3.5 Points Locked when Route Set

Before a route can be selected specific constraints must be satisfied, these constraints are specified in the control tables. One of these constraints is that all the necessary points in the route are moved and locked to the correct positions, normal or reverse; assuming that the point has not been released to the maintainer.

The following formula specifies this as a signalling principle,

$$\begin{array}{lll} \forall rt, cpt & point\_part\_of(cpt, rt) & \rightarrow (routeset(rt) \wedge \neg released(cpt)) \\ & & \rightarrow \left(\begin{array}{cc} & \left(\begin{array}{c} pointnormal(cpt, rt) \rightarrow \\ normal(cpt) \end{array} \right) \\ \wedge & \left(\begin{array}{c} pointreverse(cpt, rt) \rightarrow \\ reverse(cpt) \end{array} \right) \end{array} \right) \end{array}$$

where $point\_part\_of(cpt, rt)$ holds only when cpt is a point in route rt and is resolved using the topology model. $pointnormal(cpt, rt)$ and $pointreverse(cpt, rt)$ hold *iff*, cpt is normal or reverse respectively, in route rt. The other predicates are used to construct the safety conditions.

The last conclusion consisting of a conjunction of two formulæ is used to select whether a point should be normal or reverse in the route. When resolving the *pointnormal* and *pointreverse* predicates in the conjunction, only one of these will remain after applying Boolean laws for removing constants as a point should never be specified to be in the normal and reverse positions within the same route.

The generated safety conditions take the form of,

$$(routeset(rt) \wedge released(cpt)) \rightarrow \Box(cpt)$$

where $\Box$ is either *normal* or *reverse* as appropriate for the point *cpt* in route *rt*. One of these safety conditions is produced for each combined point in each route. *i.e. suppose there are two combined points and four routes, each using both of these points as in* Table 2.1*, then there would be 8 safety conditions produced.* In the example railway yard being verified, there are 8 safety conditions, the total time taken to verify these conditions was 7 minutes and 6 seconds. Counter examples were found in all the inductive steps, although Invensys has said that these counter examples were already known about and do not present a safety threat as the interlocking will correct itself in the subsequent cycle, Sect. 8.5.1. The cycle time is at most 1 second, [Wes06].

7.3.6 Tracks Clear with Green Aspect and Route Set

One of the constraints for a route to be selected is that all track segments that make up the route should be unoccupied, assuming the train has not entered into the route. When the train enters into the route, the main signal guarding the route changes from a green (*proceed*) aspect to a red aspect. Thus, when a route is selected and the guarding signal shows a green aspect, then all track segments within the route should be unoccupied. This can be written as a signalling principle as,

$$\begin{aligned} \forall rt, s \quad route\_main\_signal(s, rt) \ &\rightarrow greenaspect(s) \wedge routeset(rt) \\ &\rightarrow \forall ts \quad \begin{pmatrix} part\_of(ts, rt) \rightarrow \\ \neg occupied(ts) \end{pmatrix} \end{aligned}$$

which is not in prenex normal form, although it has the correct semantics. When converting this into safety conditions the formula is automatically translated into prenex normal form, yielding

$$\begin{aligned} \forall rt, s, ts \quad route\_main\_signal(s, rt) \ &\rightarrow greenaspect(s) \wedge routeset(rt) \\ &\rightarrow \begin{pmatrix} part\_of(ts, rt) \rightarrow \\ \neg occupied(ts) \end{pmatrix} \end{aligned}$$

where $route\_main\_signal(s, rt)$ holds *iff* signal s is the main signal of route rt and $part\_of(ts, rt)$ holds *iff* track segment ts is part of route rt. Both of these predicates are resolved using the topology model. The other predicates are used for producing the safety conditions. The safety conditions are of the form,

$$(greenaspect(s) \wedge routeset(rt)) \rightarrow (\neg occupied(ts))$$

such that the necessary constraints are satisfied. One safety condition is produced for each track segment in each route; this has the effect that if the condition does not hold for one track segment, then this track segment is immediately identifiable. In the example railway yard this condition produced 25 safety conditions. Verification of these 25 conditions took 23 minutes and 14 seconds with no counter examples found.

Chapter 8

Conclusions

8.1 Feasibility

The main focus of the project was to determine whether formal verification of ladder logic with respect to railway signalling principles is feasible. The project has been concerned with a specific ladder logic program with 331 rungs and 599 variables. The actual station the ladder logic is for can not be named for legal reasons but has a similar design to the example station shown in the introduction, Sect. 2.2. The names of railway yard entities differ and the topology is more complex.

The task of formal verification can be split into two main sub tasks: generation of the proof formulæ as described in previous sections and actually entering these fromulæ into a proof engine to yield a result. The current implementation of the generation of the proof formulæ is not optimal. There are many techniques which can be applied that will decrease the time required for the complete verification cycle but this non-optimal program has shown that the task is feasible. After many discussions with researchers in this area, it has been suggested that the time taken for the clause set generation could be in the order of seconds not minutes.

8.2 Software Review

As already indicated the software is not optimal but works. The software is written to run on Linux using a multitude of languages including:

Haskell Used for parsing the ladder logic program and generating the clause sets. This part of the system is open for many optimisations. An optimal solution would be to use a native language such as `C++` along with optimal algorithms.

Python Post processing of the counter examples.

Latex Production of documentation of the counter examples.

Java Used to translate the signalling principles into safety conditions.

Bash Scripting Used for many different purposes, mainly used to incorporate the different parts of the program into one program.

Makefile Coordinates the proving of clause sets and generation of the documentation.

The major component of the software is written in Haskell primarily because Haskell is a high level language that allows for complex functions to be written simply and provides a type checking system that picks up many erroneous statements that would not be discovered in a conventional programming language until the testing phase. A major drawback of Haskell is efficiency compared to a language such as `C++`.

The program uses the *Naïve* translation introduced in Sect. 5.2 to produce the model of the ladder, whereas the *Optimised* translation would be better as the result is a conjunction of equivalences which would save processing later when producing the clause sets[1]. The optimised translation was not considered until the software had been written; it was decided that as the software works it was best not to modify it.

The parser of the ladder logic which is stored in a file format developed by Invensys needs to be extended because it currently was only written to interpret the ladder logic used for the specific interlocking being verified. It was decided that parsing of any arbitrary ladder logic in the file format was not required for this project, but could be required for subsequent work, thus the basic architecture within the software is in place.

8.3 Limitations

8.3.1 *Pre* and *Post* Variable References

The safety conditions can not reference both values before and after execution of the loop, i.e. all output variables are recomputed after each iteration of the ladder. Thus if a safety condition relies upon the previous value of an output

[1] Clause sets are in conjunctive normal form.

variable, then the safety condition can not be represented. It is possible to change the generated proof formulæ such that it allows for pre and post variables to be extracted, but as the majority of safety conditions can be expressed with the current system the change was not applied, although further research of this principle will be required for a comprehensive system capable of fulfilling Invensys's requirements.

One possibility is to adapt the proof formulæ as follows:

$$I \wedge \varphi_{\mathcal{L}_0} \rightarrow \psi_1$$

$$I \wedge \varphi_{\mathcal{L}_0 \wedge \mathcal{L}_1} \rightarrow \psi_2$$

$$\varphi_{\mathcal{L}_n} \wedge \psi_n \wedge \varphi_{\mathcal{L}_{(n+1)}} \wedge \psi_{(n+1)} \rightarrow \psi_{(n+2)}$$

where I is the initial configuration of variables, $\varphi_{\mathcal{L}_i}$ is a propositional model of the ladder representing the i^{th} iteration of the ladder and ψ_i is the safety condition for the i^{th} iteration. All of the above formulæ omit the invariance for readability.

The proof formula as shown in Sect. 6.3 has only one base case, whereas those shown above have two. The reason for this is because the inductive step requires that the safety condition holds in the previous two states.

The inductive step has two iterations of the ladder, thus, it is possible to select pre and post variables for all the variables in the ladder. The current system only has one iteration allowing for only pre and post values of the variables which are computed during the execution of the ladder.

8.3.2 Counter Example Traces

Originally the scope of the project was to not only formally verify the interlocking, but in the cases where counter examples were discovered an explanation of how to reproduce these violations (*a trace*) was required. As the project progressed it became clear that the chosen methodologies did not allow for this. The idea was to encode into the clause set extra information which would allow for this trace to be automatically constructed, but time was a limiting factor, thus this branch of research was never examined.

The reason for this limitation comes from the proof formulæ used; the proof formulæ rely upon the principle of induction. Thus, when a counter example is discovered for an arbitrary state f, there is no information on how to get from the initial state to state f. See *Fig. 8.1*, which demonstrates, that in order to get from the initial state b to state f there are many ways. Supposing the state transition diagram has not been constructed, then a backwards reachability analysis is required. One possible solution is to start with state f and enumerate all possible states that lead to f, in the example from *Fig. 8.1* this would only be state d. Continuing this analysis recursively until the initial state has been enumerated, from state d there are 3 states, namely $\{b, c, d\}$, here b, the initial state has been enumerated, thus a trace of getting from the initial state to a counter example found in state f could be $b \longrightarrow d \longrightarrow f$, but there are many others, some including cycles allowing for an infinite number of different traces, although the use of fixed points can help here. If a counter example had been found in state s, there is no way of getting back to the initial state so the counter example is not valid.

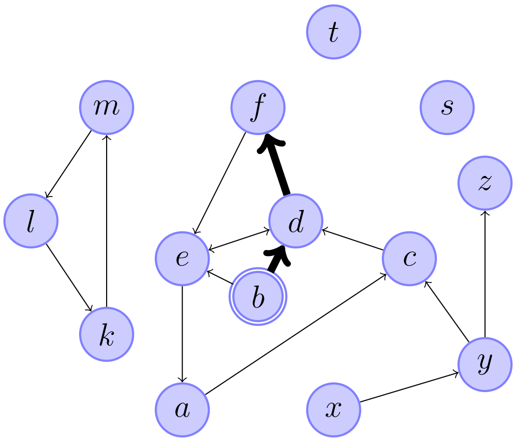


Figure 8.1: Each circle indicates a state, a state comprises of the configuration of the variables. The double lined state is the initial state and arrows define the transitions between the states.

This is a computationally expensive problem as the algorithm works backwards, *i.e. traversing the arrows of the state diagram in reverse.* Travelling in the direction of the arrows is relatively simple as these are logical conclusions.

8.4 Areas of Further Research

8.4.1 Verification versus Validation

This project is concerned with verification, *i.e. proving that the ladder logic fulfils an arbitrary safety condition*, but specification of these safety conditions is a separate issue. A typical suggestion is to derive the safety conditions from the control tables, i.e. *check that the ladder logic program fulfils the constraints within the control tables.* There are however other safety conditions which the control tables do not cover, these are the signalling principles.

Typically, verifying that the interlocking fulfils a signalling principle is straight forward assuming the signalling principles have been specified precisely. L.H. Eriksson while formally verifying the Swedish national railways was presented with this problem, [Eri97b, Eri97a]. A major part of the time Eriksson spent on the project was formalising the signalling principles. The signalling principles were in books written by signalling engineers, but while formalising these principles ambiguities were found, requiring a discussion with the engineers to resolve these ambiguities. Although the signalling principles appear to be comprehensive and have fulfilled the needs of the railway industry thus far, to aid in the formal verification these principles they need to be specified in a formal language which leaves no ambiguities.

8.4.2 Reachability

There has been a great deal of research in the area of backwards reachability, mainly in the field of model checking with finite state automata which could help with this project. Shankar Govindaraju and David Dill have described such a method in [GD98]. The method works by successive approximations of subsets of the total set of states. If these subsets are empty, then the counter example was not a real one, but in the cases where the approximation is not an empty subset, then there is a possibility that a real counter example has been found and requires further analysis. Hence the naming of the method. These approximations are built by working from the initial state and the counter example to find a path between these states, known as forward and backward reachability. This method was applied to the Stanford FLASH Multiprocessor and real counter examples were found.

A second method which requires modification of the proof formula is to simulate the interlocking for n cycles where n is sufficiently large. When a counter example is found, then a trace of how to reproduce the example can be extracted from the clause sets through a canonical procedure. The serious problem with this method is to select a sufficiently large n, it could be possible that a counter example is not found in n iterations, but one is found in $n + 1$ iterations of the ladder, no matter how unlikely. The one exception is when all reachable states have been visited and the $n^{th} + 1$ only visits a previously visited state for n.

8.5 Recommendations

8.5.1 Proof Formulæ

Throughout the course of the project a number of counter examples were found, many of these were later disregarded due to bugs in the software used to generate the proof formulæ.

The counter examples found that were real are already known by Invensys, details are omitted for legal and security reasons. These counter examples were said to only be intermittent, i.e. self correcting after one cycle and did not pose a safety issue. A typical execution cycle is variable between 0.5 and 1 second, [Wes06].

It is possible to check for these self correcting problems by changing the proof formulæ to check every cycle and second cycle, see the formula below.

$$\neg\psi \wedge \varphi_{\mathcal{L}} \rightarrow \psi'$$

where ψ is a safety condition, ψ' is the renamed safety condition and $\varphi_{\mathcal{L}}$ is a propositional model of the ladder. The proof formula states that, if there exists a state where the safety condition ψ does not hold, then in the direct successor state it must hold. This effectively allows a safety condition to be violated for a state such that it holds in the previous and next states. This can be depicted as a time line, see *Fig. 8.2* and *Fig. 8.3*.

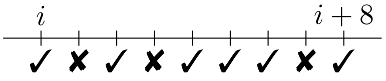


Figure 8.2: Valid Time Line

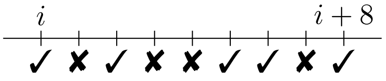


Figure 8.3: Invalid Time Line

The check marks indicate that the safety condition holds at the given state and a cross indicates that the condition does not hold. The time lines are arbitrary segments of greater time lines starting at the $i^{th} \geq 0$ iteration of the ladder and ending at the $i^{th} + 8$ iteration of the ladder. In the second figure, the time line is invalid because the safety condition does not hold for more than one consecutive iteration.

8.5.2 Proof Engine

The selected proof engine for this project was `OKSolver`, version 2002. This version only accepts DIMACS file formats; these are formulæ represented as clause sets, (*conjunctive normal form*). There are many other solvers which will accept formulæ not written in conjunctive normal form. An advantage of using a proof engine which accepts arbitrary propositional formulæ over the base $\{\wedge, \vee, \rightarrow, \leftrightarrow, \neg\}$ is that information is not lost. One major advantage of not first converting the proof formulæ into clause sets comes when equivalences are used, the proof engines can make use of this information to improve the efficiency. This improvement is particularly important when one of the operands is an atomic proposition and can be replaced by the other operand throughout the formula, reducing the search space. Typically there are many different methods of translating a formula into conjunctive normal form, these different methods have different impacts upon the proof engine, thus if the translation is left to the proof engine, then the proof engine will choose the most appropriate method available to it.

8.5.3 Model Checking

The use of SAT applied to this problem works well, but perhaps a more applicable method would be to apply model checking. Model checking has been used successfully for many verification applications, and lots of research has been done in various different areas of model checking.

One area which model checking has been used successfully is verifying finite state automata. Not only would model checking verify the ladder, it would also provide a backwards reachability trace of how to reproduce the counter example which would be of great interest to Invensys.

8.6 Implementation

The project included implementation of a prototype system which can formally verify whether ladder logic specifying an interlocking satisfies arbitrary signalling principles.

This required building a topological model of the railway yard, then using this model to build safety conditions from the signalling principles.

The verification stage takes as input, a safety condition and the ladder logic specification[2], this results in a two clause sets being generated, one for

[2] File format of the ladder logic specification is property of Invensys.

the base case and one for the inductive step. These clauses sets are then input into a SAT solver, which determines satisfiability of the cases.

Appendix A

Userguide

A.1 Introduction

This is a short user guide that explains important information required to utilise the software for the rail verification tools.

The guide is split up into three sections, installation, architecture and usage. Throughout this guide there are web links, these links are intended to provide extra information that will help with the understanding and increase proficiency but not is essential reading.

It is assumed that the user will have read the thesis accompanying this user guide as it explains what is going on conceptually, especially understanding the results requires this knowledge.

A.1.1 Notations

Within this user guide the notion of a general safety condition is equivalent to a signalling principle and specific safety condition is equivalent to safety condition as used in the thesis.

Symbol	**Units**
B	1 Byte
KB	1024 Bytes
MB	1024 KB
GB	1024 MB
˜	Home directory
makefile	See `http://www.gnu.org/make/`
quotation mark	`"`

Table A.1: Symbols and Units

This is a Document
This is a Process

Figure A.1: Flow Diagram Symbols

Flow Diagrams *Fig. A.1* shows the basic symbols for flow diagrams, documents (*i.e. data*) and processes. Data flow is shown by an arrow between processes or documents. Data flow could be a file read/write operation or sending the output from a program to another program.

A.2 Installation

If only testing or performing a clean install (including Linux) then see Sect. A.2.3.

A.2.1 Requirements

Currently only install packages for debian Linux[1] have been produced, but it is not necessary to run debian, the programs will work on Red Hat Linux, although will have to be manually installed.

Hardware requirements are as follows:

- 512MB ram (1GB if running as ‘live cd’), 2GB Recommended.
- At least 1GB free hard disk space after Linux install.
- Printer.

[1] see http://www.debian.org

The following software is required by various phases, assuming you have a working Linux install:

GNU Make Version 3.81

`http://www.gnu.org/software/make/`

Latex A complete latex install is required for building of the documentation. `pdfTeX 1.40.3` was used during development which is part of `texlive`.

`http://www.tug.org/applications/pdftex/`

`http://www.tug.org/texlive/`

dviconcat This is sometimes included in a complete latex install or in some axillary package such as `dviutils`.

`http://www.ctan.org/tex-archive/help/Catalogue/entries/dviconcat.html`

These software packages are usually available through the distributions repositories. They are all available for Ubuntu, and are pre-installed on the live CD's.

A.2.2 Process

There should be four `*.deb` files:

1. oksolver-X.Y.Z-i386.deb
2. railverifier-X.Y.Z-i386.deb
3. railverifier-doc-X.Y.Z-all.deb
4. generalformula-X.Y.Z-all.deb

Where X, Y and Z are version numbers.

Install these files in the order listed, depending on the Linux distribution there are different ways of achieving this. Usually double clicking on the file will start the installer. **NOTE:** if they are installed out of order then dependency errors will be raised, also a connection to the Internet will be required as other dependency's are automatically resolved.

A.2.3 Live CD

The live CD has two purposes, firstly it is a computer operating system that is executed upon boot, without installation to a hard disk drive and secondly it can install to the hard disk drive a new operating system.

The first feature will not affect the existing software on the computer but there will be a large performance hit, also any changes made and files saved will be discarded when the system is rebooted as it is running from RAM. To use this mode place the live CD in the computer and reboot, the BIOS might need to be reconfigured to boot from CD. Once the CD has loaded, select your language from the list and then select the first option which reads "Try Ubuntu Without Any Change To Your Computer". Ubuntu should boot up with all the required software installed. Be prepared to spend 10 minutes or more waiting for the system to boot.

The second feature is recommended for execution speed and files can be saved to the hard disk drive such that they persist after a reboot. The process is similar to the above, except when the CD first loads select "Install Ubuntu". This will boot into Ubuntu and launch the installer. It is the authors **recommendation** that this only be done on a machine which has no valuable data on, preferably a machine without any operating system installed. It is possible to "*dual boot*" a computer such that multiple operating systems are installed, a menu is displayed at boot and the required operating system can be selected. Dual booting is not covered in this document.

A.2.4 Built-In Help

This userguide is also on the CD along with example signalling principles that are explained in the thesis. The userguide can be located at

```
/usr/share/verifier/userguide.pdf
```

and the examples are located

```
/usr/share/verifier/examples/
```

Within this directory are 6 `*.gen` files. Each of these filese contains one signalling principle decribed by the file name.

A.3 Software Architecture

The process of general proving safety conditions has been split into four parts, conversion of the general safety conditions into grounded safety conditions and three phases, 1, 2 and 3, that control the verification process.

***Phase 1* - Clause Set Generation** Translates the grounded safety condition groups into clause sets that describe the problem.

***Phase 2* - Proving** The clause sets are entered into a proof engine, the results are documented.

***Phase 3* - Documentation** All the documentation is gathered and concatenated into a single document.

General safety conditons are generated using a program called `general`. The following sections describe the the architecture of the four parts and the whole system.

A.3.1 Top Level

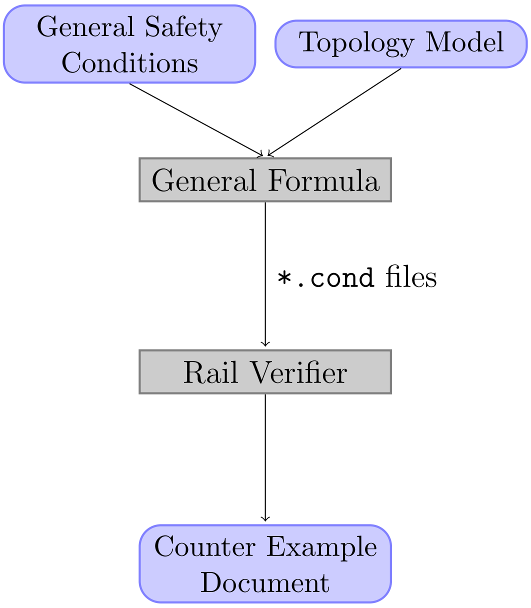


Figure A.2: Top Level Dataflow

A.3.2 General Formula

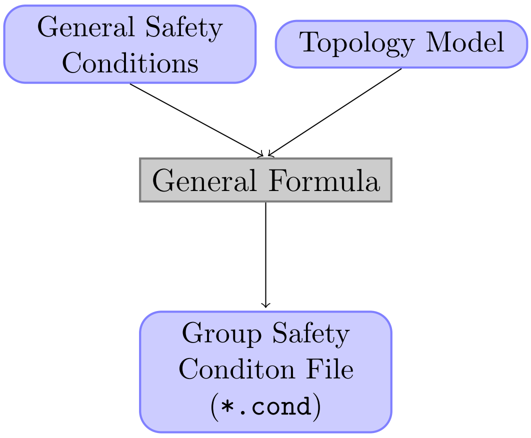

General Safety Conditions
Topology Model
General Formula
Group Safety Conditon File (*.cond)

Figure A.3: General Formula Architecture

A.3.2.1 Programs

Program	Description
`general`	Runs the General Formula program without a graphical user interface.
`generalgui`	Runs the General Formula program with a graphical user interface. Useful for development of formulas.

Table A.2: General Formula programs

A.3.2.2 Files

Along with the files listed in *Table A.3*, there is a directory located at:

```
/usr/share/generalformula/java/
```

which contains all the Java classes and Java sources (in case modifications are required). These classes are executed by running the programs in *Table A.2*.

File	Description
`model.pl`	A topology model of the station to be verified written in Prolog[2]. The file should be located in `/usr/share/generalformula/` directory.
`gnuprolog.jar`	An open source Java/Prolog library licenced under LGPL[3]. Should be located in `/usr/share/generalformula/` directory.

Table A.3: General Formula files

A.3.2.3 Usage

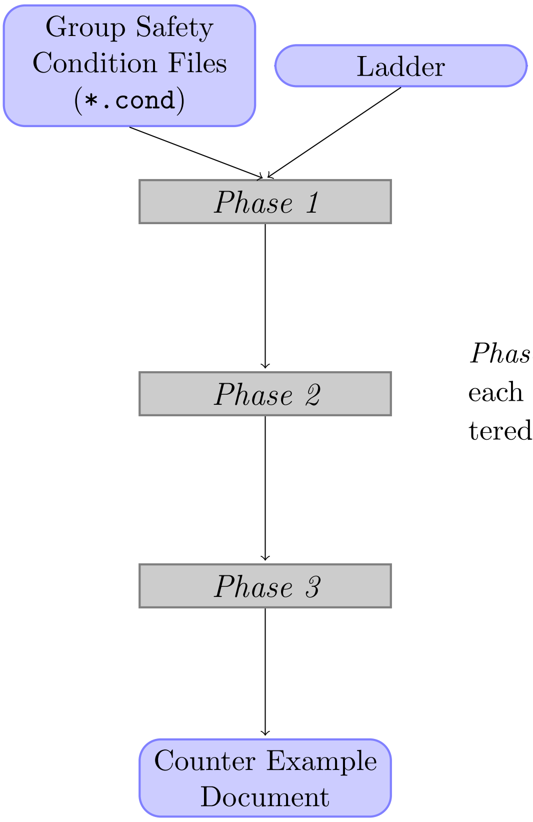
Only usage of `phase1` is covered here. The other programs are not intended for use by the end user, they are used internally by other programs and scripts.

A.3.3 Rail Verifier

Rail Verifier componet is a "*meta process*", meaning it acts like a single program but consists of other programs, in this case *Phase 1*, *Phase 2* and *Phase 3*. These phases are described in subsequent sections. See *Fig. A.4* for the architecture of Rail Verifier.

[2] `http://www.gprolog.org/`

[3] `http://www.gnu.org/copyleft/lgpl.html`



Phase 2 is executed once for each group condition file entered during *Phase 1*.

Figure A.4: Rail Verifier Architecture

A.3.4 *Phase 1*

Requires a directory with group safety condition files[4] in. For each of these files set up a directory structure (*environment*). Then populate this environment with the generated clause sets.

This phase can take a long time, for each safety condition a wait of a couple minutes can be expected.

See *Fig. A.5*.

[4] `*.cond`

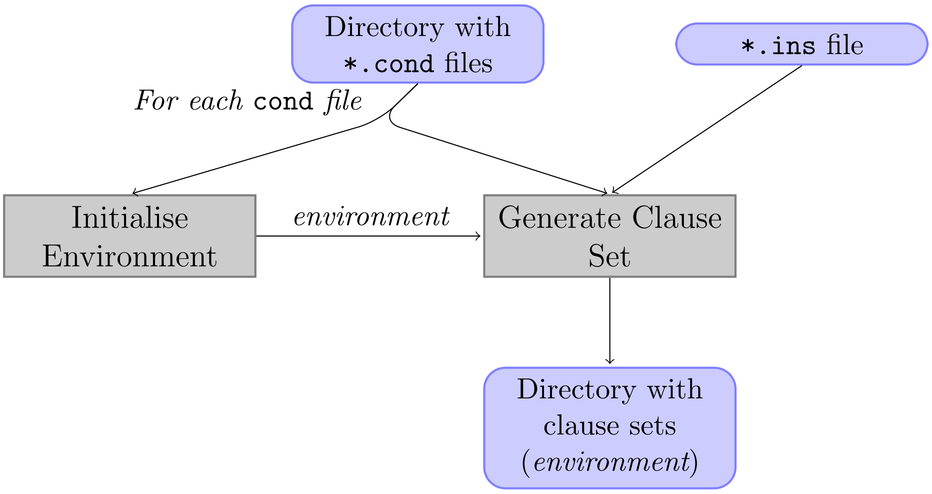


Figure A.5: *Phase 1* Architecture

A.3.4.1 Programs

Program	**Description**
`phase1`	Main script that should be executed, controls *Phase 1*.
`clausegen`	Takes a group safety condition and generates all the associated clause sets.
`setup-env`	Sets up a directory structure/environment.

Table A.4: *Phase 1* programs

A.3.4.2 Files

File	**Description**
`master.mak`	Makefile, not used in the phase but required when initialising the environment as links are made to this file. The file should be located in `/usr/share/verifier/` directory.

Table A.5: *Phase 1* files

A.3.4.3 Usage

Only usage of `phase1` is covered here. The other programs are not intended for use by the end user, they are used internally by other programs and scripts.

Help generated by phase1:

```
Usage: phase1 GroupSafetyConditionDirectory EnvironmentPath [GCSFile]

    GroupSafetyConditionDirectory - Directory with *cond files

    EnvironmentPath - Directory to contain the outputs, i.e. the generated
                      clause sets.

    GCSFile - Can be omitted if there is a ladder file located:
                /usr/share/verifier/EPPOM700.INS
              Otherwise it should be a path to a ladder in the
              GCSSv3 format.

Environment:
    To specify where to find clausegen set CLAUSEGEN_PATH env var.
    To specify where to find setup-env set SETUPENV_PATH env var.
    To specify where to find GCSFile set GCSFILE_PATH env var.
```

Example Supposing ~/saftyconds/ contains *.cond files and the generated clause sets should be placed under ~/clausesets/ then the following command can be executed at a terminal:

```
> phase1 ~/saftyconds/ ~/clausesets/
```

Phase 1 takes a long time, be prepared to **wait** also depending on the number of safety conditions *Phase 1* can consume large quantities of hard disk drive **space**. It has been witnessed consuming over half a gigabyte.

A.3.5 *Phase 2*

Takes an environment path populated with clause sets, attempts to satisfy the clause sets, in cases where clause sets are satisfiable, document the counter example.

Phase 2 requires lots of helper programs and source files for producing the documentation.

The makefile that is linked in *Phase 1* is the main script for *Phase 2* but is triggered by phase2 script.

See *Fig. A.5.*

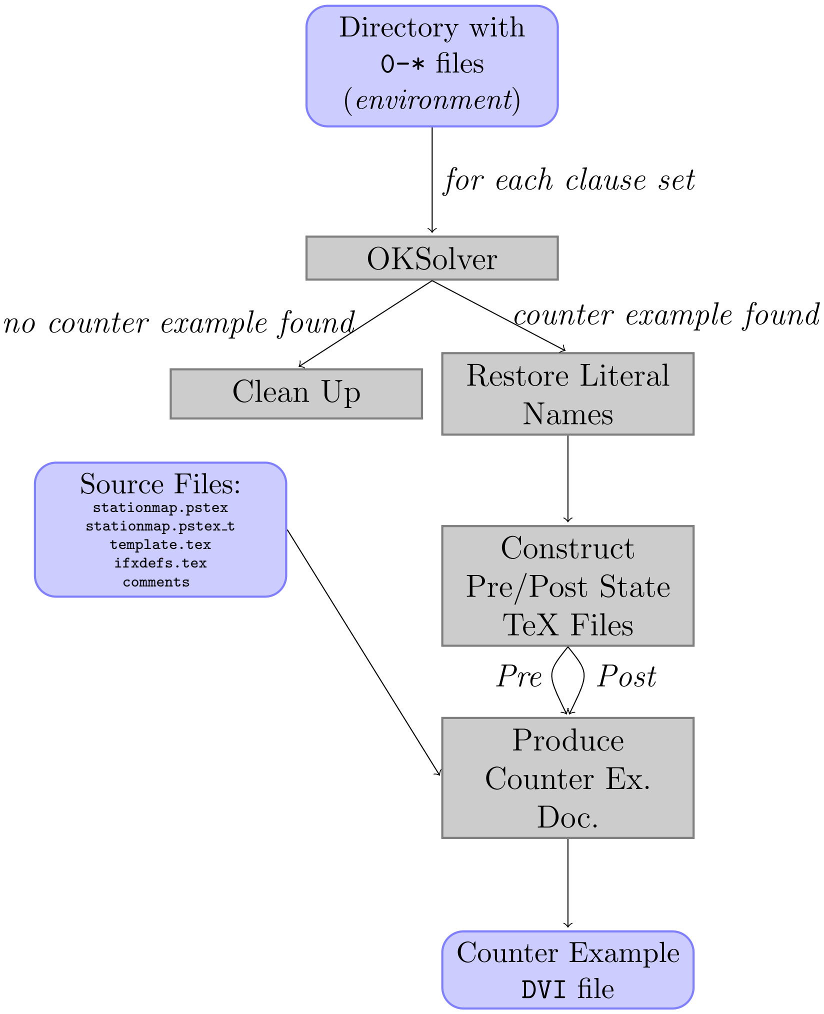


Figure A.6: *Phase 2* Architecture

A.3.5.1 Programs

Program	**Description**
`phase2`	Main script that should be executed, triggers the makefile.
`oksolver`	OKSolver, proof engine currently being utilised for verification. There are also numerous other versions of OKSolver in the `/usr/bin/` directory, only versions which contain `osa`[5] in there name are useful for this phase.
`oksolver.tested`	Wrapper script for `oksolver` that makes its return value conform to standard return values. *i.e. return 0 when clause set is satisfiable, >0 otherwise.*
`hex2ascii`	Processes the output from `oksolver`, converts literals from hex strings into ASCII strings; `buildtexdefs` and `texlink` process the output of `hex2ascii`.
`buildtexdefs`	Requires an output from `hex2ascii`, produces a tex file which contains lots of macros for representing the states of variables and timers. Only needs to executed once for each model, does not contain any state information from the counter example. The output of this program is dumped to the terminal, it should be redirected to a file called `ifxdefs.tex`.
`texlink`	Complement to `buildtexdefs`, takes a counter example processed by `hex2ascii` and produces a tex file which encodes the state of the counter example in the form of `def` macros.

Table A.6: *Phase 2* programs

A.3.5.2 Files

All the files in this section should be located in the `/usr/share/verifier/` directory. In most cases the files can be moved assuming the correct envi-

[5] Output Satisfying Assignment

ronment variable is configured.

File	Description
`stationmap.pstex`	A postscript file with a small diagram of the station; used for showing counter examples. Generated by `Xfig`.
`stationmap.pstex_t`	A tex file showing where to overlay text onto the postscript file above. Generated by `Xfig`.
`template.tex`	A tex file with the basic structure for the counter examples.
`titletemplate.tex`	A tex file template of the title page for each safety condition group.
`ifxdefs.tex`	Pre generated for the station, each new ladder will require this to be regenerated via `buildtexdefs`.

Table A.7: *Phase 2* files

A.3.5.3 Usage

Usage of `phase2` is covered here. The other programs are not intended for use by the end user, they are used internally by other programs and scripts.

Help generated by `phase2`:

```
Usage: phase2 EnvironmentPath

    EnvironmentPath - Directory that contains the output from
                      phase1, typically a directory with a
                      subdirectory for each safety condition group.

Environment:"
    none, see the master.mak for more information about this phase,
    this file should be linked in each of the sub-dirs and
    located at /usr/share/verifier/
```

Example Supposing `~/clausesets/` contains the output from *Phase 1*, the following command can be executed at a terminal:

```
> phase2 ~/clausesets/
```

Phase 2 does not typically take a long time. Although a lot of text is displayed on the console screen during processing due to the documentation generation. Often errors will be displayed during this phase, these are produced no counter example is found. These errors can be safely ignored.

A.3.6 *Phase 3*

Phase 3 is the final phase, all the documentation produced in *Phase 2* is collected and front sheets are produced for the various safety condition groups. The result is a `pdf` file containing all the counter examples.

In comparison to *Phase 1* and *Phase 2*, *Phase 3* is a much simpler phase, it consists of a single script and only processes documentation. See *Fig. A.7*.

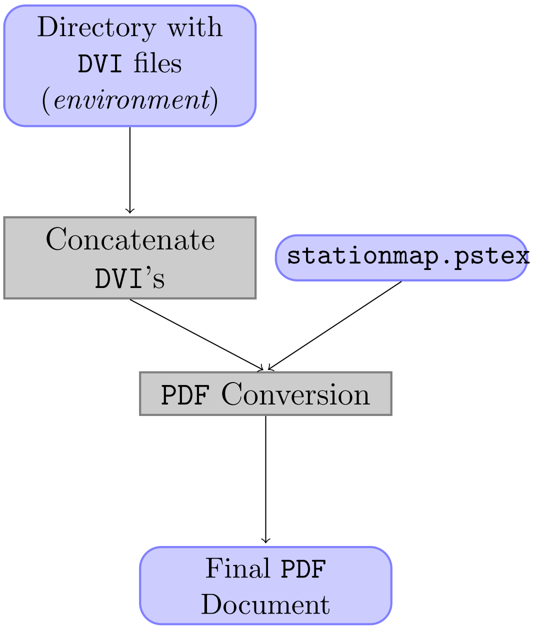


Figure A.7: *Phase 3* Architecture

A.3.6.1 Programs

Program	Description
`phase3`	Main script that should be executed, controls the concatenation process.

Table A.8: *Phase 3* programs

A.3.6.2 Files

File	Description
`stationmap.pstex`	A postscript file with a small diagram of the station; used for showing counter examples. Generated by `Xfig`. This file is required to convert the `DVI` into a `PDF`. Should be located in `/usr/share/verifier/`

Table A.9: *Phase 3* files

A.3.6.3 Usage

Usage of `phase3` is covered here. The help generated by `phase3`:

```
Usage: phase3 EnvironmentPath

    EnvironmentPath - Directory that contains the output from
                      phase2, typically a directory with a
                      subdirectory for each safety condition group.
                      There should also be *dvi's generated by
                      phase2 for this phase to succeed.
```

Example Supposing `~/clausesets/` contains the output from *Phase 2*, the following command can be executed at a terminal:

```
> phase3 ~/clausesets/
```

After executing there should be a `PDF` file located in `~/clausesets/ces.pdf`.

A.4 Usage

A.4.1 File Formats

There are different formats for the system, the most important being the general safety condition files. Each file should contain exactly one general formula. Other formats include the safety condition files and outputs from the solver. The topology model[6] is discussed in Sect. A.4.1.4.

[6] See `http://www.gprolog.org/manual/gprolog.html` for comprehensive usage of this file.

A.4.1.1 General Safety Condition

General safety condition file format is essentially a formula that relies upon definitions in the topology model.

A formula can be defined using **E**xtended **B**ackus**N**aur **F**orm (*EBNF*). Defining the atomic parts of the grammar as follows,

```
literal ::= " ([A..Z]|[a..z]|[0..9]|[ ./\()_-])* "
identifier ::= ([A..Z]|[a..z])+

and ::= AND
or ::= OR
not ::= NOT
equiv ::= EQUALS
imply ::= IMPLIES
all ::= ALL
some ::= SOME
lparen ::= (
rparen ::= )
comma ::= ,

type ::= Signal | TrackSegment | Route | Point
       | CombinedPoint | Releasable
```

The difference between a `literal` and a `identifier` is simple, `literal`'s require that they are encapsulated by quotation marks, can contain symbols and white space. The `identifier` is a restricted version that does not require quotation marks and can only range over standard English alphabetic characters.

The main body of the EBNF follows, it has been organised in the form for LL(k) parsers. Most notable consideration for LL is that the grammar is not left recursive, *i.e. recursion should happen on the right most non terminal.*

```
formula ::= <universalformula>

universalformula ::= <all> <literal> : <type> <universalformula>
                   | <existentialformula>

existentialformula ::= <some> <literal> : <type> <universalformula>
                     | <andexpression>

andexpression ::= <orexpression> [ <and> <andexpression> ]
```

```
orexpression ::= <impexpression> [ <or> <orexpression> ]

impexpression ::= <eqexpression> [ <imply> <impexpression> ]

eqexpression ::= <negateexpression> [ <equiv> <eqexpression> ]

negateexpression ::= <not> <atomicexpression>
                   | <atomicexpression>

atomicexpression ::= <literal>
                   | <predicate>
                   | <bracketedformula>

predicate ::= <identifier> <lparen> <variablelist> <rparen>

variablelist ::= <variable> [ <comma> <variablelist> ]

variable ::= <literal> | <identifier>

bracketedformula ::= <lparen> <formula> <rparen>
```

Examples Some example strings that would be accepted by this grammar along with there meanings follow.

- `"MAINT"`

  The mnemonic `"MAINT"` is always high in the ladder.

- `"PHASE1" AND "PHASE2"`

  The system is always in phase 1 and phase 2, obviously should not be the case.

-
  ```
  ALL cpt : CombinedPoint
    (NOT normal(cpt)) OR (NOT reverse(cpt))
  ```

  All points in the railyard can never be driven normal and reverse. This principle can be written equivalently (*which might be more intuitive*) as

  ```
  ALL cpt : CombinedPoint
    NOT (normal(cpt) AND reverse(cpt))
  ```

- `ALL rta : Route`

```
  ALL rtb : Route
    ALL ts : TrackSegment
      NOT equal(rta,rtb)
      IMPLIES
      (
        (part_of(ts,rta) AND part_of(ts,rtb))
        IMPLIES
        NOT (routeset(rta) AND routeset(rtb))
      )
```

Conflicting routes can not be set. Intuitively, this formula has the meaning that all routes that share a track segment can never be set simultaneously.

-
  ```
  ALL rt : Route
    ALL cpt : CombinedPoint
      point_part_of(cpt,rt)
      IMPLIES
      (
        (routeset(rt) AND NOT released(cpt))
        IMPLIES
        (
          (
            pointnormal(cpt,rt)
            IMPLIES
            (normal(cpt) AND NOT reverse(cpt))
          )
          AND
          (
            pointreverse(cpt,rt)
            IMPLIES
            (NOT normal(cpt) AND reverse(cpt))
          )
        )
      )
  ```

  If a route is set, then for all points in the route which are not released, must always be normal or reverse as the route demands. This final example looks complicated but demonstrates the expressive power of the proof system. The reader is advised to study this condition example to understand how and why it works.

If an incorrect string is entered into the program an error will be produced. The common errors include:

- *Existence* – The predicate indicated can not be reduced to a literal or a constant Boolean value.
- *Parse* – The string entered has not been accepted by the parser, possibly a misspelled keyword or incorrect bracketing is responsible.

All the errors can not be listed as not all errors can be predicted. I.e. *out of memory error*. It is possible that the safety condition entered here will introduce new mnemonics causing counter examples to be identified erroneously. For example, suppose the following string is entered:

```
ALL pt :  Point (NOT normal(pt)) OR (NOT reverse(pt))
```

This is wrong as `pt` is quantified over the type `Points` whereas `CombinedPoints` would be the correct type. Instead of producing two safety conditions in the example station for the combined points *cpt1* and *cpt2* the program will produce four safety conditions for the points *pt1*, …, *pt4*. Then when reducing the predicates `normal/1` and `reverse/1` the mnemonics `pt1.NL`, …, `pt4.NL` and `pt1.RL`, …, `pt4.RL` will be produced erroneously. The correct mnemonics should be `cpt1.NL`, `cpt1.RL`, `cpt2.NL` and `cpt2.RL`. If this error is made, the program will not give any warnings.

The predicates `normal` and `reverse` in the previous example are defined in the Java source file `StationPredicateResolver.java`. These predicates; among others enforce the stations naming conventions, it was not deemed necessary to define a new file format for these conventions as the program is currently limited to only the station.

Table A.10 lists all currently implemented predicates in the above mentioned source file:

Predicate	**Description**
`id(o)`	Used for debugging, returns the value passed in. i.e. `id(o)=o` Will most likely not be of use for actual safety conditions.
`occupied(ts)`	Indicates that the track segment ts is occupied, appends `.T` onto the end of `ts`. i.e. `occupied(ts)=ts.T`
`normal(o)`	Normal Latch, can be used for points and other track side equipment, appends `.NL` onto the end of `o`. i.e. `normal(o)=o.NL`

Continued on next page

Table A.10 – continued from previous page

Predicate	Description
`reverse(o)`	Reverse Latch, can be used for points and other track side equipment, appends `.RL` onto the end of `o`. i.e. `reverse(o)=o.RL`
`released(o)`	Indicates that `o` has been released locally, appends `.REL` onto the end of `o`. i.e. `released(o)=o.REL`
`freenormal(o)`	Free to go normal latch, can be used for points and other track side equipment. Appends `.NWZ` onto the end of `o`.
`freereverse(o)`	Free to go reverse latch, can be used for points and other track side equipment. Appends `.RWZ` onto the end of `o`.
`routeset(rt)`	Indicates that the route `rt` is set, appends `.RU` onto the end of `rt`.
`custom(lt,suffix)`	This predicate allows for custom mnemonics to be generated by appending suffix to lt. This predicate can define the above predicates. i.e. `custom(lt,suffix)=ltsuffix`. `lt` and `suffix` can be either a literal or a variable.

Table A.10: The stations Predicates

A.4.1.2 Safety Condition

The `cond` files store grounded safety conditions, ideally each `cond` file stores a group of related safety conditions. Each safety condition is stored along side a comment that is used for documentation purposes, the generated `cond` files use this comment space for displaying the formula that is being proved.

The file consists of many records, where the each record resembles a safety condition. A record is of the form:

```
[name]
#
comment
#
condition
```

where the `comment` and its encapsulating hash signs are optional. `name` is the name of the safety condition, this is used when the clause sets are generated to name the file.

The `condition` is a simple propositional formula consisting of the base conjunction, disjunction, negation, implication and equivalence, see *Table A.11* for the syntax of these operations. The literals in the formula must be encapsulated by quotations marks, this is to allow them to consist of *exotic* symbols. The characters that can be used for the literals are any printable characters that do not include the quotation mark.

Operation	**How to Write**
conjunction	`&`
disjunction	`|`
negation	`~`
implication	`->`
equivalence	`<->`

Table A.11: Syntax of Operations in `cond` files

Examples The following are examples of records in a `cond` file:

```
[condition1]
#
this is a comment for condition 1
#
"a" -> (~"b" | ("c" & "b"))
```

This condition is called `condition1`, has a comment and a condition that has the semantics of a implies that b is false or c and b are true.

```
[condition2]
"START" <-> ~"START"
```

Simple condition that is always false, specifies that `START` must be true and false simultaneously.

```
[condition3]
#
The condition to prove is
\[
pt.locked \rightarrow \neg pt.cangoleft
\]
```

```
with the meaning: if point is locked, then it is not free
to go left
#
"pt.locked" -> ~"pt.cangoleft"
```

The comment in this condition demonstrates the use of LaTeX code in the comments. The line '`pt.locked \rightarrow \neg pt.cangoleft`' in the comment will produce, if counter example is identified in the documentation:

$$pt.locked \rightarrow \neg pt.cangoleft$$

There is no limit on the complexity of the formulas that can be entered. Selecting the correct literal names is important. If names are used that do not exist in the ladder then they will be created during the verification stage. This results in typing errors leading to false results from the verification.

Typically the literals that should be used exist in the ladder postfixed by `_0` or `_1`. The number represents the version the variable. Each time a variable is modified then its number is version number is incremented. Some input variables do not have a `1` version as they are not modified during the ladder cycle, all outputs and latches are recomputed every cycle so have a `1` version as well as a `0` version.

A.4.1.3 Solver Output

There are two outputs from the solver, the *raw* version is encoded by converting all the literals into hex encodings of the ascii values. This is because the literals used in the ladders contain symbols that are not supported. The *raw* output is passed through `hex2ascii` to restore the literal names. Typically the renamed versions of the outputs are postfixed by `.namesrestored`.

The actual format of the file is as follows, lines beginning with `c` are comments. All the literals used in the clause set are listed in the file, those that were found satisfied for the counterexample are printed normally those that were falsified are prefixed with a `-`, minus sign.

A.4.1.4 Topology Model

The topology model of the railway yard is written in a programming language called `Prolog`, this section is a basic tutorial of `Prolog` necessary to write a topology model.

There are three basic statements used to write programs, *facts*, *rules* and *queries*. Facts are the basic building blocks of all programs, an example of a fact is

```
person(george).
```

which asserts that george is a person, and

```
sister(julie,george).
```

which asserts that julie is the sister of george. Facts are used to assert information about objects and relationships between these objects. In the context of the topology model, facts are used to define the names of railway yard entities, the types of these entities and relations ships between these entities. Facts are the most important part of the topology model.

Facts can effectively can build up a simple database structure, a database table person about people can be constructed in prolog by

```
person(george,12,male).
person(julie,14,female).
```

where each entry in the table has three pieces of information, a name, age, and gender, in that order. From this table it can be seen that julie is 14. An alternative representation of the same information can be achieved as follows

```
person(george).
person(julie).

age(george,12).
age(julie,14).

gender(george,male).
gender(julie,female).
```

where the person predicate acts as a primary key in the table and age and gender are other columns relating an attribute to a given primary key entry. The second representation is preferred as it allows for only relevant information to be gathered and the table is much easier to modify.

Queries can be asked of these databases, typically a query results in a truth value. It could be asked if george is a person, in the first database representation this would be asked as (when entered into the GNU Prolog interpreter)

```
| ?- person(george,_,_).

yes
```

where the underscores represent values which are undefined. The result is `yes`, meaning that the query is a logical conclusion of the facts already asserted. Similarly for the second representation the query would be phrased as

```
| ?- person(george).

yes
```

Queries can also be used to ask general questions, such as *who are people and what are there ages?* This would be phased as follows for the first database representation

```
| ?- person(P,A,_).

A = 14
P = julie ? ;

A = 12
P = george

yes
```

where P and A are variables which have two sets of assignments (*solutions*) such that the query entered is satisfied. These assignments are computed by prolog and displayed. The second database representation uses multiple predicates, thus the query must use the relevant ones, the query can be phrased as

```
| ?- person(P), age(P,A).

A = 14
P = julie ? ;

A = 12
P = george

yes
```

where P and A are as before. The query can be interpreted as P is a person *and* P has age A. The comma in the query is interpreted as a logical conjunction and the conjuncts are evaluated from left to right.

These queries can be made up from conjunctions using a comma and disjunctions using the `|` symbol, also braces are allowed. Supposing the following facts are also entered into the database

```
person(alice).      age(alice,13).      gender(alice,female).
person(matt).       age(matt,16).       gender(matt,male).
```

It is possible to ask interesting questions of the database such as list all females and males of the age 12 or under.

```
| ?- person(P), (gender(P,female); (gender(P,male),age(P,A),A=<12)).

P = alice ? ;

P = julie ? ;

A = 12
P = george

yes
```

Rules are used to specify general relations between objects, typically rules "wrap" queries. Rules are used in the context of the topology model to dissatain relations between entities automatically, i.e. which position a point should be in for a given route. A rule has the syntax of a fact, with the exceptions variables can be used as operands and all rules have a body consisting of a query. The query above can be translated into a rule called `womanandchildren` as

```
womanandchildren(P) :- person(P),
                       (
                        gender(P,female);
                        (gender(P,male),age(P,A),A=<12)
                       ).
```

This rule behaves as multiple facts, when the rule is queried the following is observed:

```
| ?- womanandchildren(P).

P = alice ? ;

P = julie ? ;
```

```
P = george

yes
```

The information in this section should be enough to understand subsequent sections on the structure of the topology model file. Sterling and Shapiro have written a good book explaining Prolog in much greater depths called *The Art of Prolog*.

A.4.1.5 External Formats

There are a range of external formats used by the program, the most important being `GNU Prolog` as discussed in the previous section. The documentation is built using `Latex` which uses `tex` files. The `tex` format is very powerful, originally developed for type setting mathematical equations. A good resource explaining the format can be found at:

`http://www.tug.org/texlive/doc.html`

and

`http://www.latex-project.org`

A.4.2 Proof Cycle

A typical proof cycle consists of entering a general formula to produce a `*.cond` file, many of these files can be produced and placed into a folder. The path of this folder will be passed to the `railverifier` program along with a folder to hold the temporary files and generated counter example documentation.

Example Supposing the directory `~/conditions/` contains `*.cond` files, the following command can be executed:

```
$ railverifier ~/conditions/ ~/environment/
```

The directory `~/environment/` will contain all a sub directory for each `*.cond` file in `~/conditions/`. Each of these sub directories will contain generated clauses sets and other files used for the post processing of the counter examples. If counter examples are found, then a file called `ces.dvi` will exist in the sub directories. During phase 3 all of these `ces.dvi` files are gathered together and concatenated into a file called `~/environment/ces.pdf`.

A.5 Produced Counterexamples

Each counterexample consists of *two* sections and an optional comment about the example which shows the actual formula that has been disproved. Firstly, a simple line diagram of the station which shows which track circuits are occupied and the aspects of the signals[7] and secondly the last section shows various mnemonics[8] from the ladder and there logical states.

The name of the counterexample can be appended by "`noinitial`" to indicate that this is an arbitrary cycle through the Westrace as opposed to the initial cycle.

A.5.1 Comment

The comment is an optional section, it is only added if there is a comment for the current counterexample. Typically the comment will show the condition that was violated in propositional logic. The syntax used for the logic is as follows:

Operation	**Symbol**
And	$\wedge$
Or	$\vee$
Negation	$\neg$
Implication	$\rightarrow$

A.5.2 Pictorial of Track Circuits

Track segments are shown as 'H' like objects with a number written above or below them depending whether the track segment is on the top or bottom line. If a train is on the track segment then this is shown by writing an X over the track segment.

Signals can show red, or green but typically show red when a counterexample occurs. The colour is drawn onto the diagram a coloured circle on the appropriate signal. The red aspect is shown closest to the vertical line at one end of the signal and the green at the opposite end.

The direction of the points as at the end of execution is found by querying the `NWC` and `RWC` latches and drawing a small arrow appropriately next to the point. If neither `NWC` or `RWC` are high, then a question mark is shown instead.

[7] Signal A8250 and A8251 do not have their aspects represented in the diagram.

[8] Variables

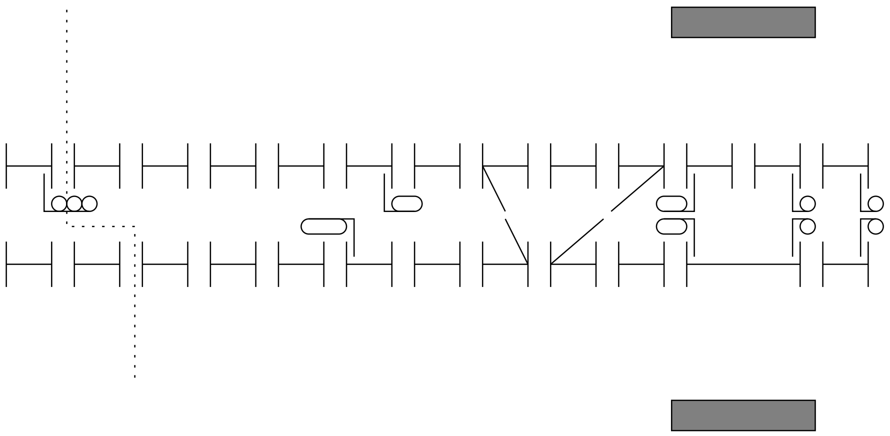


Figure A.8: Basic layout of the station, with all text removed.

A.5.3 Railway Yard States

All the mnemonics listed in the last section have two values associated with them, *pre* and *post* states. The first value is the value before the ladder is executed and after the reading of inputs, the second value is after the ladder has been executed. *i.e* given a variable x with the states *high*/*low* it translates to x being *high* before execution and *low* after execution.

The variables are split up into related groups, *routes*, *points*, *signals*, *phases* and *special*.

A.5.3.1 Timers

Each timer has two mnemonics/variables associated with it, namely, trigger and output. These are 3 valid states associated with each timer, *off*, *timing* and *elapsed*. *Off* is when trigger is low, *timing* is when trigger is high and output is low, *elapsed* is when both the trigger and output are high.

A.6 Expanding

A.6.1 Other Interlockings

Expansion of the software for different interlockings can be achieved. Currently, the major limitation of the software is within the `ins` file parser, the parser has only been matured to the extent where it can parse the stations `ins` file.

When translating from general signalling principles into specific safety conditions, resolving predicates into actual mnemonics from the ladder will cause problems as currently only the stations naming convention is supported.

A.6.1.1 Parser

The parser was written using two complementing tools for the Haskell programming language. Alex, which splits the input text into tokens (*lexical analyser*) and Happy which is a parser generator that uses a input syntax similar to Yacc for C. Happy is a LALR parser generator.

The Alex and Happy source files can be located in the `GcssParser` subdirectory for the GCSS parser and `ConditionParser` subdirectory for the `*.cond` file parser. Both of these directories are located within the main Haskell source directory. The Alex source files have a `*.x` extension and Happy source files have a `*.y` extension.

A.6.1.2 Predicate Naming

Ideally a further level of abstraction is required to allow for an arbitrary signalling principle to be converted into specific safety conditions for an arbitrary interlocking.

The inclusion of the `custom/2` predicate allows for a simple *hack* but is not ideal as each signalling principle will have to be manually edited when being applied to a new railyard. This is to reflect different naming conventions.

Currently, there is a Java source file which defines the predicates called `StationPredicateResolver.java`. The purpose of this file is to resolve specified predicates into mnemonics, in the case predicates are not specified within this file the topology model is queried.

Topology model querying is managed by `BasicPredicateResolver.java`. To expand the system for another interlocking, use this class as a base class, examine the structure of `StationPredicateResolver.java` for more insight on how to do this. Also, modify `Builder.java` in the `build()` method. Locate the following line

```
builder.visitors.Compiler c =
    new builder.visitors.Compiler(q,new StationPredicateResolver(r));
```

Replace `StationPredicateResolver` with the new predicate resolver class name.

A.6.1.3 Counter Examples

The figure used for the stations track plan will also have to modified to reflect the railyard being verified.

The picture was created using `xfig`, and exported using the "Combined PS/LaTeX (Both Parts)" option. This will generate two files, one will be a post script file of the actual image without any text and the other will be a latex file containing the text to be placed over the image. This allows for latex to process the text before it is shown allowing for the actual status of the track circuits and signals to be drawn on top of the image.

For the text to be processed a static latex file containing macro definitions is constructed for each different station. This file is built using `buildtexdefs`.

`buildtexdefs` reads the output from OKSolver and produces for each mnemonic two macros, one for each pre and post states of the mnemonics. These macros produce textual representations for the state of the mnemonics. This step only needs to be done once for each different interlocking being verified as mnemonics do not change.

The actual counter example is then passed through `texlink` which constructs a latex file containing definitions for mnemonics which are *high*. These definitions are used by the latex file produced by `buildtexdefs` to show the state of the mnemonics.

A.6.2 Different Proof Formulæ

To change the actual proof formulæ generated by the program is not particularly hard as Haskell is used but should be done with caution. A mistake in the proof formulæ can invalidate verification.

The file `Safety.hs` has a function `appendsafetyconds` which is responsible for constructing the proof formulæ. It has two cases, one for the base case and one for the inductive step.

The safety condition data type takes a number and returns a formula, this was originally intended to allow for an arbitrary number of iterations of the ladder to be executed. Then the safety condition would have the correct mnemonics after for the final iteration of the ladder. This is also the case for the invariants data type but this functionality has not been enabled for a long time within the code so will need a *careful analysis* before being used.

Bibliography

[Abr96] J.R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[Alo06] F.A. Aloul. Search techniques for SAT-based Boolean optimization. *Journal of the Franklin Institute*, 343(4-5):436–447, 2006.

[BBFM99] P. Behm, P. Benoit, A. Faivre, and J.M. Meynadier. Météor: A Successful Application of B in a Large Project. *Proc. FM-99-World Conference on Formal Methods in the Development of Computing Systems*, pages 369–387, 1999.

[BBV95] T. Basten, R. Bol, and M. Voorhoeve. Simulating and Analyzing Railway Interlockings in ExSpect. 1995.

[Bjø04] D. Bjørner. TRain: The Railway Domain. In *Building the Information Society*, volume 156/2004 of *IFIP International Federation for Information Processing*, pages 607–611. Springer Boston, 2004.

[Bus94] S.R. Buss. On Herbrands Theorem. *Lecture Notes in Computer Science*, 960:195–209, 1994.

[Coo71] S.A. Cook. The Complexity of Theorem-Proving Procedures. *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[Cri87] A.H. Cribbens. Solid-state interlocking(SSI): an integrated electronic signalling system for mainline railways. *IEE Proceedings B. Electric Power Applications*, 134:148–58, 1987.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DP60] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[EF99] L.H. Eriksson and M. Fahlén. An Interlocking Specification Language. *ASPECT IRSE*, 99, 1999.

[Eri97a] L.H. Eriksson. Formal Verification of Railway Interlockings. *Swedish National Rail Administration Technical Report*, 4, 1997.

[Eri97b] L.H. Eriksson. Formalising Railway Interlocking Requirements. *Swedish National Rail Administration Technical Report*, 3, 1997.

[FGHvV98] W.J. Fokkink, J.F. Groote, M. Hollenberg, and B. van Vlijmen. LARIS 1.0: LAnguage for Railway Interlocking Specification. *Report, CWI, Amsterdam*, 1998.

[FHG$^+$98] W.J. Fokkink, P.R. Hollingshead, J.F. Groote, S.P. Luttik, and J.J. van Wamel. Verification of interlockings: from control tables to ladder logic diagrams. *Proceedings 3rd Workshop on Formal Methods for Industrial Critical Systems (FMICS'98)*, pages 171–185, 1998.

[FM07] Z. Fu and S. Malik. Extracting logic circuit structure from conjunctive normal form descriptions. In *Proceedings of International Conference on VLSI Design*, January 2007.

[GD98] S.G. Govindaraju and D.L. Dill. Verification by approximate forward and backward reachability. *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 366–370, 1998.

[GK05] N. Galesi and O. Kullmann. Polynomial time SAT decision, hypergraph transversals and the hermitian rank. *The Seventh International Conference on Theory and Applications of Satisfiability Testing*, pages 76–85, 2005.

[GOMS04] E. Gregoire, R. Ostrowski, B. Mazure, and L. Sais. Automatic extraction of functional dependencies. *Proc. SAT*, 2004.

[Han94] K.M. Hansen. Formalising Railway Interlocking Systems. *Nordic Seminar on Dependable Computing Systems*, pages 83–94, 1994.

[Her71] J. Herbrand. *Logical Writings*. Springer, 1971.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Kin94] T. King. Formalising British Rails Signalling Rules. *FME '94: Industrial Benefit of Formal Methods*, 873:45–54, 1994.

[KR01] D. Kerr and T. Rowbotham. *Introduction to Railway Signalling*. Institution of Railway Signal Engineers, 2001.

[Kul08] O. Kullmann. Present and future of practical SAT solving. Technical Report CSR 8-2008, Swansea University, 2008.

[Lea03] M. Leach. *RAILWAY Control Systems*. Institution of Railway Signal Engineers, 2^{nd} edition, 2003.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[Mon92] M. Monigel. Formal representation of track topologies by double vertex graphs. *Proceedings of Railcomp 92 held in Washington DC*, 2:359–370, 1992.

[Noc02] O.S. Nock. *Railway Signalling*. Institution of Railway Signal Engineers, 2^{nd} edition, 2002.

[PV04] G. Pan and M.Y. Vardi. Search vs. symbolic techniques in satisfiability solving. *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, 2004.

[ROTS04] W. Reif, F. Ortmeier, A Thums, and G. Schellhorn. Integrated Formal Methods for Safety Analysis of Train Systems. In *Building the Information Society*, volume 156/2004 of *IFIP International Federation for Information Processing*, pages 637–642. Springer Boston, 2004.

[Sab04] D. Sabatier. Reusing Formal Models. In *Building the Information Society*, volume 156/2004 of *IFIP International Federation for Information Processing*, pages 613–619. Springer Boston, 2004.

[She04] D. Sheridan. The optimality of a fast CNF conversion and its use with SAT. *The 7th International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[Tse68] G.S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, 2:115–125, 1968.

[vD04] D. van Dalen. *Logic and Structure.* Springer, 2004.

[Vel04] M.N. Velev. Efficient translation of Boolean formulas to CNF in formal verification of microprocessors. *Proceedings of the 2004 conference on Asia South Pacific design automation: electronic design and solution fair 2004-Volume 00*, pages 310–315, 2004.

[Wes06] Westinghouse Rail Systems Australia, 179-185 Normanby Rd, South Melbourne, Victoria 3205, Australia. *WESTRACE Application Manual*, 9.0 edition, 2006.