Verification of Railway Interlockings in *SCADE*

Andrew Lawrence

February 18, 2011

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

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

Declaration

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

February 18, 2011

Signed:

Statement 1

This thesis is being submitted in partial fulfilment of the requirements for the degree of a MRes in Logic and Computation.

February 18, 2011

Signed:

Statement 2

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

February 18, 2011

Signed:

Statement 3

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

February 18, 2011

Signed:

Contents

Chapter 1

Introduction

1.1 Introduction

This thesis is concerned with the use of *SCADE* Suite (Esterel Technologies) for the verification of railway interlockings. This is a feasibility study done in co-operation with Invensys Rail, a leading international company for the design, construction, and validation of railway control systems.

1.2 Aim

In this thesis we will be exploring the verification of railway interlockings with a view to practical application in industry and investigating the use of a commercial piece of software *SCADE* suite from Esterel Technologies. We intend to answer the question: How well does *SCADE* suite perform the verification of code and what ways is it applicable to the verification of railway interlockings? We explore the application of model checking where we specifically concentrate on two different modelling approaches.

In the first approach we translate existing specifications of railway interlockings written in so-called Ladder Logic into *SCADE* and verify them. To do this a new formalism is introduced to capture the semantics of ladder logic using labelled transition systems. We first use *SCADE* to verify a small toy ladder logic program. In our first attempt the translation was done manually. To scale up this approach a tool was created to automatically translate ladder logic into *SCADE* language. Our work on formalizing ladder logic builds on two previous MRes projects by Kanso [Kan08] and James [Jam10]. The problem of verifying ladder logic was first approached by Kanso [Kan08] who was the first to systematically describe ladder logic and who developed a prototype translation and verification tool. James [Jam10] expanded the functionality of the prototype tool by adding model checking techniques which could be applied to the verification problem. We have applied *SCADE* to the example case studies considered in these two previously completed MRes Projects. These include two real world railway control systems of considerable size and complexity.

The second approach presented in this project was driven by the desire of industry to change

the way they model their systems. There is a drive away from low level languages towards more powerful high level languages. In a bid to drive down costs the concept of reuse is important. Industry would like to have a toolkit of components and pre-verified modules from which they can model new railways. This led to the development of a new modelling approach which embodies these points. An example railway was modelled to provide a case study into its use. Verification was then performed on our railway example, on individual components and on modules of components.

1.3 Thesis Outline

In Chapter 2 we will introduce a large amount of background information regarding the railway, its history and composition. We will give an introduction to ladder logic and describe the operation of the Westrace Railway Interlocking.

In Chapter 3 we will discuss the theoretical background under-pinning the *SCADE* suite. We will explore Ståmarck's Saturation method giving both an informal explanation using several examples and a formal explanation providing the theoretical background and implementation details. This is followed by a survey of another technique used to decide the satisfiability of propositional formulae namely binary decision diagrams. We provide a formal introduction to binary decision diagrams as well as a previously known result regarding their complexity.

In Chapter 4 we will explore the verification of ladder logic programs. We begin by introducing a toy example ladder logic program for a pelican crossing. Throughout this chapter we present several possible formalisations of this in the *SCADE* language. A new method is presented which captures the semantics of ladder logic programs using labelled transition systems. We then present an approach for the verification of ladder logic programs using the *SCADE* suite. This includes a automatic translation of ladder logic programs into the *SCADE* language. We follow this with a discussion of invariants and their addition to *SCADE* models. Finally we perform a comparison between *SCADE*, the tool produced by James [Jam10] and the KIND safety property verifier.

In Chapter 5 we will introduce a new modelling approach to capture railway domain in a concrete and modular fashion. This begins with a discussion of components modelled, some motivation for the modelling and a description of their behaviour. Following this discussion an example is presented of an abstract railway modelled using the aforementioned components. We then have a discussion of the verification of our new modelling approach. This includes the verification of topological properties and modules of components. We conclude this chapter with a comparison between the first approach and this new modelling approach.

Finally in Chapter 6 we provide a conclusion to this thesis. We provide a summary of the work performed and the results achieved. Then we conclude by discussing possible directions for future work.

The Appendix is split into two sections. The first contains rules for several proof systems including Stålmarck's saturation method. The second contains the *SCADE* language code for the concrete modelling approach.

Chapter 2

Background

From their birth in the 1800s to the present day, the railway and its control systems have seen many advances. Its control and safety has gone from being a completely manual human based system, to a mechanical system and finally to the electronic system we see today. We will now look at a brief history of the railway followed by information on our industrial partner Invensys Rail. We then look more closely at modern railways and the equipment which constitutes them. We also study Westrace interlocking which is produced by Invensys and the ladder logic programs which run on it. Finally, we look at some previous work in this field.

2.1 A History of Railway Signalling and Control Systems

Prior to the days of fixed signals, Policemen would be stationed at junctions and railway stations. They changed points manually and gave instructions to train drivers by using a system of either flags or oil lamps depending on the visibility. Since this was before the time of telecommunications and electricity there was no way of telling where a train was once it left a station and went out of sight. The only safety precaution that could be taken was to use an egg timer to delay the departure of the next train in order to give the previous train time to progress along the track. Train speeds were not very high during this period so this was an acceptable way of ensuring safety.

Modern railway signally makes use of **fixed signals**. These are permanently positioned by the side of the track and provide some visual information to the train driver. The original fixed signal consisted of a shaped wooden board that could be rotated on pole round a vertical axis. If the board was visible to the driver then he would have to stop the train. On the other hand if the driver couldn't see the board because it was side-on to him then he would be able to proceed.

One of the major developments in railway signalling was the introduction of the **Semaphore** fixed signal. These consisted of a board that could be moved into several preset positions. Typically these would have 3 different visible "aspects" which they could be set to: One aspect to indicate the driver can proceed, another that indicates the driver can proceed with caution and finally an aspect which indicates that the driver should stop.

Around about the same time as the introduction of the semaphore signal, the system for controlling the signals went under drastic change. The Policemen were replaced with professional **Signallers** whose job was specifically to manage the railways. A system of pulleys, wires and levers was also devised to allow multiple signals and points to be controlled from a central position. This central position became known as a **signal box** and was manned by one or more signallers. This centralisation allowed for further safety mechanisms to be installed. One in particular, namely the **interlocking**, is of interest to us. The interlocking physically locked levers if they were unsafe to move.

The next leap in railway technology came from the invention of the electronic **track circuit**. These would activate an indicator in the signal box if a segment of track was occupied by a train. As more and more track circuits became installed it was no longer necessary to have human intervention to control certain signals. **Automatic signals** were introduced which operated completely by track circuits without any intervention from human signallers. Around this time **electric point machines** were introduced removing a large amount of physical work performed by signallers allowing for a greater area of control for each signaller. Around this time electromechanical **relays** began to replace purely mechanical relays reducing the amount of space needed for a signal box.

In the 1920s **colour light signals** replaced mechanical semaphore signals these where much brighter than the oil lamps fitted to semaphores and greatly increased the safety of night time train travel. In the 1930s the mechanical levers were replaced with an electronic **control panel** containing switches and buttons. This allowed for the introduction of **route setting** where with the press of a button configurations of signals and points would be associated with a particular route could become activated. Prior to this time many levers would have had to have been pulled to set many different pieces of equipment. During the 1980s the most important advance from our point of view took place. The advent of electronic microprocessors enabled the replacement of the relay and mechanical interlockings with an electronic **solid state interlocking** system (SSI) [Cri87]. The main focus of this project will be to investigate the safety of such solid state interlockings.

2.2 Invensys Rail

Invensys Rail [Inv] and its previous incarnation Westinghouse Rail Systems Ltd have been involved for over 140 years in producing equipment to increase safety in the railway industry. Originally they produced air brakes for trains, these had a failsafe state such that if the power was cut the brakes would automatically stop the train. Later on in the company's development they provided support to British Rail when the first solid state digital railway interlocking was installed in Leamington Spa. Today they supply railway control equipment to companies based around the globe, including companies based in Australia, Hong Kong, Germany, Spain and the UK. This project is mainly concerned with one of the solid state railway interlockings Invensys produces called the Westrace. The Westrace railway interlocking continuously runs a ladder logic program which prevents the railway control systems from entering a dangerous state. Ladder logic will be explained in a later chapter. David Kerr and Tony Rowbotham produced a book that explains the terminology and methodology used in the railway industry and by Invensys (See [KR01]).

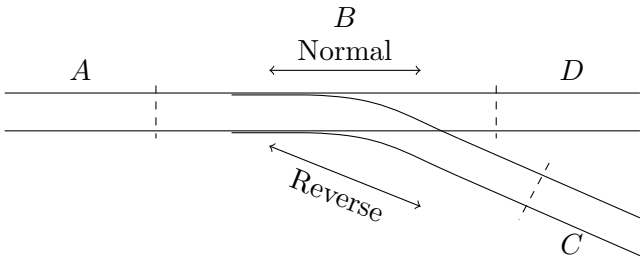


Figure 2.1: A Typical Junction

2.3 An Overview of the Railway Domain

In this section we present the features of the railway domain that are in the scope of this thesis. We hope to provide the reader with the background information and terminology necessary to understand the parts of this thesis.

2.3.1 The Railway Topology: Track and Points

We will now present an overview of the physical railway from a topological point of view. To do this we will present an example of a small track plan of a junction. If the reader is interested in learning more about the topology of the railway a more detailed description can be found in [KR01]

2.3.1.1 Track Segments

A section of track is typically broken down into track segments each containing one or more track circuits to detect the presence of a train. Typically track segments become larger on long straight stretches of track without any interesting topological features such as junctions or stations. Likewise track segments become smaller around junctions and stations where control over train movement is of greater importance.

2.3.1.2 Points

A point is a physical piece of equipment that is used to form a junction. Due to the nature of the rails and trains it is not possible to physically to just join two segments of track. Instead a point is needed to act as physical switch controlling the flow of trains through a junction. A point has two positions which are referred to as **normal** and **reverse**. This presents a safety hazard, for example see figure 2.1, if a train enters the junction b from c when the junction is locked in the position for normal then the train will be derailed.

2.3.2 Railway Signalling

Signals are the main means used to communicate information regarding the state of the track ahead of the train. Typically they are placed either on the track side or over hanging the railway. Visual indications known as aspects are used to convey information to the driver. A signal will have many such aspects which can be displayed, each with a particular meaning. The main type of signal considered in this project is the coloured light signal. Typically these have between one - four aspects each conveying a different indication about the state of the track ahead. Below is a description of the aspects used for a three light signal.

Green - If this aspect is displayed it indicates that track ahead is clear for a sufficient distance and the train driver can proceed at full speed to the next signal.

Yellow - This aspect indicates the track immediately ahead in between this signal and the next is clear however the driver should proceed with caution as a train could be in the track after that.

Red - This aspect indicates that the track ahead is not clear, the driver should stop and wait at this signal.

The one aspect signal is typically a fixed red indicating that is not possible to proceed down the track at this current point in time. The two - four aspect signals are used on tracks with different speeds to convey different stopping distances. The two aspect signal for instance would be used on a low speed track segment where stopping distances are relatively short. Whereas the four aspect signal would be used on a high speed line where stopping distances are long and the driver needs information for a greater length of track. These signalling schemes are fixed in the UK however they are not fixed from country to country. On the continent, for example, they may use different conventions, colours and number of lights on each signal.

2.3.3 The Westrace Interlocking

The railway interlocking is a key component in ensuring the safety of the railway. Its job is to apply a set of rules to the requests and commands it receives from the control system and check whether or not the future state of the railway is safe. If the control signals it receives do not violate the safety of the railway then these signals are committed to the physical infrastructure. For example if the human controller requests for a route to be set the interlocking will process this request and ensure that it does not conflict with other routes before allowing the command to be passed to the physical railway.

The railway interlocking repeatedly executes a program or set of rules over some discrete time interval. Each time it uses the set of rules it contains to process a new set of inputs before committing them as outputs. The Westrace interlocking used by Invensys Rail executes a so-called ladder logic program to perform this process. The following are the three main stages of operation in the running of an Westrace interlocking.

Reading of Inputs - Read inputs from the control systems as well as the physical railway infrastructure.

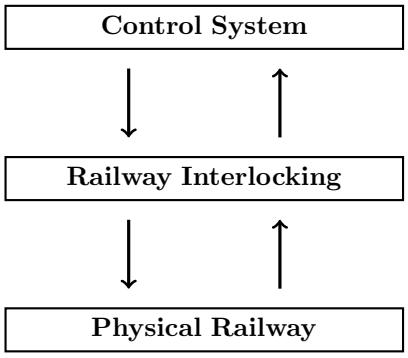


Figure 2.2: The Location of the Railway Interlocking

Internal Processing - Execute the ladder logic program with the above inputs and calculate outputs.

Committing of Outputs - The outputs calculated in the previous cycle are then passed on to various places including the physical railway.

2.4 Ladder Logic

The Westrace Interlocking performs calculations by executing a ladder logic program [61103]. In the following we will look in more detail at these ladder logic programs. The main concepts behind their construction and behaviour will be presented. In later chapters we will provide a formal framework for the verification of these programs.

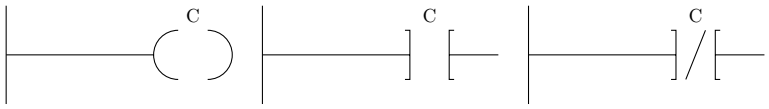
The international standard for programmable logic controllers IEC 61131 [61103] describes the graphical language ladder logic. It gets its name from its graphical "ladder" like appearance which was chosen to suit the control engineers responsible for their design. Each rung of the ladder is used to compute an output variable from one or more input variables in the rung. In the railway industry these input variables are referred to as contacts and the output variables are referred to as coils. A description of the entities representing these variables is as follows:

Coils : These are used to represent values that are both stored for later use and output from the program. The value of a coil is calculated when a rung fires making use of the current set of inputs, the previous set of outputs and any outputs already computed for this cycle. The coil is always the right most entity of the rung and its value is computed by executing the rung from left to right.

Open Contacts : This entity represents the value of an un-negated variable

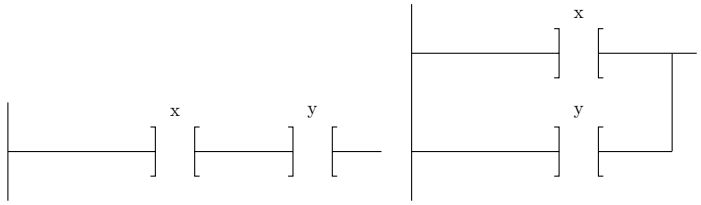
Closed Contacts : This entity represents the value of a negated variable.

A Ladder logic rung is built using these entities and connections between them. The shapes of the connections between the contacts determines how the value of the coil is computed from them. Using propositional logic for comparison, a horizontal connection between two contacts represents logical conjunction and a vertical connection between two contacts represents logical disjunction see Figure 2.4.



(a) A coil (b) An open contact (c) A closed contact

Figure 2.3: The Entities Used In Ladder Logic



(a) $x \wedge y$ Conjunction (b) $x \vee y$ Disjunction

Figure 2.4: Logical Connectives In Ladder Logic

In section 4 an approach is presented to capture the semantics of ladder logic programs using propositional logic.

2.5 Previous Work in this Field

Previously James carried out work for Invensys, applying various SAT and model checking techniques to verify the correctness of a simple pelican crossing and two existing railway interlockings consisting of approximately 500 rungs (see James [Jam10]).

James used Kanso's work (See [Kan08]), in particular his translation from ladder logic into propositional logic, and applied several model checking techniques in order to try and reduce the complexity of the problems. Both the work by Kanso and James was based on an early feasibility study by Fokkink and Hollingshead [FH98]. The relationship between a ladder logic program and propositional logic was discussed in great detail. A method for formulating such a ladder logic program as a formula in propositional logic was presented. This laid the ground work for all successive projects involving ladder logic. The possible application of program slicing was discussed and this was later applied in the work by James [Jam10].

Some of the techniques applied by James to the verification of ladder logic are discussed below.

Bounded model checking: This was the main topic of the work of Phil James. It had the advantage that it produced counter example traces which are highly valuable to the engineers at Invensys. It allowed for the verification of 2000 iterations of the ladder logic programs provided without programming slicing and up to 20000 iterations of ladder logic programs with program slicing.

Temporal Induction: This is another technique used in the verification of the ladder logic programs, it succeeded whenever the inductive verification method Kanso applied also succeeded. It should however be stronger than inductive verification but no example

was found to prove this. Temporal induction produced a counter example whenever the bounded model checking produced a counter example.

Program Slicing: This technique was combined with application of bounded model checking to reduce the state space requiring verification. This reduced the number of rungs in a ladder logic program by up to a factor of 10.

Chapter 3

Model Checking

In this chapter we will discuss the concept of model checking and how model checking techniques can be used in the verification of safety critical systems. Using model checking has the advantage over deductive methods[1] of verification that it is automatic. However not every system or property can be verified by this method due to the limits of what is decidable / computable. One further advantage is that a model checking algorithm will typically provide a *counter example trace*. These traces show us the exact sequence of states which brought about an error. This property is useful to our industrial partners because it can be used by engineers to find faults in the system. One early model checking procedure was presented in the work by Clarke [CES86] which verified models formalised in CTL. One the most well known problems in the field of model checking is *state space explosion* which also studied by Clarke and McMillian [CGJ+01, BCM+92]. The number of states in a model of a concurrent system increases exponentially in relation to the increase in number of components in the system. In 2008 Pelnek published a review of the various advances made in this area [Pel08]. The main focus of the research effort so far has been to reduce the number of states in our model. This has been achieved through the use of state based reduction, path based reduction and compositional reasoning. A simple form of state based reduction would be to remove a state from the search space if it is bisimilar to another state already encountered. Other methods include program slicing which simplifies the model with respect to the safety property being verified. The model formed only contains information about the system that is relevant to the safety property. Progress has also been made in reducing the amount of physical storage needed for a model using caching and compression. A number of approaches that make use of randomisation and heuristics are also presented. While these can not prove an entire system correct they are useful for finding errors.

3.1 Applying Model Checking Techniques to Safety Critical Systems

Currently there are two main approaches to the problem of verifying the correctness of safety critical systems namely model checking (See e.g. [ADS+06, Jam10, CGP99]) and theorem

[1] Restricted forms of deduction can be automated but in general this is not the case.

proving (See e.g.[M.S01, BC04]). For the purpose of this document we are concerned with model checking which is the approach most widely used by industry. The model checking problem is as follows. Given a model and formula expressing some property we would like the model to have, under what conditions does the model satisfy the formula. The two main techniques typically used in model checking are SAT-based model checking and binary decision diagrams. I will begin by discussing the former, however in order to discuss model checking we first need several definitions that underlie it.

3.1.1 Propositional Logic and Transition Systems

One must choose a formal mathematical language in order to express the properties of a given system and reason about them. Building on the work of previous projects [Jam10, Kan08, FH98]I have chosen propositional logic (See [Mar09]) for this purpose.

Definition 1 (Syntax of Propositional Logic)**.** *Given a set of propositional variables Var, we define the set of propositional formulas $Form_{Var}$ to be the least set satisfying the following conditions:*

- $\bot \in Form_{Var}$
- $\top \in Form_{Var}$
- *If $a \in Var$, then $a \in Form_{Var}$*
- *If $\phi_1, \phi_2 \in Form_{Var}$, then $\neg\phi_1 \in Form_{Var}$, $\phi_1 \wedge \phi_2 \in Form_{Var}$, $\phi_1 \vee \phi_2 \in Form_{Var}$, $\phi_1 \rightarrow \phi_2 \in Form_{Var}$, and $\phi_1 \equiv \phi_2 \in Form_{Var}$*

If we view propositional logic as a language then our alphabet consists of the following $Var \cup \{\bot, \top, \wedge, \vee, \neg, \rightarrow, \equiv\}$ where p is a symbol representing a variable in Var and our set of strings is the set $Form_{Var}$.

Definition 2 (Valuation)**.** *A valuation is a function $v : Var \rightarrow \{0, 1\}$ which maps each variable in $x \in Var$ to a value in the set $\{0, 1\}$.*

Definition 3 (Satisfaction Relation $v \models \phi$)**.**

- $v \models \top$, $v \not\models \bot$
- $v \models p$ *if* p *is a variable and* $v(p) = 1$
- $v \models \neg\phi$ *if* $v \not\models \phi$
- $v \models \phi \wedge \psi$ *if* $v \models \phi$ *and* $v \models \psi$
- $v \models \phi \vee \psi$ *if* $v \models \phi$ *or* $v \models \psi$
- $v \models \phi \rightarrow \psi$ *if* $v \not\models \phi$ *or* $v \models \psi$
- $v \models \phi \equiv \psi$ *if* $(v \models \phi$ *and* $v \models \psi)$ *or* $(v \not\models \phi$ *and* $v \not\models \psi)$

In order to reason about systems we need a formal notion of a model. One such notion called a state transition system is presented in [ADS$^+$06]. This allows us to capture the properties of each state and the relationship between them. Another formalism used to capture models

is the Kripke structure which is presented in [CGP99]. The Kripke structure expands upon the definition of a transition system by adding a label to each state which indicates whether certain propositional formulas are true in a given state. In this thesis we have chosen not use the Kripke structures as they are more suited to capturing the semantics of temporal logics.

Definition 4 (Transition System)**.** *A transition system M is defined to be a three tuple (S, S_0, R), where*

- *S is a finite set of states.*
- *$S_0 \subseteq S$ is the set of initial states.*
- *$R \subseteq S \times S$ is a total transition relation.*

We also need to be able to reason about reachability in our transition system. That is given one state what other states can be reached from this state given the transition relation. In order to decide whether or not a given transition system M is safe we need to we need to define what it means for a state to be reachable from the initial state. This will allow us to check that all of the states reachable from the initial state are safe which means that they are a subset of states that hold under the safety condition.

Definition 5 (Reachability)**.** *Given a transition system $M = (S, S_0, R)$, we define $Reach_R(S)$ to be the set of states that are reachable from S using the transition relation R. That is $Reach_R(S)$ is the least set of states such that*[2]*:*

$$Reach_R(S) = \{s \in S | s = S_0 \vee R(r, s) \wedge r \in Reach_R(S)\}$$

We need to be able to reason whether or not a safety condition holds in a transition system structure. Therefore we define a safety condition P to be a set of states that we consider safe and acceptable. Consider a typical microwave oven with a door and an on or off switch which controls cooking. The set of safe states are those in which the door is closed or the oven is off. If the oven could start cooking with the door open it would be unsafe and could potentially be dangerous. The system should be constructed in such a way that this state is unreachable.

We can define a system to be safe if all of the reachable states in a system satisfy the safety condition.

Definition 6 (Model Safety)**.** *We say that a transition system $M = (S, S_0, R)$ is safe with respect to a safe set of states P if the following holds.*

$$Reach_R(S_0) \subseteq P$$

3.1.2 Model Checking Methods

There are several model checking methods in use today of which we will now briefly describe one. The main technique underlying the model checking component of the *SCADE* suite is called SAT-based model checking [SSS00, BCC$^+$99, ADK$^+$05]. SAT-based model checking

[2] In other words $Reach_R(S)$ is inductively defined

works by formulating a model checking problem in propositional logic and then passing that representation of the problem to a SAT-solver (see [CESS08]) which then attempts to satisfy the propositional formula. When translating the above formulae to a format that we can input into a SAT-solver we replace the explicitly stated sets symbolically with predicates.

We need a predicate to represent in propositional logic that a state s is in a set of states S. We define a predicate $S(s)$ such that:

$$s \in S \leftrightarrow S(s)$$

We also need to define a predicate that allows us to represent a sequence of states linked by the transition relation. We shall call such a sequence of states a path through our transition system.

Definition 7 (Path Predicate)**.** *The path predicate is said to hold over a sequence of states $s_0, ..., s_n$ if for every $i \in \{0, ..., n-1\}$ the transition relation holds in the corresponding state and the successive state in the sequence.*

$$path(s_0, ..., s_n) = \forall i \in \{0, ..., n-1\} : R(s_i, s_{i+1})$$

In order to formulate our model checking problem, into a format which is accepted by a SAT-Solver, we must first represent in first order logic what it means for a transition system to be safe. This is important because we want to be able to show that all paths from the initial state lead to a state in which a predicate P representing a safety property holds.

$$\forall n \geq 0 : \forall s_0 ... s_n : path(s_0, ..., s_n) \wedge S_0(s_0) \rightarrow P(s_n)$$

Lemma 1. *It is the case that a state s is reachable in a transition system $M = (S, S_0, R)$ only if there is a path from one of the initial states S_0 which ends with s.*

$$s \in Reach_R(S) \Leftrightarrow \exists path(s_0, ..., s_n) \wedge s_0 \in S_0 \wedge s_n = s$$

There are many different methods to solve the model checking problem using a SAT-solver. The first we shall discuss is called bounded model checking. This technique was devised by Clarke [CBRZ01] to solve model checking problems without the use of binary decision diagrams. This is performed using the above formula by instantiating the universal quantifier with an increasingly large n and then passing the formula to a SAT-solver to prove until a predefined bound on the search depth is reached or the formula is found to be falsifiable. This allows systems, with large state spaces that cannot be fully traversed, to have some form of verification applied to them. This presents a problem in that we do not know if the entire state space has been verified. We only know that all states that are connected by a path of length n or less to the starting state have been verified.

Definition 8 (Bounded Model Checking)**.** *Bounded model checking is defined by the following formula.*

$$bmc_n(s_0, ..., s_n) = path(s_0, ..., s_n) \wedge S_0(s_0) \rightarrow P(s_n)$$

Another SAT-Based model checking method allows us to go beyond just proving that the system is safe for n steps and allows us to prove that it is safe in its entirety, namely induction over time. However we will not cover this in this work; for more information see James [Jam10]. Moving on from general model checking techniques we will now study, in some depth, two approaches to deciding the validity of propositional formulas.

3.2 Stålmarck's Algorithm

One of the techniques behind the *SCADE* suite is a patented tautology checker known as Stålmarck's algorithm [SS00, Stå94]. It has been used successfully to verify railway interlockings in using an earlier incarnation of *SCADE* by Prover Technologies called NP-Tools [Bor98] and in an un-named tool [GvVK95]. The wide spread usage of the algorithm has caused it to come under much scrutiny by both academia and industry. This has led to the correctness of the algorithm being explored. In one case study the algorithm was verified in the interactive theorem prover Coq by Letouzey [LT00]. Such scrutiny has given it the reputation for reliability and it is trusted by industry for use in safety critical applications. More recently the algorithm has been extended to first order logic [Bjö05, Bjö09] which is just one of many improvements and developments in its history. In the following we will try to provide the reader with an understanding of how Stålmarck's algorithm works. This algorithm was influenced by two well established proof systems. The cut elimination principle of Gentzen's Sequent Calculus [Bus98] inspired a branch and merge operation called the Dilemma Rule. The semantic tableaux of Smullyan [Smu69] inspired the proof rules. We will begin by presenting the necessary theoretical background and then proceed to discuss the algorithm in detail.

3.2.1 Introduction

Stålmarck's algorithm makes use of equivalence classes through the use of a data structure known as a triplet. A set of simple rules is then applied to these triplets in order to deduce new equations from existing ones. If these rules become exhausted and no contradiction is found, then the algorithm applies the so-called Dilemma rule. The application of the Dilemma rule to our proof search causes it to branch into two derivations the conclusions of which are then merged. I will now provide a brief overview of the algorithm so that the reader is able to understand these concepts before moving on to a more detailed description of the algorithm. When performing examples we will restrict ourselves to formulas built only from negation and implication. The complete proof system can handle propositional formulas built from all of the standard connectives ($\wedge, \vee, \Rightarrow, \neg, =, \equiv, ...$).

We begin by transforming any propositional logic formula into a formula containing only propositional variables, implication operator and $\bot$ (false). This is performed by repeatedly applying the meaning preserving transformations that follow.

$$A \vee B \text{ to } \neg A \to B$$
$$A \wedge B \text{ to } \neg(A \to \neg B)$$
$$\neg\neg A \text{ to } A$$
$$\neg A \text{ to } A \to \bot$$

Now that we have a formula built from only one propositional operator, we can translate it into the triplets. These represent sub-formulas using so-called triplet variables. Given an implication formula A consisting of variables $a_1, \ldots, a_n$, and compound sub-formulas $B_1, \ldots, B_k$ where the sub-formula B_k is A, every sub-formula B_i is of the form $C_i \to D_i$, C_i and D_i are sub-formulas of A and occur amongst $B_1, \ldots, B_{i-1}$ or $a_1, \ldots, a_n$. We introduce new variables $b_1, \ldots, b_k$ to represent each of the compound sub-formulas these are called *triplet variables*. When a triplet variable b_i represents a formula B_i we write $rep(B_i) = b_i$, propositional variables represent themselves i.e. $rep(a_i) = a_i$. Using this notation it is now possible to represent the formula A as the following set of *triplets*

$$\{(b_1, rep(C_1), rep(D_1)), \ldots, (b_k, rep(C_k), rep(D_k))\}$$

The notation (x, y, z) abbreviates the triplet formula $x \leftrightarrow (y \to z)$. Falsity and truth are treated as a special class of variable, we write $\bot$ as 0 and $\top$ as 1.
Example 1. *The formula* $((p \to q) \to p) \to q$ *becomes*

$$(b_1, p, q)$$
$$(b_2, b_1, p)$$
$$(b_3, b_2, q)$$

where b_3 *is the triplet variable representing the original formula and* b_2 *represents* $(p \to q) \to p$ *and* b_1 *represents* $p \to q$.

To check whether a formula is a tautology, we assume that it is false and then apply either *simple rules* or a branch and merge operation called the *dilemma rule*. Our final goal being to derive a contradiction from the formula. We will now look at proofs using the simple rules.

The simple rules take a triplet that matches its premises and its conclusions provide us with fresh information allowing us to derive a new set of triplets. Consider the following triplet rule

$$\text{(r1)}\ \frac{(0, y, z)}{y/1\ /\ z/0} \tag{3.1}$$

by the definition of the $\to$ operator, if formula $y \to z$ is false, then we know that y is true and z is false. We call triplets that are contradictory *terminal*. The terminal triplets are $(1, 1, 0)$, $(0, x, 1)$ and $(0, 0, x)$. Taking the triplet $(0, x, 1)$ as an example, we know that the

formula $x \to 1$ is a tautology, the whole triplet therefore is a contradiction. We have devised the following example to demonstrate this approach.

Example 2. *The following triplets represent the formula* $(p \to q) \to p \to q$

$$
\begin{array}{l}
(b_1, p, q) \\
(b_2, p, q) \\
(b_3, b_1, b_2)
\end{array}
$$

We assume that the formula is false i.e. $b_3/0$. *We then apply the simple rule* $r1$ *to the triplet that results from that assumption. That triplet is then dropped from the set.*

$$
\begin{array}{l}
(b_1, p, q) \\
(b_2, p, q) \\
(0, b_1, b_2) \implies \\
(b_1, p, q)[b_1/1, b_2/0] \\
(b_2, p, q)[b_1/1, b_2/0] \implies \\
(1, p, q) \\
(0, p, q)
\end{array}
$$

To complete the proof we apply the rule $r1$ *again to the triplet* $(0, p, q)$.

$$
\begin{array}{l}
(1, p, q) \\
(0, p, q) \implies \\
(1, p, q)[p/1, q/0] \implies \\
(1, 1, 0)
\end{array}
$$

The resulting triplet $(1, 1, 0)$ *is terminal. Since we assumed that our original formula* $(p \to q) \to p \to q$ *is false and have shown that this assumption produces a contradiction, we can conclude that the formula is a tautology.*

There are six remaining simple rules which we have not used in this example:

$$
(r2)\ \frac{(x, y, 1)}{x/1} \quad (r3)\ \frac{(x, 0, z)}{x/1}
$$

$$
(r4)\ \frac{(x, 1, z)}{x/z} \quad (r5)\ \frac{(x, y, 0)}{x/\neg y}
$$

$$(r6)\ \frac{(x,x,z)}{x/1\ z/1} \quad (r7)\ \frac{(x,y,y)}{x/1}$$

We use $\neg y$ to represent the negated value of the boolean variable. The simple rules by themselves do not yield a complete proof system. A form of branching is required for completeness.

3.2.2 Dilemma Rule

The branching required for completeness comes in the form of the dilemma rule. Applying this to a set of triplets T causes two branches to form with a variable x assumed to be true in one and false in the other. The two sets of triplets $T[x/1]$ and $T[x/0]$ are the starting point for the derivations D_1 and D_2 respectively. The output of the merge operation depends on whether the resulting set of triplets for each derivation contains a terminal triplet. If D_1 produces a terminal triplet, then the result of the merge operation is the outcome of D_2 and vice versa. If neither of the sets contain a terminal triplet, then what results is a variable instantiation S which is the intersection of the triplets contained in S_1 and S_2. The value of x does not affect the conclusions, they are independent of the value of x, therefore they are collected by this intersection. The following example was taken from the work of Stålmarck [SS00].

Example 3. *We will now show that the following formula is a tautology*

$$(((p \to p) \to p) \to (p \to q)) \to (((p \to q) \to p) \to q)$$

We derive the following triplets from the formula

$$\begin{array}{l}(b_1, p, q)\\(b_2, b_1, p)\\(b_3, b_2, q)\\(b_4, p, q)\\(b_5, p, p)\\(b_6, b_5, p)\\(b_7, b_6, b_4)\\(b_8, b_7, b_3)\end{array}$$

The following shows the correspondence between the triplets and the sub-formulas.

$$(((p \xrightarrow{b_5} p) \xrightarrow{b_6} p) \xrightarrow{b_7} (p \xrightarrow{b_4} q)) \xrightarrow{b_8} (((p \xrightarrow{b_1} q) \xrightarrow{b_2} p) \xrightarrow{b_3} q)$$

As with the previous example we start our proof by assuming that the formula, which corresponds to the triplet b_8, is false. We then proceed to apply the rules repeatedly and show a

contradiction. The proof derivation starts with the application of the rule $r1$ *to the triplet for* b_8

$$
\begin{array}{l}
(b_1, p, q) \\
(b_2, b_1, p) \\
(0, b_2, q) \\
(b_4, p, q) \\
(b_5, p, p) \\
(b_6, b_5, p) \\
(1, b_6, b_4) \\
[b_7/1, b_3/0]
\end{array}
$$

We then apply the rule $r1$ *again this time to the triplet* $(0, b_2, q)$*:*

$$
\begin{array}{l}
(b_1, p, 0) \\
(1, b_1, p) \\
[b_2/1, q/0] \\
(b_4, p, 0) \\
(b_5, p, p) \\
(b_6, b_5, p) \\
(1, b_6, b_4) \\
[b_7/1, b_3/0]
\end{array}
$$

This is followed by the application of $r7$ *to the triplet* (b_5, p, p) *and the rule* $r4$ *to the resulting triplet* $(b_6, 1, p)$*.*

$$
\begin{array}{l}
(b_1, p, 0) \\
(1, b_1, p) \\
[b_2/1, q/0] \\
(b_4, p, 0) \\
[b_5/1] \\
[b_6/p] \\
(1, p, b_4) \\
[b_7/1, b_3/0]
\end{array}
$$

The final simple rule we are going to apply is $r5$ *to each of the triplets of the form* $(x, y, 0)$.

$$
\begin{array}{l}
(b_1, p, 0) \\
(1, b_1, p) \\
[b_2/1, q/0] \\
(b_4, p, 0) \\
[b_5/1] \\
[b_6/p] \\
(1, p, b_4) \\
[b_7/1, b_3/0]
\end{array}
$$

Following the above derivation it is no longer possible for us to apply any simple rules. We are left with only two triplets: $(1, \neg p, p)$ and $(1, p, \neg p)$. The only option left for us is to apply the dilemma rule. We branch on p, assuming it is true in one branch and false in the other. Both branches contain the triplets $(1, 0, 1)$ and $(1, 1, 0)$. The triplet $(1, 1, 0)$ is terminal and therefore results in a contradiction being reached in both branches. Therefore a contradiction can be derived from our original assumption that the formula was false. We can conclude that this is not the case and that this formula is indeed a tautology.

3.3 Underlying Theory

We will now discuss the design of an efficient algorithm for finding derivations in the proof system described in the previous section. The algorithm is known as "Stålmarck's saturation method". In order to get a better understanding of how a tautology checker such as Stålmarck's algorithm functions in a correct and efficient manner one might look at some of the proof systems which underlie it. One would investigate Gentzen's Sequent Calculus PK which can be found in Appendix A.1. However if the reader is interested in a more detailed discussion of the Sequent calculus it can be found in a book by Buss [Bus98].

There is one more classic proof method that is needed in order to understand the underlying theorem behind Stålmarcks algorithm. Smullyan's analytic tableaux [Smu69] system works by analysing cases where a valuation was assigned to a formula and its effect on the valuation of its sub-formulae, see appendix A.2. It is possible for this method to repeat a search in a part of the search space it has already explored. This is due to the fact it does not retain any information about the search space it has already explored. Stålmarck saw it necessary to introduce a set of rules that not only took into account the valuation of a formula but also the valuations of its immediate sub-formulae and their complements. It was also beneficial to capture whether or not two formulas have the same truth value. Stålmarck's algorithm works on equivalence relations of formulas. The domain of the equivalence relations is the set of subformulas and their negations of a given formula X that one tries to refute. The triplets presented in Sect. 3.2 are used to denote representations of equivalences classes of these relations. This brings us to the set of rules which underlie Stålmarck's proof system.

Definition 9 (Proper Rules)**.** *Given a rule for a connective $\circ$ in the following form:*

$$\frac{F_1 \equiv G_1, \ldots, F_n \equiv G_n}{F \equiv G}$$

Where $\equiv$ is logical equivalence and with every $F_i, G_i \in \{A, B, A \circ B, \neg A, \neg B, \top, \bot\}$. The rule is proper if and only if it satisfies the following properties:

$$\{F_1 \equiv G_1, \ldots, F_n \equiv G_n\} \models F \equiv G \tag{3.2a}$$

$$\{F_1 \equiv G_1, \ldots, F_n \equiv G_n\} \not\models \bot \tag{3.2b}$$

$$\{F_1 \equiv G_1, \ldots, F_n \equiv G_n\} - \{F_i \equiv G_i\} \not\models F \equiv G \tag{3.2c}$$

An in depth discussion of these proper rules and the proof system can be found in the work by Jakob Nordström [Nor01]. We have included the proper rules presented in the work by Nordström in appendix A.3.

These rules alone do not give a complete system for propositional logic. In order to achieve this we put a form of branching back into our system whilst retaining the sub-formula property. A proof system obeys the sub-formula property if every proof that can be made using the proof system is constructed using sub-formulas of the formulas to be proved. The following principle of bivalence was part of the proof system KE/I introduced by Mondadori and studied in detailed by D'Agostino [D90].

Definition 10 (Principle of Bivalence[3])**.**

$$(PB)\ \frac{}{A \equiv \top | A \equiv \bot}$$

A combination of the principle of bivalence and the proper rules yields a system that is complete for propositional logic. A proof system is said to be complete for propositional logic if given a tautological formula, the proof system can derive that it is in fact a tautology for more information see [Bus98]. We now need to be able to reason about what it means for formulas to have the same value. This can be achieved through the use of formula relations and equivalences classes.

Definition 11 (Formula Complement)**.** *We write the complement of a formula A as A'. If $A = \neg B$ then A' is B and if this is not the case then $A' = \neg A$.*

Definition 12 (Formula Relation)**.** *We define a formula relation $\sim$ to be an equivalence on a formula X . The domain of the equivalence relation is $S(X)$, the set containing all of the subformulas of X (including $\top$) and the negations of these subformulas. it is closed under following constraint; if $A \sim B$ then $A' \sim B'$*

We define $R(A \equiv B)$ as the least formula relation that relates A and B and contains R. We say that $A \equiv B$ is an association between A and B.

If we take an association $m : A \equiv B$ then the complementary association is $m' : A \equiv B'$. These two associations are contradictory. Having this property of associations allows us to

[3]The "|" in this definition means "or". Either A is true or A is false.

refute a formula. The identity relation on $S(X)$ is the smallest formula relation and is written X^+. We use the smallest formula relation to perform partial valuations of a formula X. The partial evaluation $X^+(X \equiv \top)$ is denoted by $X^\top$ and $X^+(X \equiv \bot)$ by $X^\bot$ We will now present a short example demonstrating how the simple rules can be applied to a partial evaluation.

Example 4. *If we take a propositional formula $X = \neg(C \vee D)$ where C and D are both propositional variables. Then we can take a partial evaluation of the formula:*

$$X^\top = \{[\neg(C \vee D), \top], [C], [D], [C \vee D, \bot], [\neg C], [\neg D]\}$$

The square brackets seen above are used to denote equivalence classes. We can then apply the following rules in sequence to the above set of equivalence classes.

$$\frac{P \vee Q \equiv \bot}{P \equiv \bot} \qquad \frac{P \vee Q \equiv \bot}{Q \equiv \bot} \qquad \frac{P \equiv \bot}{P' \equiv \top}$$

The application of the first rule and third rule results in R_1 the application of the second rule and third rule to R_1 results in R_2.

$$R_1 = \{[C \vee D, \bot, C], [D][\neg(C \vee D), \neg C, \top], [\neg D]\}$$

$$R_2 = \{[C \vee D, \bot, C, D], [\neg(C \vee D), \top, \neg C, \neg D]\}$$

Since we can not apply any futher rules to R_2 we say that it is a model of $\neg(C \vee D)$.

3.3.1 The Dilemma Rule

Since the simple rules alone do not give a system that is complete for propositional logic a form of branching is needed. This branching comes from the Dilemma rule which allows us to perform a branch and merge operation. A Dilemma derivation takes the following format.

$$\dfrac{\dfrac{\dfrac{R}{\dfrac{R(A \equiv B) \qquad R(A \equiv \neg B)}{\text{(derivation)} \qquad \text{(derivation)}}}}{\dfrac{R_1 \qquad R_2}{R_1 \sqcap R_2}}}$$

Starting with a formula relation R, we then choose two different non complementary equivalence classes in R and from each of these we take a formula namely A and B. We then form two new dilemma derivations using the formula relation with A associated with B, $R(A \equiv B)$ and where A is associated with $\neg B$, $R(A \equiv \neg B)$. The conclusion of these two derivations is R_1 and R_2 respectively. We take the intersection of R_1 and R_2 in order to collect the common conclusions of both derivations. We cease applying rules as soon as a derivation causes both a formula and its complement to be placed in the same equivalence class as this is explicity contradictory. We define $R_1 \sqcap R_2$ to be R_2 if a contradiction is reached in R_1, R_1

if a contradiction is reached in R_2. If neither of the previous two cases are applicable then $R_1 \sqcap R_2$ is $R_1 \bigcap R_2$, the intersection of the equivalence classes in R_1 and R_2.

Definition 13 (The Dilemma Derivations)**.** *We will now define the notion of a dilemma derivation.*

Simple Rules. If we apply one of the simple rules to an equivalence class R_1 and the result is R_2, then we define Π to be a Dilemma derivation of R_2 from R_1 i.e. $\Pi = R_1 R_2$. This is written as the assertion $\Pi : R_1 \implies R_2$. If we are in the case where it is not possible to apply a simple rule to R then R itself is defined to be a derivation of R from R. The proof depth for both of these cases $d(\Pi) = 0$.

Composition. It is possible to compose two proofs. If $\Pi_1 : R_1 \implies R_2$ and $\Pi_2 : R_2 \implies R_3$ then after composition they become $\Pi_1\Pi_2 : R_1 \implies R_3$. This has the advantage that the composed proof only contains one copy of R_2 whereas the two separate proofs each have their own copy of the relation. The depth of the proof formed by the composition is defined to be $d(\Pi_1, \Pi_2) = max(d(\Pi_1), d(\Pi_2))$.

Dilemma Rule. If $\Pi_1 : R(A \equiv B) \implies R_1$ and $\Pi_2 : R(A \equiv \neg B) \implies R_2$, then an application of the dilemma rule takes the following form:

$$\dfrac{\dfrac{R}{\Pi_1 \quad \Pi_2}}{R_1 \sqcap R_2}$$

We call the maximum number of branches open simulataneously the depth of the proof. This is a derivation of $R_1 \sqcap R_2$ from R which has a depth $max(d(\Pi_1), d(\Pi_2)) + 1$.

Remark 1 (Proof Hardness)**.** *We say that a formula relation R with a derivation $\Pi : R \implies \perp$ and $d(\Pi) \leq k$ is k-easy. Similarly we say that a formula relation R for which there does not exist any derivation $\Pi : R \implies \perp$ with $d(\Pi) > k$ is k-hard.*

Definition 14 (Hardness Degree)**.** *A relation R that is both k-easy and k-hard for some k is said to have a hardness degree k .*

According to Stålmarck [SS00] most of the problems encountered in industry are either 1 or 2 hard. One of the explanations of this is that the large number of propagation rules cause proofs performed using the system to only have a few nested branches. Industrial systems that have been designed by one team for a single purpose can typically be modelled as a large number of easy formulae. If however one has several complicated systems that are running concurrently and interacting with one another they can produce behaviour that is hard to reason about formally.

3.3.2 Implementing the Algorithm

Definition 15 (Triplets)**.** *The data structure used in the implementation of the algorithm is the triplet. These triplets encode the information contained within our equivalence classes. The triplet representation of a formula X is obtained by adding extra triplet variables that represent compound subformulas of X. Triplets have the form $x : y \circ z$ where x is the triplet variable representing the subformula containing two literals y, z and a binary operator $\circ$. A literal is a real variable or triplet variable that is either negated or un-negated.*

Example 5. *The following is the triplet representation of the formula:*

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

t: $\top$

a: A

b: B

c: $a \to b$

d: $\neg b \to \neg a$

e: $c \wedge d$

f: $\neg c \wedge \neg d$

g: $f \vee e$

Definition 16 (Saturation Algorithm)**.** *We say that a relation R is k-saturated if and only if for every possible dilemma derivation $\Pi : R \implies S$ with $d(\Pi) \leq k$ it is the case that $R = S$ holds. This means that no new conclusions are derived from proofs of depth k or less. 0-saturation is performed by applying every possible propagation rule to the relation. It proceeds by picking a compound subformula from the relation and applying a rule. It then applies rules to triplets that contain variables affected by the application of the first rule. If no contradiction is reached some formula of branching must be applied and we proceed by applying $k+1$-saturation.*

```
saturate(R,0)0 =

Q := Compound(R)
while non-empty(Q)
  do
    remove some q from set Q
    if contradictory(R(q))
    then return R(q)
    else Q := Q union affected(R(q) - R)
         R := R(q)
  od
return R
```

The function `Compound(R)` *returns the set of compound subformulas in the domain of R. In order to keep the complexity of the algorithm to a minimum this is restricted to contain only one representitive subformula from each equivalence class. This is referred to as using indeterminate equivalence classes. At the start of the algorithm Q contains our initial collection of sub formulas. We then pick one element q from Q and apply a simple rule that will allow us to deduce some information from q. This is the point in the algorithm where one of the propagation rules is applied. The function $R(q)$ applies one propagation rule if any are applicable. A rule is said to be applicable in this situation if all of the formulas in the premise of*

the rule can be found in the set of formulas containing q. The result of $R(q)$ is the formula relation $R(A \equiv B)$ if an applicable rule can be found otherwise it is R. The application of a propagation rule produces a set of new equivalences. This set which we shall refer to as N is described using the set minus operation in the algorithm, $R(q) - R$. We now add to our pool the set of sub formulas that have been affected by the application of the rule. The formulas that may have been affected include all of those in N, all the sub formulas contained in an equivalence class or its complement for every formula in N and any formula that contains a formula in N as a sub formula. The following optimisation is made in order to lower the complexity of the algorithm. If a variable v is in one of the indeterminate equivalence classes then any other formulas that also contain v are only placed into our set of formulas to be processed if they contain a second variable in the same equivalence class as v or its compliment.

$(k+1)$-saturation is the embodiment of the dilemma rule. Branching is achieved by making two recursive calls to the k-saturation algorithm with the two complimentary formula relations.

```
saturate(R, k+1) =

repeat
  L := Sub(R)
  R' := R
  for each l in L
   do
     R1 := saturate(R(l equiv FALSE), k)
     R2 := saturate(R(l equiv TRUE), k)
     if contradictory(R1) and contradictory(R2)
        then return R1 union R2
        else if contradictory(R1)
             then R := R2
             else if contradictory(R2)
                  then R := R1
                  else R := R1 intersect R2


   od

until R' = R
return R
```

The function sub(R) returns the set of all the subformulas in R including variables and compound subformulas. In a similar fashion to 0-saturation only one formula from each of the indeterminate equivalence classes is placed into our set of formulas. The algorithm then iterates, picking formulas from this set, performing a dilemma derivation by making two recursive calls of the saturation algorithm. If both of the derivations are contradictory then the loop finishes and $R1 \cup R2$ is returned as the result of this function call. If one of the derivations is contradictory then that derivation is discarded and the formula relation produced by the remaining derivation is assigned to R. If neither of the formula relations produced are contradictory then R becomes the intersection of the two formula relations produced by the

derivations. This is repeated until no new equivalences can be derived by from our set of formulas. This differs from a breath first search or iterative deepening algorithm in that we are constantly gathering information in the form of equivalences. These equivalences one from level of saturation (k-saturation) can be used in the next level of saturation ($k+1$-saturation).

Given that we have given a detailed description of the working of Stålmarcks algorithm in both a theoretical and practical setting. We will now proceed to give a detailed description of another technique that is used in the Scade suite.

3.4 Binary Decision Diagrams

We will now discuss another technique used to decide the validity of propositional formulas called binary decision diagrams. These were originally introduced by Akers [Ake78] in order to provide an "implementation free" description of boolean functions. This is another of the techniques used in the verification engine of *SCADE*.

3.4.1 Introduction

In the early days of verification the state space of a system being modelled was expressed explicitly using data structures such as adjacency lists. While this approach was sufficient for the verification of small systems with only a few interacting components it ran into difficulties as systems became increasingly complex. Models of systems created using this apporach suffered from state space explosion and grew exponentially in size. One solution to this is symbolic model checking which was proposed by McMillian in his PhD Thesis [McM92]. The state space is represented implicity using Bryant's ordered binary decision diagrams (OBBDs) [Bry86]. A performance review of several BDD packages can be found in the work by Bryant [YBO+98]. While this review may be slightly out of date in terms of performance, it does present a comparison between the different types of BDD techniques available.

A BDD is a graph representing a boolean function and a form of symbolic model checking which represents state space implicitly rather than explicitly. This was one method for dealing with the state space explosion problem. Vertices represent boolean variables and transitions out of a vertex represent some assignment to that variable.

They offer the following benefits from a model checking perspective.

- We can convert any boolean function into a unique and canonical BDD representation.
- The logical operators $\wedge, \vee, \neg \ldots$ *etc* have a representation with complexity proportional to the product of the complexity of their inputs.

3.4.2 A Formal Introduction to BDDs

In the following we will present a more formal definition of binary decision diagrams which follows the work of Bryant [Bry86]. Bryant introduces the notion of a reduced ordered binary decision diagram (ROBDD) which combines ideas presented earlier by several different people

working in the field. Using these ROBBDs Bryant then discusses the possibilities for their practical use in verification. He then provides a concrete example of the verification of an arithmetic logic unit demonstrating this.

Definition 17 (Binary Decision Diagram)**.** *We define a* binary decision diagram *to be an acyclic and directed graph $G(V, E)$ with:*

- *A set of labelled vertices V*
  - *Terminal vertices that are labelled* 0 *or* 1
  - *Non-terminal vertices that are labelled by a variable $x_i \in X$*
- *A set of directed edges $E \supseteq V \times V$ that are one of the following*
  - *A "low-edge" labelled by 0*
  - *A "high-edge" labelled by 1*

These binary decision diagrams on their own do not respresent a canonical form for the representation of propositional formulae. We must restrict them to a subset which obey certain properties. The first of which we shall consider is an ordering on the variables.

Definition 18 (Ordered Binary Decision Diagram)**.** *An ordered binary decision diagram (OBDD) is defined by extending the above definition with an ordering on the variables.*

An ordering on the set of variables used to labelled vertices: $x_0 < x_1 < x_2, \ldots, < x_{n-1}$.

Figure 3.1 shows an example of a binary decision diagram of the following formula.

$$(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$$

This should allow the reader to visualise how a binary decision diagram would look like. The above formula can be written using boolean function notation as follows [Bry86]. I will use this notation in the context of BDDs.

$$x_1 \cdot x_2 + x_3 \cdot x_4$$

We are now ready to introduce an important sub-class of BDD, the canonical *Reduced Ordered BDDs*. These are widely used in model checking as they provide the most compact representation of a given boolean function and are one of the techniques employed by *SCADE*'s built in model checker.

Definition 19 (Reduced Ordered Binary Decision Diagram)**.** *A BDD is a Reduced Ordered BDD if it has the following properties.*

1. *The rank of any parent vertex is less than its two children.*

$$\underset{v \in V \setminus \{0,1\}}{\forall} rank(v) < min(rank(high(v)), rank(low(v)))$$

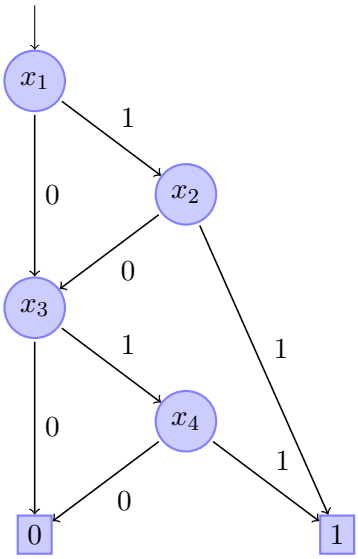


Figure 3.1: A Simple Binary Decision Diagram

2. *The children of internal vertices are distinct, that is for an internal vertex v, $high(v) \neq low(v)$*

3. *The BDD does not contain any isomorphic subgraphs.*

Requiring that the children of internal vertices are distinct removes nodes that are redundant and whose value has no effect on the evaluation of the function from this point on in the graph. The isomorphism condition removes subgraphs that repeat some property of the formula.

We will now provide a formal basis for reasoning about binary decision diagrams. To do this we represent propositional formulae as functions. We will use a function f which is of the type $f : B^n \to B$. The main operation we will be applying is to restrict an argument of a function to a certain value. This leads us to the following definition.

Definition 20 (Restriction)**.** *A restriction on the variable x_i in a function f to the value b is denoted by $f|_{x_i=b}$. For any arguments $x_1, \ldots, x_n$ a restriction is defined as:*

$$f|_{x_i=b}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n)$$

Using this restriction we can perform a "brute force" construction of a binary decision diagram. This is achieved through branching over each variable and restricting the subsequent function call.

Definition 21 (Shannon Expansion)**.** *Restriction upon a function allows us to define the* ***Shannon expansion*** *of a function around a variable x_i as the following:*

$$f = x_i \cdot f|_{x_i=1} + \overline{x}_i \cdot f|_{x_i=0}$$

The value of a function depends on specific input variables which can be represented using the following.

Definition 22 (Dependency Set)**.** *The dependency set of a function f is defined as*

$$I_f = \{i | f|_{x_i=0} \neq f|_{x_i=1}\}$$

We would like to be able to reason about the combinations of variables that cause a boolean function to evaluate to 1.

Definition 23 (Satisfying Set)**.** *The satisfying set of a function f is defined as follows:*

$$S_f = \{(x_1, \ldots, x_n) | f(x_1, \ldots, x_n) = 1\}$$

Since we are interested in the application of binary decision diagrams to real life industrial problems, it is necessary to take into account the efficiency of the technique we are applying. There are many different techniques for the optimisation of verification using binary decision diagrams. We will now consider one aspect of a binary decision diagram which has a substantial effect on it's graph size. That aspect is the ordering of the arguments in the boolean function being represented.

Consider the following two functions:

$$x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6 \tag{3.3}$$

$$x_1 \cdot x_4 + x_2 \cdot x_5 + x_3 \cdot x_6 \tag{3.4}$$

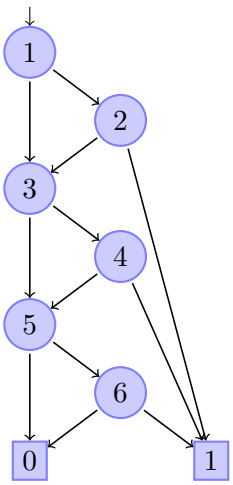


Figure 3.2: ROBDD of (3.3)

This can be generalised to $2n$ arguments.

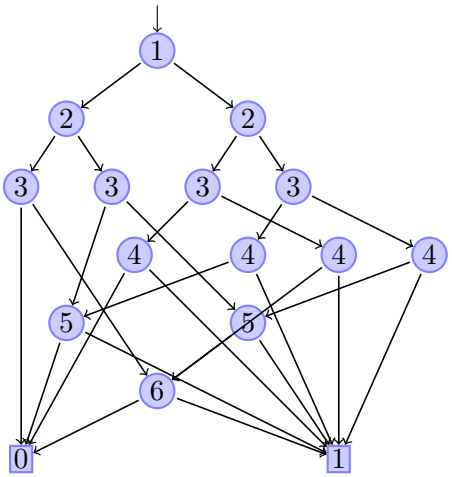


Figure 3.3: ROBDD of (3.4)

- $x_1 \cdot x_2 + \ldots + x_{2n-1} \cdot x_{2n}$ is represented by a graph with $2n + 2$ vertices
- $x_1 \cdot x_{n+1} + \ldots + x_n \cdot x_{2n}$ is represented by a graph with 2^{n+1} vertices

It is clear from Figures 3.2 and 3.3 that the ordering of the variables greatly affects the size of the resulting BDD. Once a variable order has been picked deciding whether a better ordering can be found is hard problem [BW96] and is known to be NP-complete [CLRS01]. There are quicker ways of finding a good variable ordering which typically involve using heuristics, however these are not guaranteed to produce results.

It is not always the case that we can find some ordering which allows us to represent a function using a "small" BDD. There are certain classes of function which have an inherent complexity. Functions in these classes have "large" BDDs irrespective of the ordering applied to the variables. One such class is the functions for binary integer multiplication.

Definition 24 (Binary Integer Multiplication)**.** *A binary integer multiplier is defined as having two binary words as inputs,* $a_1, \ldots, a_n$ *and* $b_1, \ldots, b_n$ *.The output of the integer multiplier is described by a function*

$$mul_i(a_1, \ldots, a_n, b_1, \ldots, b_n)$$

for all i *with* $1 \leq i \leq 2n$ *representing a binary encoding for the* $2n$ *output bits.*

We then take a permutation $\pi : \{1, \ldots, 2n\} \to \{1, \ldots, 2n\}$ of the numbers $1, \ldots, 2n$. The function $mul_i(x_{\pi(1)}, \ldots, x_{\pi(2n)})$ is denoted by a graph $G(i, \pi)$ with the permutation π being applied to the inputs $x_1, \ldots, x_{2n}$

Theorem 1. *For any* π *there exists an* i*,* $1 \leq i \leq 2n$*, such that* $G(i)$ *contains at least* $2^{n/8}$ *vertices.*

The proof of Theorem 1 makes use of 2 properties of binary multiplication namely commutativity and binary bit shifts which we will now explain.

Definition 25 (Commutativity)**.** *A binary relation* $*$ *on a set* S *is said to be commutative if it satisfies the following property.*

$$\forall x, y \in S : x * y = y * x$$

If one of the input words of the binary multiplication is a power of 2 then the multiplication algorithm acts as a bit shifter. If the word b represents a number 2^j then

$$[a \cdot b]_i = \begin{cases} a_{i-j} & \text{if } j < i \leq j + n \\ 0 & \text{otherwise} \end{cases}$$

We will now demonstrate a binary bit shift using a small example to give the reader an impression of the behaviour of this property.

Example 6 (Example Bit Shift)**.** *Consider the a binary encoding of the number* 2^3 *with* $j = 3$ $a = 00001000$ *and an arbitrary binary number* $b = x_8x_7x_6x_5x_4x_3x_2x_1$ *. If we multiply these two numbers together we obtain a third number* $c = a * b = x_8x_7x_6x_5x_4x_3x_2x_1000$*. c is the number b with a left shift of* 3 *applied. The output word c consists of j-many zeros with the first 5 bits of a appended them.*

We will now begin the proof of the inherent hardness of BDD size for binary multiplication.

Proof 1 (Theorem 1)**.** *We begin by defining a number t for a permutation* $\pi : \{1, \ldots, 2n\} \to \{1, \ldots, 2n\}$ *this allows us to consider all possible orderings of the input variables.*

Inorder to decide which input to use as our control we need to determine whether or not the permutation has caused each numbers input bits to migrate out of their half of the original input sequence. We define a number t to be the number of input bits from a which remain in their half of the input steam.

$$t = |\{\pi(j) | 1 \leq j \leq n, \pi(j) \leq n\}|$$

We define the sets F and L as follows:

If $t \geq \frac{n}{2}$ *then*

$$F = \{\pi(j) | 1 \leq j \leq n, \pi(j) \leq n\}$$

$$L = \{\pi(j) | n + 1 \leq j \leq 2n, \pi(j) > n\}$$

If $t < \frac{n}{2}$ *then*

$$F = \{\pi(j) | 1 \leq j \leq n, \pi(j) > n\}$$

$$L = \{\pi(j) | n + 1 \leq j \leq 2n, \pi(j) \leq n\}$$

For $1 \leq i \leq 2n$ *we define a set* F_i *as*

$$F_i = \{j | j \in F, \exists k \in L(j + k - (n+1) = i)\}$$

we set $q_i = |F_i|$

We then define a set of sequences:

$$S_i = \{x_1, \ldots x_n | \forall j \in \{1, \ldots, n\}, x_j = 0 \text{ if } \pi(j) \notin F_i\}$$

There are $n - q_i$ *variables in these sequences which are fixed.* S_i *therefore contains* 2^{q_i} *elements.*

Every sequence in S_i *leads to a unique vertex in* $G(i)$*.*

This is shown by a proof by contradiction.

We assume we are able to find two sequences $x_1, \ldots, x_n$ *and* $x'_1, \ldots, x'_n$ *such that for some* j*,* $\pi(j) \in F_i$ *and* $x_j \neq x'_j$ *(i.e.* $x_1, \ldots, x_n \neq x'_1 \ldots x'_n$*) that both lead to the same vertex in* $G(i)$

We then define two more sequences, $x_{n+1}, \ldots, x_{2n}$ *and* $x'_{n+1}, \ldots, x'_{2n}$ *such that:*

$$x_k = x'_k = \begin{cases} 1 & \text{if } \pi(j) + \pi(k) - (n+1) = i \\ 0 & \text{otherwise} \end{cases}$$

There is exactly one solution to the equation $\pi(j) + \pi(k) - (n+1) = i$ *and therefore one value of* k *for which* x_k *and* x'_k *equal* 1*.*

Since the sequences we just defined were the same, we know that both of the sequences $x_1, \ldots, x_{2n}$ *and* $x'_1, \ldots, x'_{2n}$ *end at the same terminal vertex in* $G(i)$*.*

This cannot be the case however since we know that the output functions are different.

$$mul_i(x_{\pi(1)}, \ldots, x_{\pi(2n)}) = x_j$$

$$mul_i(x'_{\pi(1)}, \ldots, x'_{\pi(2n)}) = x'_j \neq x_j$$

From this contradiction we can therefore conclude that $G(i)$ *must contain at least* 2^{q_i} *vertices.*

Hence it suffices to show that for at least one i *the number* q_i *has order of magnitude* n*.*

This is shown by the following lemma.

Lemma 2 (Counting Argument)**.** *If* $A, B \subseteq \{1, \ldots, n\}$ *each containing at least* $\frac{n}{2}$ *elements. For* $1 \leq i \leq 2n - 1$ *let*

$$q_i = |\{\langle a, b \rangle | a \in A, b \in B, a + b = i + 1\}|$$

Then there is some i *such that* $q_i \geq \frac{n}{8}$*.*

Proof of Lemma 1.

We have that the sets A and B both contain at least $\frac{n}{2}$ elements each.

Therefore using all the elements $a \in A$ and $b \in B$ we must be able to construct at least $\frac{n^2}{4}$ ordered pairs $\langle a, b \rangle$. Hence:

$$\sum_{j=1}^{2n-1} q_j \geq \frac{n^2}{4}$$

Since we know that some q_i must be at least as large as the average value of the $q_j's$ the following holds.

$$q_i \geq \frac{1}{2n-1} \cdot \frac{n^2}{4} \geq \frac{n}{8}$$

This completes our proof of the inherent complexity of binary multiplication. We have seen that there are certain classes of hard problems for which there is no "small" BDD representation. Despite this, a large number of the problems found in practice do have small representations, therefore BDDs are feasible for verification.

BDD Implementation

Moving on from our theoretical discussion we will now provide a brief discussion regarding the implementation of binary decision diagrams [Bry86]. We provide an overview of the algorithms used in the implementation of binary decision diagrams along with their time complexity.

The following are some procedures that can be implemented for binary decision diagrams.

Procedure	**Result**	**Time Complexity**
Reduce	G reduced to canonical form	$O(\|G\| \cdot log\|G\|)$
Apply	$f_1 \ \langle op \rangle \ f_2$	$O(\|G_1\| \cdot \|G_2\|)$
Restrict	$f\|_{x_i=b}$	$O(\|G\| \cdot log\|G\|)$
Compose	$f_1\|_{x_1=f_2}$	$O(\|G_1\|^2 \cdot \|G_2\|)$
Satisfy-one	some element of S_f	$O(n)$
Satisfy-all	S_f	$O(n \cdot \|S_f\|)$
Satisfy-count	$\|S_f\|$	$O(\|G\|)$

Figure 3.4: Binary Decision Diagram Procedures

In the following we briefly discuss the above procedures.

3.4.2.1 Reduction

The **Reduction** algorithm takes an arbitrary function graph as input and then transforms it into a reduced function graph of the same function.

- Vertices are tagged with labels starting from the terminal vertices and working back to the root. The label $id(v)$ for a vertex v is assigned such that $id(u) = id(v) \Leftrightarrow f_u = f_v$.
- The algorithm then produces a graph with one vertex for each unique label.
- This removes:
  1. Redundant vertices where $id(low(v)) = id(high(v))$.
  2. Isomorphic subgraphs, rooted by two distinct vertices v and u with $id(low(v)) = id(low(u))$ and $id(high(v)) = id(high(u))$.

3.4.2.2 Apply

The **Apply** operation is defined as follows:

$$[f_1 \langle op \rangle f_2](x_1, \ldots, x_n) = f_1(x_1, \ldots, x_n) \langle op \rangle f_2(x_1, \ldots, x_n)$$

The operator op is applied pointwise to the functions f_1 and f_2 and the set of input variables $x_1, \ldots, x_n$.

The Apply operation is implemented through a modification of the Shannon expansion.

$$f_1 \langle op \rangle f_2 = \overline{x_i} \cdot (f_1|_{x_i=0} \langle op \rangle f_2|_{x_i=0}) + x_i \cdot (f_1|_{x_i=1} \langle op \rangle f_2|_{x_i=1})$$

In order to apply the operator to the two graphs rooted at v_1 and v_2 we must consider the following cases.

- v_1 and v_2 are both terminal vertices. Then the result is a terminal vertex having the value $value(v_1) \langle op \rangle value(v_2)$.
- v_1 and v_2 are non terminal vertices with $index(v_1) = index(v_2) = i$. Then we create a new vertex with u with index i. This is followed by applying the procedure recursively to the subgraphs rooted by $low(v_1)$ and $low(v_2)$ to create a new subgraph with the root $low(u)$. a new subgraph $high(u)$ is created similarly.
- v_1 is a non terminal vertex with $index(v_1)$ but v_2 is either a terminal vertex or $index(v_2) > i$. The function f_2 corresponding to the graph rooted by v_2 is independent of x_i:

  $$f_2|_{x_i=0} = f_2|_{x_i=1} = f_2$$

  We therefore create a vertex u having index i then apply the algorithm to $low(v_1)$ and v_2 to create a subgraph who's root is $low(u)$. We then apply a similar approach to get a subgraph for $high(u)$.
- A similar approach is applied if the previous case is reversed.

3.4.2.3 Restriction

The procedure for **Restriction** applied to the variable x_i is fairly trivial. One replaces every vertex v with $index(v) = i$ with either $low(v)$ or $high(v)$ corresponding to the value of restriction.

3.4.2.4 Satisfy

These procedures are based on depth first traversal.

Satisfy-one This procedure returns false if there are no satisfying assignments, or true and an array containing the satisfying sequence which is satisfiable.

Satisfy-all The procedure is similar to satisfy one except it performs an exhaustive search and produces all of the satisfying sequences.

Satisfy-count This performs *Satisfy-all* and then counts the number of satisfying sequences returned by the procedure.

Conclusion

We have seen a selection of the techniques used by *SCADE* suite to perform verification. This began with looking into ways of encoding a model checking problem into propositional logic, approaches to do this included induction over time and bounded model checking. We then discussed in some detail two of the approaches used by the *SCADE* suite to decided the validity of propositional statements. Now that we have covered the theoretical background we are in a position to discuss the practical application of the *SCADE* suite to a verification problem.

Chapter 4

Verification of Ladder Logic Programs in SCADE

In the following we present an approach which allows the verification of ladder logic programs using the *SCADE* Suite. There have been several cases studies into the verification of safety critical embedded systems using the *SCADE* Suite . However the formalisation and verification of ladder logic programs has yet to be performed. Reliability analysis and verification has been performed on several industrial systems in [ADS+06] using techniques such as failure modes and effect analysis and fault tree analysis. In a much larger case study an avionics sensor voter has also been verified in *SCADE* [DBCB04]. It is clearly known that *SCADE* can verify large safety critical systems. The question we set out to answer is: can *SCADE* verify ladder logic programs?

We begin by discussing a method to capture the semantics of a ladder logic program using labelled transition systems. Previously an attempt was made by James [Jam10] to capture the semantics of a ladder logic program using an autonmaton. The advantages of our method are presented and discussed along side a comparison between the two methods. We then present a solution to the problem of how to translate ladder logic into the *SCADE* language, a dialect of Lustre. The problem of translating ladder logic into formal language was first studied in detail by Kanso [Kan08] and later extended by James [Jam10]. This work led to a tool which automatically translated ladder logic into the TPTP specification language. We built a modification of this tool to automatically translate ladder logic programs into the *SCADE* language before verifying them using the *SCADE* suite. We successfully apply this approach to a toy example before scaling it up to two real world railway interlockings. We then discuss techniques to add invariants to models using the *SCADE* suite, solving the problem of under-specification. Finally a comparison is performed between *SCADE*, the tool produced by James [Jam10] and the safety property verifier KIND. In the process we discuss one approach employed by KIND to verify Lustre programs.

4.1 Pelican Crossing

In the previous MRes projects by Kanso and James [Kan08, Jam10] that were centred around the verification of ladder logic programs Kanso presented a running example ladder logic program for a pelican crossing which was adopted by James. Pelican crossings are a common feature of road traffic networks in the UK and variations of these crossings exist throughout the world. Each crossing is a simple computer system that controls four sets of lights which in turn control the flow of cars and allow pedestrians a safe means to cross a road. We will now discuss the composition of a pelican crossing. In the example we will follow the naming convention used by James [Jam10] for the boolean variables representing the pelican crossing.

External Features of the Pelican Crossing

This is a description of how the pelican crossing is composed from an external view point. If one was a pedestrian and operated the crossing these are the features that would be noticeable.

Traffic Lights A pelican crossing has two traffic lights that control the flow of traffic in either direction. Each traffic light has a red and a green aspect indicating whether the flow of traffic should stop in the case of the former and that the traffic flow is clear to pass the crossing in the case of the latter. It is normally the case that one aspect is always shown, showing both aspects or no aspects would cause confusion as to whether the flow should stop or proceed. We will use 4 boolean variables to represent the lights, the names of these variables come from the possible combinations of two prefixes “pla” and “plb” representing the two traffic lights and two suffixes , “g” and “r” representing the possible aspects show on each light. This gives us a unique variable name for each aspect on each light.

Pedestrian Lights A pelican crossing also has two pedestrian lights that controls the flow of pedestrians across the road. These are positioned on either side of the road and face the opposing side of the road. As with the traffic lights these lights also have two aspects, a red aspect indicates that it is not safe for pedestrians to cross the road and a green aspect indicates that it should be safe for pedestrians to cross the road. Again as with the traffic lights it is normally the case that one aspect is always shown.

Audio As well as the disable aspects the pedestrian crossing also has an audio aspect to indicate to pedestrians whether or not it is safe to cross the road.

Pedestrian Buttons There are two pedestrian buttons on either side of the road that allow pedestrians to make a request to cross.

Internal Features of the Pelican Crossing

Our Pelican crossing has two internal variables “required” and “crossing” these allow us to model some internal states of the machine but where as the state of a light might be visible to a user these are not.

Crossing We require an internal variable that represents that a person is crossing the road. In the work by James a Boolean variable "crossing" is used we will follow that example and also use a Boolean variable with that name to represent this property.

Required We also require an internal variable that represents whether one of the pedestrian buttons has been pressed in a previous iteration of the program and there is a request to cross. Again we will follow the work by James and use a boolean variable "req" to represent this.

Ladder Logic Program for the Pelican Crossing

One possible formalisation for the above description of a pelican crossing in Ladder logic can be seen in Figure 4.1. Other formalisations are possible however we are will be following the work of James [Jam10] and use the same formalisation but with a slight modification. We have removed the coil representing the "audio" variable as this is simply a repetition of "plag" and "plbg".

Possible Formalisation in Propositional Logic

The above formalisation of the pelican crossing has an equivalent representation in propositional logic. This can be see in Figure 4.1.

Now that we have looked at representing a system using both ladder logic and propositional logic we will move on modelling ladder logic programs using propositional logic formulae.

4.2 Modelling Ladder Logic

The Westrace railway interlockings produced by Invensys Rail are programmed using ladder logic. Ladder logic is part of an international standard programming logic controllers IEC 61131 (see [61103]). Every ladder logic program has an equivalent representation in a subset of propositional logic that is called the ladder logic formulae. A translation from ladder logic into propositional logic was first performed by Kanso [Kan08]. Here we will restate a definition for these formulae is presented in the work by James [Jam10]. Given a propositional formula ϕ we define a function $vars(\phi)$ to return the set of propositional variables contained within ϕ. New variables are creating by applying a "prime" to current ones. We denote this set of new variables created from an old set of variables V with $V' = \{v' | v \in V\}$. We now need to define the sets of variables from which our ladder logic programs are formed. A ladder logic program is constructed in terms of a finite set of input variables I and a finite set of output variables C, with the property that $I \cap C = \emptyset$.

Definition 26 (Ladder Logic Formulae)**.** *A ladder logic formula ψ is a propositional formula composed from a set of input variables I and a set of output variables C that satisfies the following conditions:*

$$\psi \equiv ((c'_1 \leftrightarrow \psi_1) \wedge (c'_2 \leftrightarrow \psi_2) \wedge \ldots \wedge (c'_n \leftrightarrow \psi_n)$$

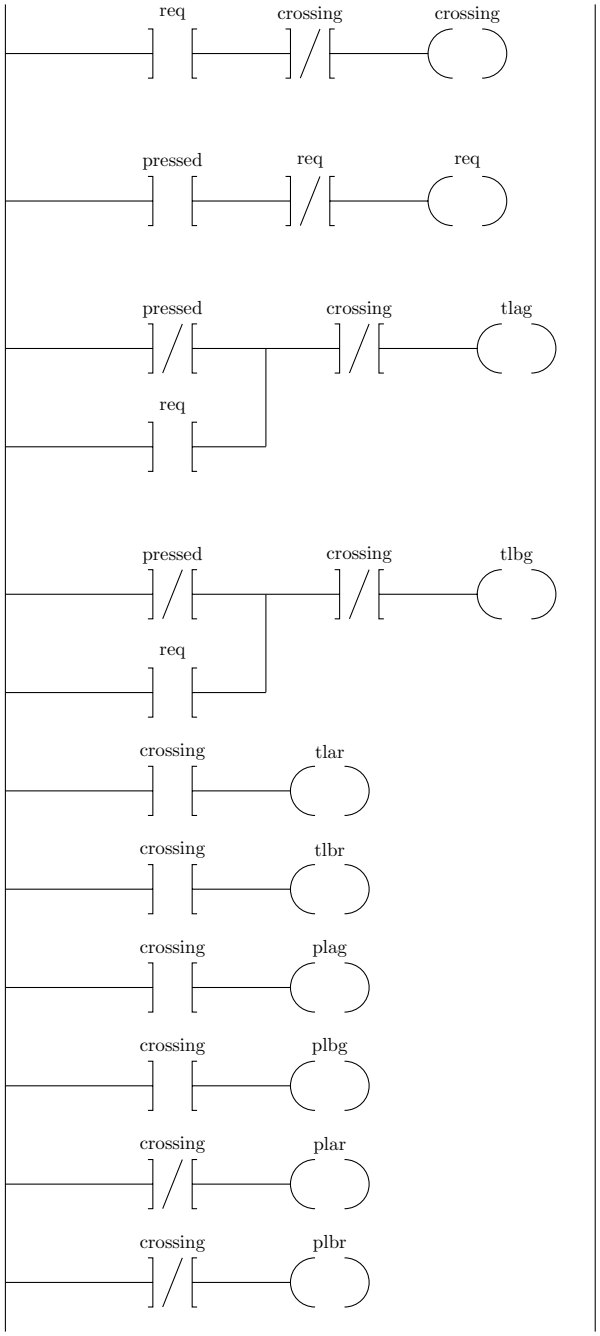


Figure 4.1: The Ladder Logic Program for the Pelican Crossing Example

For some $n \geq 0$ where each ψ_i with $1 \leq i \leq n$ is a propositional formula which has the following form.

- $\forall i.1 \leq i \leq n : c'_i \in C'$
- $\forall i, j.1 \leq i, j \leq n : i \neq j \rightarrow c'_i \neq c'_j$
- $\forall 1 \leq i \leq n : vars(\psi_i) \subseteq I \cup \{c'_1, \ldots, c'_{i-1}\} \cup \{c_i, \ldots, c_n\}$

$$
\begin{aligned}
[crossing' &\leftrightarrow (req \wedge \neg crossing,\\
req' &\leftrightarrow (pressed \wedge \neg req),\\
tlag' &\leftrightarrow ((\neg crossing') \wedge (\neg pressed \vee req')),\\
tlbg' &\leftrightarrow ((\neg crossing') \wedge (\neg pressed \vee req')),\\
tlar' &\leftrightarrow crossing',\\
tlbr' &\leftrightarrow crossing',\\
plag' &\leftrightarrow crossing',\\
plbg' &\leftrightarrow crossing',\\
tlag' &\leftrightarrow (\neg crossing'),\\
tlag' &\leftrightarrow (\neg crossing'),\\
tlar' &\leftrightarrow crossing',]
\end{aligned}
$$

Figure 4.2: A possible formalisation of the pelican from [Jam10]

4.2.1 Representing a Ladder Logic Program as a Labelled Transition System

Using the above definition we will now provide a new method for the semantic modelling of ladder logic programs using labelled transition systems. We define two sets of valulations representing the values of the input variables and the output variabes. We then use a function which captures the semantics of a ladder logic program to compute a new set of valuations for the output variables from the current valuations for the input and output variables.It uses a function which This differs from the approach presented by James [Jam10] where the semantics of ladder logic programs was modelled using an automaton and paired valuations.

We begin our definition by defining a set containing valuations for each of the input variables and a similar set for the output variables. We refer to these two sets as being our environments.

$$
\begin{aligned}
ENV_I &= \{\mu_I | \mu_I : I \to \{0,1\}\}\\
&= \{0,1\}^I\\
ENV_C &= \{\mu_C | \mu_C : C \to \{0,1\}\}\\
&= \{0,1\}^C
\end{aligned}
$$

We represent the semantics of our ladder logic program as a function $[\Psi]$ which takes the two current environments and returns a new environment for output variables representing a new state.

$$[\psi] : ENV_I \times ENV_C \to ENV_C$$
$$[\psi](\mu_I, \mu_C) = \mu'_C$$

Then each state variable c_i is calculated in order using the corresponding i'th rung of the ladder ψ_i where $1 \leq i \leq n$. The calculation makes use of the current valuations of environment variables μ_c, μ_I and the new valuations of output variables restricted to those evaluated before the current variable $\mu'_C \upharpoonright \{c_1, \ldots, c_n\}$. It could also be the case that there are certain output variables that remain constant and do not get affected by the execution of the ladder. Such a variable remains constant if it does not have a corresponding ladder logic rung.

$$\mu'_C(c_i) = [\psi_i](\mu_I, \mu_C, \mu'_C \upharpoonright \{c_1, \ldots, c_{i-1}\})$$
$$\mu'_C(c) = \mu_C(c) \textbf{ if } c \notin \{c_1, \ldots, c_n\}$$

Next we make use of the above to form a labelled transition system representing the ladder logic program.

Definition 27 (Labelled Transition System)**.** *A labelled transition system M is defined to be a four tuple (S, T, S_0, R), where*

- *S is a finite set of states.*
- *T is a finite set of transition labels.*
- *$S_0 \subseteq S$ is the set of initial states.*
- *$R \subseteq S \times T \times S$ is a labelled transition relation .*

We will now use the above definition to define a labelled transition system for ladder logic. This will give us representation of a ladder logic program which we can apply model checking to.

Definition 28 (Ladder Logic Labelled Transition System)**.** *We define a labelled transition system M for a ladder logic formula ψ to be a four tuple (S, T, S_0, R) which satisfies the following.*

- *$S = ENV_C$*
- *$T = ENV_I$*
- *$\mu_C \xrightarrow{\mu_I} \mu'_C$ if $[\psi](\mu_I, \mu_C) = \mu'_C$*
- *$S_0 = Init(ENV_C)$ where Init returns μ_C where all variables are set to false except those variables representing red lights*

We now want to be able to speak about the properties of the system that ensure safety. As we have seen in section three these properties are called safety conditions. An example of a safety condition for our pelican crossing example would be "Traffic lights and pedestrian lights are not green at the same time"

$$SafeLights \equiv (tla_g \wedge tlb_g \wedge \neg pla_g \wedge \neg plb_g) \vee (\neg tla_g \wedge \neg tlb_g \wedge pla_g \wedge plb_g)$$

Following the work of James we will define a safety condition for a ladder logic program as follows. The following definition is motivated by the fact that safety conditions tend to describe properties which hold for two consecutive cycles of the ladder logic program.

Definition 29 (Safety Conditions for a Ladder Logic Program)**.** *Given a ladder logic formula ψ over the variables in $I \cup C$ a **safety condition** is a propositional formula formed from the variables in $I \cup C \cup C'$*

Now that we have defined the model of our system and the type of properties we want to be able to speak about in that model, we must answer the following question. Given a model of our system and a safety condition, how do we check that the safety condition holds in that model?

Definition 30 (The Verification Problem for Ladder Logic Programs)**.** *We define the verification problem for a ladder logic formula ψ for a safety condition ϕ*

$$LTS(\psi) \models \phi$$

iff $\mu_C, \mu_I, \mu'_C \models \phi$ for all reachable pairs of states μ_C, μ'_C with transition condition μ_I in $LTS(\psi)$.

Modelling ladder logic programs using this method rather than the approach used by James [Jam10] has two advantages. The first is that due to the removal of the input variables from the set of state variables we have reduced the state space. The second is that we have managed to simplify the definition removing some of the redundancy found in that method.

Figure 4.3[1] shows the labelled for the pelican crossing example. This four state labelled transition system is simpler than the six state automaton[2] found in [Jam10]. There is one unreachable state in which both "crossing" and "req" are true.

Remark 2. *A different method for calculating the initial states was used in [Jam10]. The set is calculated by setting all input variables to false and performing one transition from every state. The states reached after one transition are the initial states. This method could be formulated as follows using our approach:*

$$S_0 = \{\mu'_C \in ENV_c | \exists \mu_C \in ENV_C, \mu_C \xrightarrow{False_I} \mu'_C\}$$

Where

$$False_I = \{\mu_I | \mu_I : I \to \{0\}\}$$

Using this approach would give the example labelled transition system seen in Figure 4.3 two initial states rather than one.

[1] The transition labelled 0,1 is in fact two transitions, one labelled with 1 and the other labelled with 0.
[2] James's automaton has eight states if unreachable states are included.

4.2.2 Formalisation of the Pelican Crossing Example

We will now present one possible formalisations of the pelican crossing example in the *SCADE* language. Other formalisations are possible and are presented in a later section of this thesis. We will then proceed to proving that two safety conditions hold in these representations of the ladder logic program.

In the following formalisation the boolean flows representing the red lights are initialised to true, all other flows are initialised to false. This is the initialisation used by Invensys in the Westrace railway interlockings. All successive values of the flows are dependent on the initial state and the value of *pressed*. The input of the node[3] is a boolean flow pressed that represents whether or not the button has been *pressed*. The output of the node are the results of each of the ladder logic rungs computed by the ladder logic program. The ladder logic program itself is contained in the declaration of the node. The initial state is specified by using the `->` operator. In order to express the dependency of the value of flows in the current state on flows in the previous state the `pre` operator is used.

```
node PelicanLadderLogic1(pressed: bool)

returns (req, crossing, tlag, tlar, tlbg, tlbr, plag,
 plar, plbg, plbr, audio: bool)

let
crossing = false -> pre req and (not (pre crossing));
req = false -> (not pre req) and pressed;
tlag = false -> ((not pressed) or req) and (not crossing);
tlbg = false -> ((not pressed) or req) and (not crossing);
tlar = true -> crossing;
tlbr = true -> crossing;
plag = false -> crossing;
plbg = false -> crossing;
plar =  true -> not crossing;
plbr =  true -> not crossing;
audio = false -> crossing;
tel
```

As already pointed out in 4.2.1 Remark 2 this formalisation differs from the one presented [Jam10] in the initial states. He assumes that the internal state of the system is unknown initially. Instead of setting all the internal variables he sets the inputs to false and checks what states are reachable in one transition from every possible state of the internal variables. This approach has the advantage that the system is proven safe from starting from every possible internal state. However since this is not used in industry some of the initial states are redundant.

[3] A node is the *SCADE* equivalent of a procedure or function.

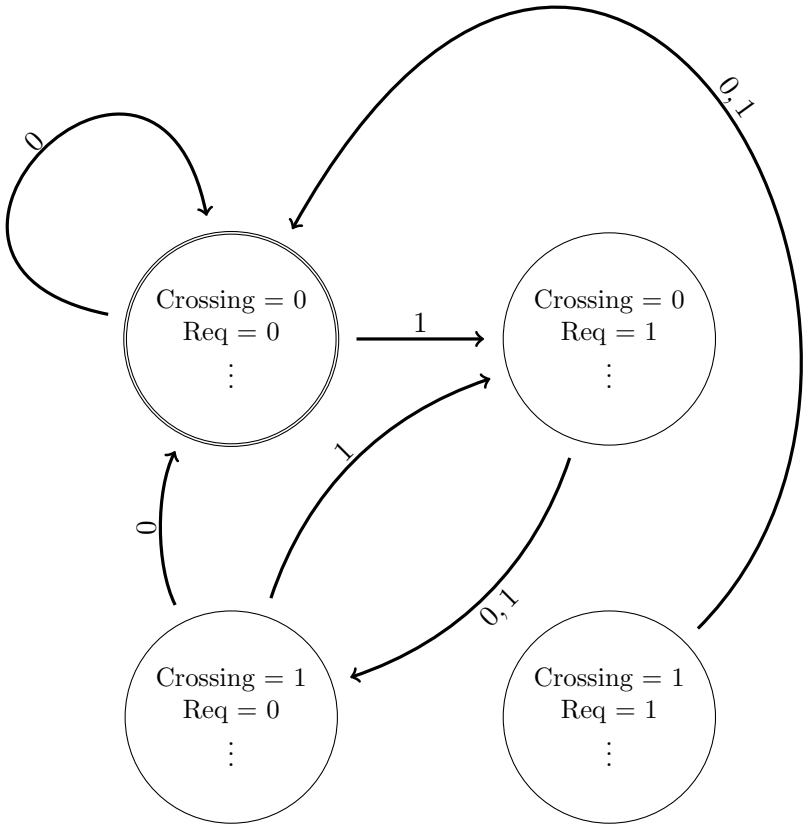


Figure 4.3: Pelican Crossing Transition System

4.3 Verification of Ladder Logic

We will now move on to the discussion of the verification of ladder logic programs. This mainly revolves around the verification of ladder logic programs using the *SCADE* suite. To begin this process another node was declared to model check Pelican Crossing 1. Like the ladder logic programs under test this node contains one boolean input flow namely pressed. It contains two extra outputs representing the two safety conditions being tested on each of the ladder logic programs. The body of the node contains an instantiation of the ladder logic with pressed as input. Each of the output flows have exactly the same name as they do in the ladder logic program. The safety conditions are then constructed by composing the output flows of the ladder logic program with logical connectives in order to express certain

properties. We modified the safety conditions so that they were not tested in the initial state which allowed the ladder logic program to stabilise into a safe state. This was done by using the temporal operator `->` to initialise the flow representing the safety condition to true. The flow then becomes the value of the safety condition in successive states.

safelights The safelights safety condition ensures that it is never the case that either of the traffic lights has a red aspect or a green aspect showing at the same time.

$$\neg(tlar \wedge tlag) \wedge (tlar \vee tlag) \wedge \neg(tlbr \wedge tlbg) \wedge (tlbr \vee tlbg)$$

safecross The safecross condition ensures that it is never the case that crossing is true and that the green lights are showing.

$$(crossing \wedge tlar \wedge tlbr) \vee (\neg crossing \wedge tlag \wedge tlbg)$$

```
node PelicanSafetyCond1(pressed: bool)

returns (safelights, safecross, req, crossing, tlag,
         tlar, tlbg, tlbr, plag, plar, plbg, plbr, audio : bool)
let

req, crossing, tlag, tlar, tlbg, tlbr, plag, plar, plbg, plbr, audio =
 PelicanLadderLogic1(pressed);



safelights = (tlar xor tlag) and (tlbr xor tlbg) ;
safecross = (crossing and tlar and tlbr) or (not crossing and tlag and
                    tlbg)

tel
```

The safety conditions “safelights” and “safecross” are both true in the initial state for our formalisation of the pelican crossing and therefore do not need to be initialised themselves.

4.4 Alternative Modelling Approaches

We will now present two alternative approaches that were considered to model the pelican crossing. The second formalisation is the same in all aspects as the first except, that the starting state of the ladder logic program is different. In this formalisation all of the flows are initialised with the value false. After the initial state the flows are formalised as before.

```
crossing = false -> pre req and (not (pre crossing));
req = false -> (not pre req) and pressed;
tlag = false -> ((not pressed) or req) and (not crossing);
```

```
tlbg = false -> ((not pressed) or req) and (not crossing);
tlar = false -> crossing;
tlbr = false -> crossing;
plag = false -> crossing;
plbg = false -> crossing;
plar = false -> not crossing;
plbr = false -> not crossing;
audio = false -> crossing;
```

A slight modification was made to this node in order to verify the correctness of the second formalisation of the pelican crossing. We added a temporal operator to the flow containing the safety condition so that the safety condition was tested after the initial state. This gives the flows a chance to stabilise into a safe state.

```
safelights = true -> (tlar xor tlag) and (tlbr xor tlbg);
safecross = true -> (crossing and tlar and tlbr) or
                          (not crossing and tlag and tlbg);
```

The third formalisation of our ladder logic program is the most similar to the work by James [Jam10]. The difference is that instead of initialising "crossing" and "req" with a truth value, they are initialised using an initialisation vector. This allows us to check different initialisations of the "crossing" and "req" variables using *SCADE*'s model checker without manual input. Some of these states are unreachable and therefore need to be eliminated using assumptions on the initialisation vectors. These input flows are declared as the boolean inputs "crossingi" and "reqi" in the nodes interface.

```
node PelicanLadderLogic3(pressed, crossingi, reqi, tlagi, tlari, tlbgi, tlbri,
                         plagi, plari, plbgi, plbri, audioi: bool)

returns (req, crossing, tlag, tlar, tlbg, tlbr, plag, plar, plbg, plbr, audio: bool)

let
crossing = crossingi -> pre req and (not (pre crossing));
req = reqi -> (not pre req) and pressed;
tlag = tlagi -> ((not pressed) or req) and (not crossing);
tlbg = tlbgi -> ((not pressed) or req) and (not crossing);
tlar = tlari -> crossing;
tlbr = tlbri -> crossing;
plag = plagi -> crossing;
plbg = plbgi -> crossing;
plar = plari -> not crossing;
plbr = plbri -> not crossing;
audio = audioi -> crossing;

tel
```

Verification of Pelican Crossing 3

A modification to the code for the original safety condition node is required in order to apply model checking to the third possible formalisation. It was necessary to add extra inputs to cope with the initialisation flows in the ladder logic program. We also added an assumption about these inputs to prevent the ladder logic program being initialised in a state that would be unreachable in an implementation.

```
node PelicanSafetyCond3(pressed, crossingi, reqi, tlagi,
     tlari, tlbgi, tlbri, plagi, plari, plbgi,
                          plbri, audioi: bool)


returns (safelights, safecross, req, crossing, tlag, tlar,
          tlbg, tlbr, plag, plar, plbg, plbr, audio : bool)

let
assume reqxorcross: reqi xor crossingi;

req, crossing, tlag, tlar, tlbg, tlbr, plag, plar, plbg, plbr, audio =
 PelicanLadderLogic3(pressed, crossingi, reqi, tlagi,
                                         tlari, tlbgi, tlbri, plagi, plari,
                                                     plbgi, plbri, audioi);



safelights =  (tlar xor tlag) and (tlbr xor tlbg) ;
safecross = (crossing and tlar and tlbr) or ((not crossing) and tlag and tlbg);

tel
```

4.4.1 Generating Counter Examples - Incorrect Pelican Crossing

In the following we demonstrate how counter examples are produced in SCADE and show a typical counter example trace. We have formalised an incorrect pelican crossing in ladder logic. It is a slight modification of our first correct pelican crossing that we formalised. It varies on the third and fourth lines which model the green aspect of the traffic lights. The `not pressed or req` has been replaced with `not pressed` in both cases.

```
crossing = false -> pre req and (not (pre crossing));
req = false-> (not pre req) and pressed;
tlag = false -> ((not pressed)) and (not crossing);
tlbg = false -> ((not pressed)) and (not crossing);
tlar = true -> crossing;
tlbr = true -> crossing;
plag = false -> crossing;
plbg = false -> crossing;
```

```
plar = true -> not crossing;
plbr = true -> not crossing;
audio = false -> crossing;
```

We then attempted to verify this example using the two safety conditions verified for the correct pelican crossing. *SCADE* correctly managed to falsify both of the safety conditions and the same counter example trace was produced (See Figure 4.4). Counter examples show how the state of the system evolves and leads to the violation of the safety condition. Typically they show how the input, output and internal variables change over time. The incorrect pelican crossing enters an unsafe state where none of the traffic lights are shown. Such counter example traces are useful to the engineers at Invensys Rail. They are essential to understand the nature and cause of the violation.

Var	1	2
IncorrectPelican		
Inputs		
pressed	FALSE	TRUE
Outputs		
req	FALSE	TRUE
crossing	FALSE	FALSE
tlag	FALSE	FALSE
tlar	TRUE	FALSE
tlbg	FALSE	FALSE
tlbr	TRUE	FALSE
plag	FALSE	FALSE
plar	TRUE	TRUE
plbg	FALSE	FALSE
plbr	TRUE	TRUE
audio	FALSE	FALSE
safecross	TRUE	FALSE

Figure 4.4: Counter example trace for safecross

4.5 Verification of two Real World Interlockings

Due to the size of real life ladder logic programs manual translation is not feasible due to time and the likelihood of human error. Therefore we created a tool written in Haskell that automatically translates ladder logic programs into *SCADE*. Our tool is a modification of a tool created by James. Using a modified version of the tool presented by James [Jam10] it was possible to automatically translate ladder logic programs into the *SCADE* language.

$$\textit{Ladder logic} \overset{\textit{Translated}}{\underset{\textit{by tool}}{\Longrightarrow}} \textit{Scade language}$$

The tool translated the ladder logic program without encoding any of the safety conditions.

It was necessary to hand code each of the safety conditions into the *SCADE* model so that we could verify them using the model checker.

4.5.1 Adding Invariants

There are certain states in the ladder logic program that are not reachable through any possible execution of the program. This discussion is based on results produced in [Kan08].
Definition 31 (Invariant)**.** *An **invariant** is a formula ψ which holds for all reachable states of the ladder and is constructed only using atomic propositions contained within that ladder.*

If our model does not capture the invariants of a logical system then it is possible that so called false negatives are produced in the verification process. A safety condition could be violated by one of the unreachable states and a counter example produced. The execution that leads to this unsafe state would never happen in reality therefore the system could still be considered safe. To be certain however we would need to better understand the system and include the invariant in our model.

Physical Invariants

One of the disadvantages of the ladder logic formalisation it is under-specified and does not take into account some of the constraints caused by the physical environment. These constraints are called **physical invariants**. For example consider a three way switch (Figure 4.5) which has three positions (A,B,C) in which contact can be made and a circuit formed. It is only physically possible for the switch to close one of the contacts at a time. A ladder logic formalisation of the switch does not take this into account. Following the Invensys approach the switch would be formalised as three booleans, one for each of the contacts. No information would be encoded in the program to enforce that only one of the booleans can be true at any given time. Such invariants require a detailed analysis as their inclusion in the model could remove some redundancy. In this case it could be possible that a **paper clip** (short circuit) occurs and all three contacts become closed. The safest option could therefore be to model a switch using the Invensys approach as it includes the possibility of this fault. We describe a method to do this in the next paragraph.

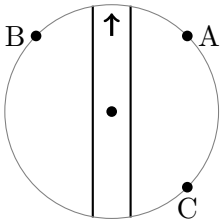


Figure 4.5: A Three-Way Switch

Design-By-Contract Approach

The *SCADE* suite makes use of the design-by-contract approach [Est09]. This was first introduced for the object-orientated language Effiel in work by Meyer [Mey92]. The motivation behind design-by-contract was to provide a new methodology which would improve the reliability of software. The general idea behind it is that you provide a contract specifying the behaviour of a node. A contract consists of preconditions that are required for a correct execution of the code along with postconditions that will be met if the code is executed correctly.

- The **assume** observer allows us to provide preconditions that speak about the value of inputs and the past value of outputs.
- The **guarantee** allows us to provide postconditions that speak about the value of inputs and outputs.

An invariant can be formalised for the three way switch Figure 4.5 in the *SCADE* language. Using the assume contact observer we add an invariants of the form $switch\_posA \rightarrow (\neg switch\_posB \wedge \neg switch\_posC)$ for each of the positions.

```
assume switchInVar1 : not switch_posA or
                      (not switch_posB and not switch_posC);

assume switchInVar2 : not switch_posB or
                      (not switch_posA and not switch_posC);

assume switchInVar3 : not switch_posC or
                      (not switch_posA or switch_posB);
```

We have used this approach combined with the work of Kanso [Kan08] to successfully add approximately one hundred invariants to one of the real world railway interlockings. These invariants however did not eliminate all of the false negatives produced in the verification process. Further work would have to be performed in co-operation with Invensys Rail to isolate the remaining invariants.

4.5.2 Results

Using the approach described above we have managed to successfully translate two real world railway interlockings into the *SCADE* language. We are not allowed to give the details of the real world example as it would break a confidentiality agreement To give an impression of the size of the programs they contained approximately six hundred variables and three hundred and fifty rungs each. Safety conditions were encoded for each of the railway interlockings into the *SCADE* language and verification was performed upon them.

- Railway Interlocking A: No invariants added , two safety conditions verified
- Railway Interlocking B: Approximately one hundred invariants added, approximately one hundred safety conditions verified of which approximately 40% produced false counter examples.

Typically the time it took to verify a safety condition was less than a second. There were a few exceptions to this where the verification process took ten seconds or more. This shows the use of *SCADE* for the verification of railway interlockings is viable.

The large amount of false counter examples produced during the verification process is due to the underspecification of the ladder logic program. We have contacted Invensys Rail regarding the false counter examples however we have yet to receive a response from them. The removal of the false counter examples and completion of the *SCADE* formalisation could be the topic of future work in this area.

4.6 A Comparison of Different Model Checkers

In order to test the performance of *SCADE* we perform a comparison between *SCADE* and other model checkers. This will give us empirical results indicating whether *SCADE* outperforms or under-performs compared with its competition i.e. KIND and the tool produced by James [Jam10].

4.6.1 Tool By James

We have mentioned several times already during this thesis that this work is based on that of James [Jam10]. Therefore it is natural for us to compare our approach to the one used by James. James created a proto-typical tool for the verification of ladder logic programs based upon earlier work by Kanso [Kan08]. Underlying the tool is the Paradox model finder [CS03] and the SAT-solver MiniSat [EN04, Min]. It allows for the application of the following model checking techniques:

- Inductive Verification
- Bounded Model Checking
- Temporal Induction

One of the aims of this approach was to increase the efficiency of the model checking process through the use of program slicing. This technique removes formulas from the model of the ladder logic program which do not affect the safety condition. This leaves only the formulas upon which the safety condition depends. The sliced ladder is typically 25% of the size on the original. This allowed up to ten times as many states to be checked using bounded model checking and a considerable performance increase.

4.6.2 The Kind Model Checker

The KIND safety property verifier [Hag08, HT08] for Lustre was used to provide a comparison between the SAT-based model checkers that have been applied to this problem so far an SMT-based model checker. KIND could be seen as one of the *SCADE* suites academic competitors and therefore a comparison would give us an insight in the capabilities of *SCADE*. KIND allows for different SMT-solvers to plugged in. In this case we have chosen to use the SMT-solver

Yices [DDM06] which has been shown to perform well in industrial applications [JES07]. KIND reasons about an idealised version of Lustre programs in which machine integers and floating point numbers are replaced with unbounded integers and infinite precision rationals respectively. These idealised programs are then modelled using a typed first-order logic that includes uninterpreted function symbols along with built-in integers and rationals. This logic is called **Idealised Lustre Logic** which we shall denote using $\mathcal{IL}$. There is a particular advantage of using $\mathcal{IL}$ that makes it attractive to use for model checking purposes. That is the problem of checking the validity of its quantifier-free formulas that contain *linear* numerical terms is decidable in such a way that particularly suited to SMT techniques.

Idealised Lustre Logic

One way of formalising Lustre into $\mathcal{IL}$ is presented in [Hag08, HT08]. To translate from Lustre programs to $\mathcal{IL}$ we represent each stream x of values of type τ using an uninterpreted function of type $\mathbb{N} \to \tau$. An equation consisting of multiple streams can be translated into universally quantified equations which speak about the values of streams. For example the stream x = y * z would be translated into $\forall n : N.x(n) = y(n) * z(n)$. In this case the integer streams x, y and z have been translated into function symbols of type $\mathbb{N} \to \tau$. The initialisation operator $\to$ is translated using the *ite* operator. For example consider the stream $x = y * z \to y * z - pre\ x$ this would be translated into the following $\forall n : \mathbb{N}.x(n) = ite(n = 0, y(0) * z(0), y(n) * z(n) - x(n-1))$.

This approach can be expanded to a (idealised) Lustre program consisting of a single node N as follows. Our Lustre program consists of stream variables $\mathbf{x} = \langle x_1, \ldots, x_p, y_1, \ldots, y_q \rangle$ these are comprised of input variables $x_1, \ldots, x_p$ and local and output variables $y_1, \ldots, y_q$. We can express the semantics of N using $\mathcal{IL}$ by applying universal quantification to a variable n over the following system of equations.

$$\Delta(n) = \begin{cases} y_1(n) &= t_1[\mathbf{x}(n), \mathbf{x}(n-1), \ldots, \mathbf{x}(n-d)] \\ \vdots & \\ y_q(n) &= t_q[\mathbf{x}(n), \mathbf{x}(n-1), \ldots, \mathbf{x}(n-d)] \end{cases}$$

For y_i as translation is performed t_i on its defining express in N. We define a state for N to be any of the tuples of values for $\mathbf{x}(n)$ that are well-typed. Each of the $y_i(n)$ is a function of the previous states at time $n, n-1, \ldots, n-d$. The maximum "nesting depth" d represents the maximum distance into past which the temporal pre operator used to take values from. In the rest of our discussion of KIND we will consider only the case that $d \leq 1$ which is typical of ladder logic programs. We need to be able to speak about possible executions over the program. Therefore we need to way of speaking about the inputs a program has received during its execution. We define a trace of a program N to be a tuple $\mathbf{s} = \langle s_1, \ldots, s_{p+q} \rangle$ where each s_i with $1 \leq i \leq p + q$ is the same type as $\mathbf{x}$. A path is defined to be a finite sequence configurations that are produce by a specific trace. A legal trace s is one which satisfies the formula $\forall n : N.\Delta(n)$ when the stream variables $\mathbf{x}$ are those of $\mathbf{s}$. A reachable configuration is one that is produced in a legal trace. If a path is in the initial segment of a legal trace then that path is said to be initial. We define an Invariant or Safety property P of configurations

for a node N if it holds in all reachable configurations. This forms the basis for the formal verification of invariants for Lustre programs. Typically this model of $\mathcal{IL}$ is then lifted and adapted so that a variety of SAT-based model checking techniques can be applied.

4.6.2.1 Applying k-Induction Using an SMT-Solver

We will now discuss one possibility for the verification of these $\mathcal{IL}$ programs following [Hag08, HT08]. We define k-induction using an equational system $\Delta(n)$ which models a node N in $\mathcal{IL}$. We express a safety property P which speaks about the configurations of N using a quantifier free formula $P(n)$ formulated in $\mathcal{IL}$ using $\mathbf{x}(n)$. For a given integer term t in $\mathcal{IL}$ it is possible to replace every occurrence of n in $\Delta(n)$ using t. The resulting formula shall be denoted as Δ_t. This replacement can also be performed with $P(n)$ and the resulting formula is denoted P_t.

We define k-induction using the following two statements, for some $k \geq 0$ and constant integer n, to show that P is a valid safety property or invariant for N.

$$\Delta_0 \wedge \Delta_1 \wedge \ldots \wedge \Delta_k \models_{\mathcal{IL}} P_0 \wedge P_1 \wedge \ldots \wedge P_k \tag{4.1a}$$

$$\Delta_n \wedge \Delta_{n+1} \wedge \ldots \wedge P_n \wedge P_{n+1} \wedge \ldots \wedge P_{n+k} \models_{\mathcal{IL}} P_{n+(k+1)} \tag{4.1b}$$

Logical entailment in $\mathcal{IL}$ is denoted using $\models_{\mathcal{IL}}$. Verification is performed by passing the two statements to an SMT-solver with an instantiated with an increasingly large k until one of the following occurs:

- The base case is proven invalid. In this case P is invalid and a it is possible to extract a counter example path from the $\mathcal{IL}$-model of $\Delta_0 \wedge \ldots \wedge \Delta_k \wedge \neg(P_0 \wedge \ldots \wedge P_k)$.
- The base case and the induction step are proven valid. This shows that P holds for reachable configurations and therefore it is an invariant or valid safety property of the node N.

4.6.3 Results of the Comparison

The one comparison we have performed so far was performed using the valid safety condition **Conflicting Routes Not Permitted** on the ladder logic program for a real railway interlocking. This is a typical example of the type of safety condition proven and therefore allows us to test the performance of the tools. It can be represented by the following formula in propositional logic.

$$\neg 8252.RU \vee \neg 8253(2).RU$$

In the Figure 4.6 we see the results of our comparison between the three tools. We see that on this safety condition the results show *SCADE* is comparable to the other two model checkers when some form of induction is applied.

A further comparison could be performed that pushes the capabilities of the all of the tools. Using some of the safety conditions that produced results deviating from the norm. However it has not been possible so far to encode such a safety condition into the model checker of tool by James.

Tool Used	Technique Applied	Result	Time Taken
SCADE	Induction	Valid	1 second

Tool Used	Technique Applied	Result	Time Taken
Tool by James with program slicing	Induction	Valid	0.122 seconds
	BMC[4] = 1000	Valid	2m 20.1s
	BMC = 750	Valid	1m 32.4s
	BMC = 500	Valid	0m 54.8s
	BMC = 250	Valid	0m 24.1s
	Temporal Induction		
	Base BMC = 10	Valid	0.990s
	Induction	Valid	4.934 s

Tool Used	Technique Applied	Result	Time Take
KIND	K-induction	Valid	0.352 seconds
	BMC = 1000	Unknown	389m 53 seconds
	BMC = 750	Unknown	222m 27 seconds
	BMC = 500	Unknown	49m 45 seconds
	BMC = 250	Unknown	6m 5.461 seconds

Figure 4.6: Results of the comparison

[4]BMC Stands for bounded model checking. Each BMC entry in the table is accompanied by a bound.

Chapter 5

Concrete Modelling of the Railway Domain

There is a large amount of interest from industry in concretely modelling parts of the railway domain.

Formalising the model in a language with formal semantics has a major advantage over writing a specification in natural language. It allows us to apply a variety of verification techniques which can be used to check the correctness of the model. Some progress has already been made in creating an approach which allows such a formalisation to be produced. One attempt used the algebraic specification language RAISE (see [HP99]) to create a highly abstract algebraic specification of the model. This algebraic specification was then refined and an implementation was produced. This approach has the advantage that it is possible to verify that safety properties hold for the abstract model. Since the implementations are refinements of the abstract model the safety properties also hold in the implementation. Another attempt has been made using Lustre to model a segment of railway as an asynchronous distributed system (see [CMSW99]).

5.1 Modelling Components

Despite the fact that the ladder logic method of specifying railway interlockings is fairly reliable and well used, Invensys Rail are looking to move towards a new approach. They would like to use higher level languages which allow them to speak about concrete properties of the railway domain. Using these richer higher level languages would also allow more of the railway to be captured in the model. This is a step away from a control engineer orientated approach towards a computer scientist orientated approach. Another aspect of modelling that industry is interested in is modularity. They would like to capture the behaviour of generic components giving them a kit from which they can assemble segments of railway. Some of the verification can be performed at the component level, verifying individual components or commonly used combinations of components. This reduces the overall amount of verification that needs to be performed on the resulting railway segment.

The requirements for the behaviour of the components was not completely specified in one individual document but rather came from multiple places. Capturing the behaviour of these components in a document is a task in itself and is beyond the scope of this thesis, however it may be part of future work. Some information was gathered from more formal documents such as the control tables and the data model document provided by Invensys Rail. Other information was taken from documents such as the Introduction to Railway Signalling [KR01]. Our aim here is to provide one insight into how a modelling process could look.

The way we have modelled the track was based on the small example found in the work by Caspi et al [CMSW99]. In this example a segment of track was modelled as a node in Lustre. The movement of a train was captured in the node by having inputs and outputs that model when a train enters or leaves that segment of track. This allowed the topology of a simple track to be captured by linking the inputs and outputs of several of the track segment nodes. In our modelling approach we will be looking to extend the idea of capturing the topology. In [CMSW99], the lights and points were modelled as being part of the track segment. We intend to capture these components separately making our approach more modular. Unlike the work [CMSW99] we will not be trying to capture the asynchronous nature of their modelling approach. The control system employed by Invensys Rail synchronises on a clock. We will now move on to discussing how the components were modelled in our approach using the *SCADE* language.

One of the aims of this approach was to model individual components of the railway in a modular fashion. We will also use this as an opportunity to examine the use of finite state machines in *SCADE*. We will now present the track components we have modelled as part of this approach along with some motivation as to why they were modelled in the way they are. The *SCADE* language code for these track components can be found in the Appendix.

5.1.1 Track Segments

Having a concrete model of track is crucial to our railway modelling. If the track was not modelled it would be hard to justify that the resulting model was indeed one of the railway.

The first thing we have to consider is; what sort of behaviour do we want from our model? This should be closely followed by the question: What sort of data should our model contain?

- When do trains enter a segment of track.
- The track should become occupied if a train enters.
- If a track is already occupied and a second train enters, then we would like to know this. A crash could occur.
- If a track is occupied and there is a adjacent signal showing only a green light, then a train should leave the track.

Rather than create a new data type for trains we have decided to follow the work by Capsi [CMSW99] and use boolean data flows to model the movement of trains. Boolean data flows were also used to indicate whether a signal is shown at the side of the track and whether a crash has possibly occurred. The behaviour of the track segment was captured by a *SCADE* state machine see 5.1. The state machine has 4 possible states (Empty, Occupied, TrainLeave,

Crash) each with 3 internal variables. The variable TrackOcc keeps track of whether a train is currently occupying the track. If another train enters this segment of track while it is occupied then the variable Crash is set to true indicating the possibility of a crash. The third variable TrainOut indicates that a train is in the process of leaving this track segment and entering the next.

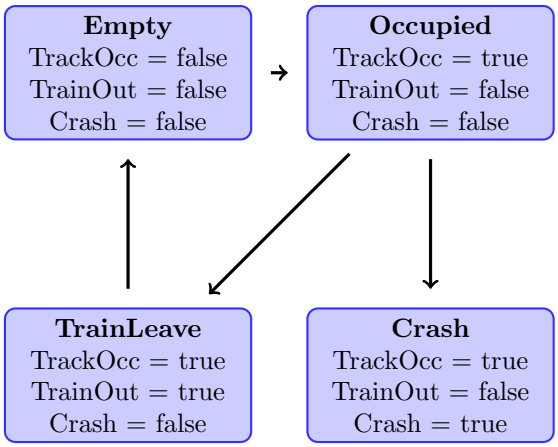


Figure 5.1: Scade State Machine For A Track Segment

Connecting Track Segments

We will now present an example demonstrating how to connect two track segments using our approach. This is performed by instantiating two Track_Segment1 nodes each of which models a straight track segment . Then we feed the output of one of the nodes into the input of the other.

```
TRACKOCC_1, TRAINOUT_1 , TRACK1_CR =
             Track_Segment1(TRAININ, false, false, true);
TRACKOCC_2, TRAINOUT_2 , TRACK2_CR =
              Track_Segment1(TRAINOUT_1, false, false, true);
```

In this example `TRAININ` represents that a train has entered the piece of track. `vTRAINOUT_1` and `vTRAINOUT_2` both represent that a train has left their corresponding segments. The track segments have 3 inputs which are used to model track side signalling. In this example they are set to false, false, true which represents that the previous light seen was green and that trains are free to proceed.

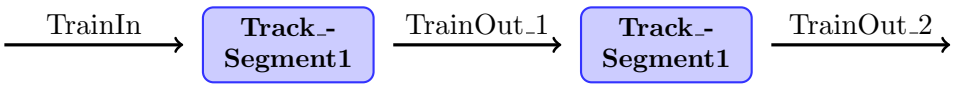


Figure 5.2: Connecting track segments to form a one way track

5.1.2 Signals

The signals were modelled as two separate components, the signals themsevles and the aspects they contain. First a light aspect was modelled separately of the signals, then multiple light aspects were instantiated to model the different types of signal.

3 different types of Signals have been modelled so far using this approach[1].

- 3 Aspect Signal: (Green, White, Red)
- 2 Aspect Signal : (Green, Red)
- Single Aspect Signal: (Fixed Red)

The railway data model provided by our industrial partner Invensys Rail contained some information regarding the formalisation of signals and signal aspects. It specified that a signal aspect should have 3 boolean fields (Available, Driven and Reported) and that a signal can be made up of any number of signal aspects. The boolean fields are described in the following:

Available This is true is the aspect is available to be set , false otherwise

Driven If this is true then the aspect is currently active and being displayed, otherwise it is false

Reported The signal is faulty and has been reported to the control system

This document did not specify any concrete behaviour however. Therefore we had to use some intuition and when formalising the behaviour of the signals. Both the signal aspect and the signals were modelled using the *SCADE*'s built in state machines.

The Signal aspect was modelled as having 3 boolean inputs representing it is being set Available, Driven and Reported. It also has 3 outputs for Available, Driven and Reported that are used to represent the internal state of the aspect. Every possible combination of outputs is allowed giving the aspect 8 internal states. The value of Available influences whether or not a signal aspect can be driven. If the aspect is available then the value of driven can be changed if it is not available then the value of driven becomes locked. The value of reported can be changed at any time, its behaviour was not implemented at the current time. It was intended to model that the light has become broken and has been reported to the control system.

Modelling the availability of a signal aspect in this way means that the complexity of the node modelling this signal increases in order to incorporate the slight delay in changing the state of a signal aspect. We also considered signal needs to be modelled in such a way that we do not enter a state where two lights or no lights are showing. If there was for instance a cut in the control cable to the signal then it should enter a safe state where a red light is showing.

The two and three aspect lights were modelled as having a boolean input for each of the aspects that could be driven. The outputs consist of 3 booleans representing whether or not

[1]One extra type of light that could be modelled in the future is a 4 aspect light (Green, White, White, Red) however since this is only used on high speed lines we have chosen not to model this at the current time

each of the signal aspects has been driven and a further boolean representing whether or not the signal is set. The signal is set if the lights are not in the process of changing and signal show is being driven by the inputs. Internal variables are used to store the value of the outputs from the previous cycle.

Every signal we have modelled initialises with the red aspect set as this is the failsafe state. In order for the aspect showing on the light to change it has to go through several intermediate stages. These intermediate states ensure that the signal has at least one signal driven and only one signal driven. The variable changes are described in the diagram 5.3.

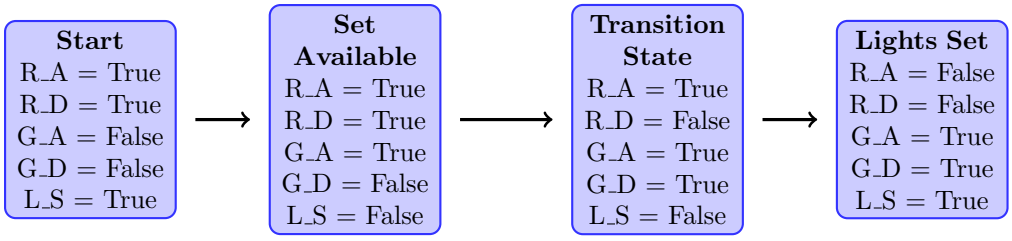


Figure 5.3: The Transition Between Signal Aspects

The following is a description of the variables in 5.3

R_A Controls the availability of the red aspect

R_D Drives the red aspect

G_A Controls the availability of the green aspect

G_D Drives the green aspect

L_S Indicates that the lights are set

Connecting a Track Segment With a Signal

We will now present an example demonstrating how to connect a track segment with a 3 aspect signal using our approach. This is performed by instantiating two *SCADE* nodes one of which models a track segment the other models a 3 aspect signal. The relationship between the two components is modelled by passing the output of the lights into the input of the track segment.

```
LIGHTS_SET , SHOWS_RED , SHOWS_WHITE, SHOWS_GREEN =
             Light3Aspect(DRIVE_RED , DRIVE_WHITE, DRIVE_GREEN);
TRACKOCC_1 , TRAINOUT_1 , TRACK1_CR =
             Track_Segment1( TRAININ, SHOWS_RED, SHOWS_WHITE, SHOWS_GREEN );
```

In this example `SHOWS_RED`, `SHOWS_WHITE` and `SHOWS_GREEN` represent that the signal is displaying the aspect which corresponds to the variable . The variables `DRIVE_RED`, `DRIVE_WHITE`

and DRIVE_GREEN represent a electronic signal from the control system driving the signal to display the corresponding aspect.

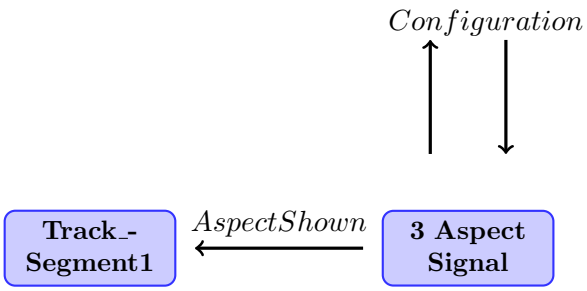


Figure 5.4: Connecting a track segment with a signal

5.1.3 Points

A point does not have any formal specification in the form of a document so we once again direct the reader to our book on railway signalling [KR01]. It can have two physical positions, i.e. normal or reverse. The point can also have a locking mechanism activated, locking it in one of the positions. These physical characteristics give our model a total of 4 states. The point can be driven by electronic signals to either the normal or reverse position. These two driving signals for the point are modelled as two separate inputs. This allows us to model the situation where there has been a communications failure between the point and the control system. This adds a form of redundancy to our model. If the track becomes occupied by a train then the point should lock in what ever position it is in until the train has left that track segment. This behaviour can be seen in the diagram 5.5.

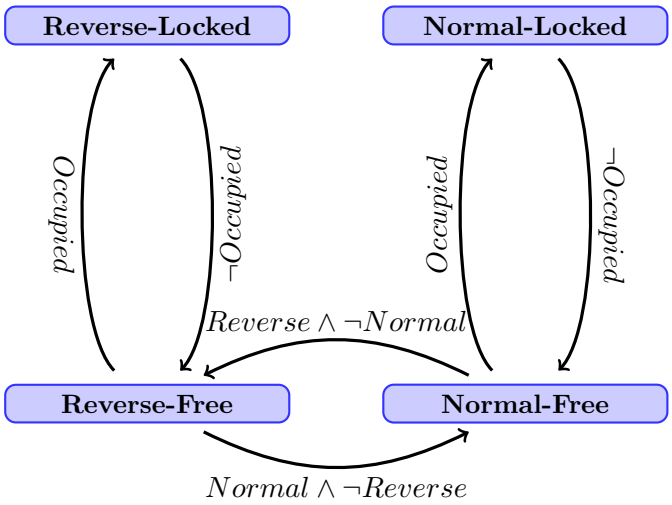


Figure 5.5: The Transition Between Signal Aspects

5.1.4 Routes

Unlike the other components, the route has a more formal specification in a document called the control table. The control table lists the routes for a track, what happens when a route is set and which routes conflict with each other. An example of a typical control table entry is as follows:

ROUTE No.	EXIT	ASPECT	SIGNAL AHEAD	TRACKS	POINTS NORMAL	POINTS REVERSE
508 A	500	Y	500 AT R	VJ VH VG	764 768	
		G	500 AT Y			
508 B-1	498	Y	498 AT R	VJ VH VG	768	764
		G	498 AT Y			

ROUTE No. This identifies the route and also identifies the signal allowing entry to the route.

EXIT This identifies the signal at which this route finishes.

ASPECT and SIGNAL AHEAD This links the aspect that is shown on the signal at the start of the route with the aspect that is shown on the signal at the end of the route. The first case on this table can be read as follows: A Yellow aspect will show on signal 508 if a Red aspect is shown on signal 500.

TRACKS This indicates which track circuits make up the route. If any of these tracks become occupied the signal from the track circuits should be fed back into the control system. This should cause the signal at the start of the route to go red.

Points Normal Points that are set Normal when the route is called.

Points Reverse Points that are set Reverse when the route is called.

Two routes are said to be conflicting if they set the same point in different positions. These conflicting routes can not be set at the same time. In the example control table above the two routes conflict on point 764. We have modelled a route as being an entity which sets the lights and points as mentioned in the control tables. Our model of a route also receives information indicating whether or not the track segments contained within are occupied.

5.2 Railway Example

Now that we have our components modelled the next task is to combine them together to form a segment of railway. Our industrial partner provided us with a detailed track plan for one of the real life interlockings. We used this track plan as a basis for an example to show how our modelling approach can be applied. Some of this detail was not necessary in order to explore the modelling capabilities of the *SCADE* language and would have created an unnecessary extra workload, so we abstracted away from some details.

5.2.1 Modelling a Junction

The following figure 5.6 shows how components are used to construct a junction which can be seen in the track plan figure 5.7. The nodes themselves are represented in the diagram by the boxes and the data which flows between them is shown with the arrows. The nodes which model the routes are not included in the diagram but the data which would be communicated to them is labelled.

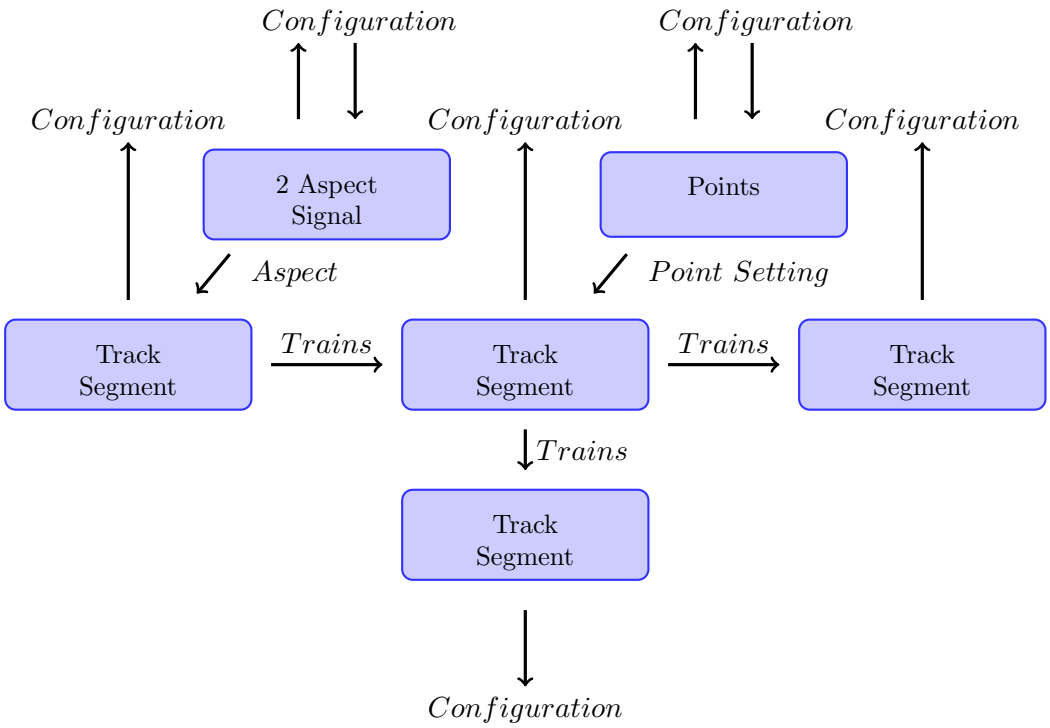


Figure 5.6: How the components communicate in a model junction

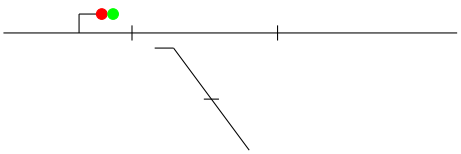


Figure 5.7: The track plan for the junction

This junction is represented in *SCADE* using the following code. We have instantiated several Track_Segment1 nodes to represent straight segments of track entering and leaving the junctioning. One Track_Segment3 node was used to model the junction itself.

```
-- Top left track and signal
LIGHTS_SET , SHOWS_RED , SHOWS_GREEN =
             Light2Aspect(DRIVE_RED , DRIVE_GREEN);
```

```
TRACKOCC_1 , TRAINOUT_1 , TRACK1_CR =
             Track_Segment1( TRAININ , SHOWS_RED, false , SHOWS_GREEN );

-- Junction

NORM_FREE , NORM_LOCK , REV_FREE , REV_LOCK =
            Point(DRIVE_NORM, DRIVE_REV, TRACKOCC_2);

TRACKOCC_2, TRAINOUT2_1 , TRAINOUT2_2, TRAINOUT2_3, =
   Track_Segment3(TRAINOUT1_0, false , false,
   false, true, false, NORM_LOCK , REV_LOCK);

-- exit tracks

TRACKOCC3_1 , TRAINOUT3_1 , TRACK3_CR =
             Track_Segment1( TRAINOUT2_2, false, false , true );

TRACKOCC4_1 , TRAINOUT4_1 , TRACK4_CR =
             Track_Segment1( TRAINOUT2_3, false, false , true );
```

We will now discuss a larger example of a railway segment which was based on a track plan provided to us by our industrial partner Invensys Rail. The simplified railway segment contains the following components. The track plan used to layout these components can be seen in figure 5.8.

- 11 segments of track
- 4 points
- 6 routes (4 of which operate points)
- 7 lights
- A route controller

The track segment contains two routes control the entrance and exit track segments for this piece of railway. These routes operate the left most signals and do not control any points. Out of the four routes that cross the junction, two of them control the flow of incoming trains across the junction. The two incoming routes set the points such that trains will be directed to enter either the top platform or the bottom platform. The remaining two routes allow trains two leave the station and proceed down the outbound track.

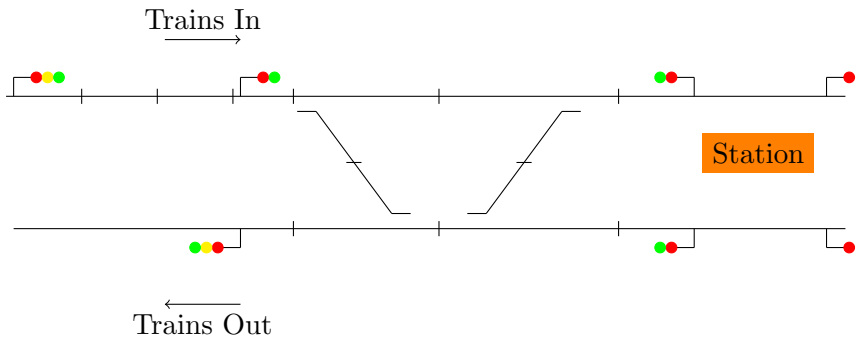


Figure 5.8: The track plan for our abstract railway

This was modelled as one node in the *SCADE* language. While the node has a large number of internal variables, it only has seven input variables. Six of these input variables represent that a route is being requested and are passed to the Route controller. The other variable represents that a train has entered our railway and is passed to the node representing the eastern most segment of track.

5.2.2 Route Controller

Using only the components we have specified so far does not allow us to model the complete behaviour of the railway. There is certain information contained in the control table that speaks about routes that we have not captured in our approach so far. Mainly we need to model that routes can conflict with each other. To do this a Route Controller was added to our model. This filters route requests and only allowing non-conflicting route requests to be passed to the Routes. Routes also pass information on whether or not they are selected back to the controller.

5.3 Verification

There are 3 different types of safety condition we managed to verify in this approach.

- Some of the applicable safety conditions from the first example were verified.
- Some of the safety conditions from the first example were modified so that they could be applied to individual components, possibly under certain side conditions.
- New safety conditions were verified expressing properties of the topology.

5.3.1 The Verification of Safety Conditions from the First Approach

In order to show that our new modelling approach has correctly captured the way the railway should behave, we attempted to verify some of the safety conditions from the first approach.

- When a route becomes set it causes its associated points to lock.

- No proceed (green or white) aspect is shown on the lights if none of the routes controlling them are set.
- No proceed aspect is shown if a train occupies the track controlled by the signal.

5.3.2 New possibilities for Verification Using this Approach

We will now look at some of the new possibilities that this approach allows for the verification of a railway model.

Modular Verification

The modular nature and topological aspect of this approach gives us new opportunities for verification that weren't possible in the first approach. The modularity of this approach allows us to verify individual components or modules containing groups of components.

In the first approach the safety condition "No green aspect should be shown if a train is in route" or

$$TrainInRoute \rightarrow \neg GreenAspect$$

was proven for each route of the ladder logic program.

The new approach has given us the opportunity to verify this safety condition for each individual route and can be seen in the appendix B.5.1. The safety condition was modified to include the side condition that the route had been called. Properties about other the other lights set by the route were also proven. The variables `W_track_clear` and `G_track_clear` both represent that the track required for their corresponding light (White or Green) is clear.

$$(RouteCall \wedge \neg W\_track\_clear \wedge \neg G\_track\_clear) \rightarrow (DriveRed \wedge \neg DriveGreen \wedge \neg DriveWhite)$$

The correctness of our signals was modified using the two safety conditions. The first (see 5.1) was used to check that no more than one light was showing. The second (see 5.2) was used to show that at least one light was always shown in our model. These could of course be combined into one safety condition but for readability we will present them as two separate formulas.

$$\neg(Green \wedge White \vee Green \wedge Red \vee White \wedge Red) \tag{5.1}$$

$$Green \vee White \vee Red \tag{5.2}$$

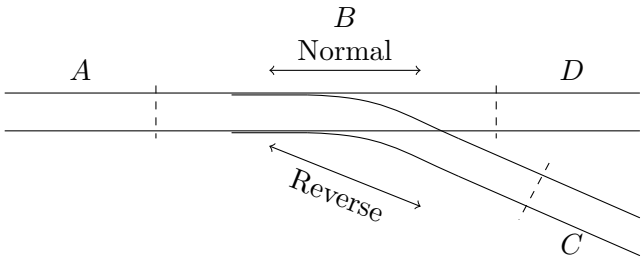


Figure 5.9: A Typical Junction

Verification of Topological Properties

Consider a simple model of a junction Figure 5.9 made up of several track segments and a point. We can now verify properties which speak about how a train should move around this junction. If a train enters junction B at A and the point is set in Normal, then the train should leave the junction heading towards D. Similarly, if a train enters the junction B at A and the point is set in Reverse then the train should leave the junction heading towards C. This safety property was formalised in the *SCADE* language along with a junction using our modelling approach. The safety property was then verified and shown to hold in our model of a junction. It is also possible to speak about how lights affect the movement of the trains whereas in the previous approach this was not possible.

The junction example also highlights another new possibility that has resulted from this approach. Individual components or modules of components can have safety properties expressed for them and verified. Safety conditions were also verified for two connected pieces of track. These ensured that any trains entering a segment 5.3 or moving between two segments 5.4 would behave as expected. We want to ensure trains do not disappear when moving around track.

$$Pre\_TrainIn \rightarrow TrackOccupied \tag{5.3}$$

$$(\neg Pre\_vTrack1\_Out) \vee (vTrack2\_TOcc \wedge (\neg Pre\_TrainIn \vee vTrack1\_TOcc)) \tag{5.4}$$

5.4 Comparison with the First Approach

This chapter will be concluded with a few points which highlight the differences between the two approaches.

1. First Approach (Chapter 4): We built an automatic translation tool.
    - Ladder logic specification given by industry. This meant that the specification had already been performed. Our task was to translate the ladder logic into a format in which it could be verified in *SCADE*.

- This approach covered a much larger model. There are many extra components captured by the ladder logic specification which lead to the behaviour of this model being more complicated.

2. Second Approach (Chapter 5): We have invented a new modelling approach which allowed us to also specify and verify the topology.
   - We have formalised reusable components which could be used to specify further segments of railway.
   - Industry wants to get away from ladder logic towards higher level languages with greater expressiveness.
   - We have been able to test another method of modelling using the *SCADE* suite. Modelling components such as the signals using finite state machines has shown us how they perform under composition.

The first approach tested the capabilities of *SCADE* on a pre-existing well defined problem. The model was given by industry and therefore captures a large amount of detail. This differed from the second approach where our problem was not so well defined and a greater number of design decisions had to be taken by the writer. It was not our aim to capture the fine detail of the railway but abstract properties e.g. about the topology, not captured by the low level ladder logic specification.

Chapter 6

Conclusion

We will now conclude this thesis with a summary of the work which has been written about in this document followed by a discussion of possibilities for future work.

6.1 Summary

This thesis contains a detailed feasibility study into the use of the *SCADE* suite for the verification of railway control systems. We began by studying the techniques and theory underlying the model checking component of the *SCADE* suite. This included the brief overview of model checking followed by an in depth discussion of two methods used to decide the validity of propositional formulae. The first being the tautology checking algorithm developed by Stålmarck and the second being binary decision diagrams. This has been followed by two approaches which allowed us to examine the use of the *SCADE* suite for modelling and verification purposes. In the first approach we provided a method to capture the semantics of a ladder logic program as a labelled transition system and produced a tool to translate programs into the *SCADE* language. The approach was successfully applied to two ladder logic programs for real world railway interlockings. These interlockings were then verified using the model checking component of the *SCADE* suite. We were able to draw several conclusions from this:

- *SCADE* is capable of verifying a complex real world system and it does so in a relatively short period of time. It produces counter examples for any falsified safety conditions which can be used to track down the cause of the violation. These factors combined with the fact that it is a commercial tool mean that it is practical for it to be used in industry.
- We were able to explore the functionality of the *SCADE* suite. The model checking component was found to be a so called black box. We were unable to fine tune the model checking process. While this inhibits scientific experimentation it could be an advantage for its use in industry.
- The model checking capabilities of *SCADE* are comparable in terms of speed with those of the previous efforts in Swansea University and an competing academic tool.

We then proceed to explore one possible approach to modelling the railway domain from scratch. This was a new approached based on previous work in the railway modelling area. Individual components of the railway were modelled from scratch and used to form an abstract model of a segment of railway. Verification was then performed in various parts of the new model. The following is a overview of results that can be drawn from the creation and application of this approach:

- The new approach is modular which encourages reuse. Verification of at the component level and using modules can be performed.
- The topology of the railway has been captured and safety conditions have been verified regarding it.
- This makes use of a higher level language which is following the trend of industry.

6.2 Future Work

Finally we will conclude this thesis by looking into some possibilities for future work. These fall into two categories:

- Extending and improving the approaches we have presented in this thesis.
- Exploring the problems presented in this thesis using a new technique.

Identifying and Adding Further Invariants to Our Interlocking Models

The *SCADE* models produced through the translation of ladder logic programs are under specified and do not capture the entirety of the railway domain. Currently each counter example has to be examined in great detail by a team of engineers before the cause of the violation can be deduced. An invariant can be added which extends the model to include the property not currently captured. It would be interesting to see if some form of automated reasoning can be used to help identify likely causes. A tool could then be developed to aid the process of adding invariants.

Extending The Concrete Modelling Approach

This project was a feasibility study and therefore we did not set out to capture the entirety of the model. So far no complete formal specification has been given. Performing the specification using a language such as CASL [ABK+02] could be the topic of future work. This modelling approach is highly abstract and does not capture many of the details of the railway, such as the large variety of signals. This model does not capture any of the behaviour of the railway station which plays a critical role in the behaviour of the first approach. There are also many areas which our model could be expanded to capture properties not treated by either approach in this paper. One such possibility is geographic properties such as distances and model the movement of trains in greater detail.

Applying a Combination of Theorem Proving and Model Checking

In work performed so far at Swansea University mainly model checking techniques have been applied to the railway verification problem. We propose that a combination of first order theorem proving and model checking may be applicable. Some work has already been carried out in the area combining the theorem prover PVS with a BDD based model checker for the mu-calculus [RSS95] and the Symbolic Analysis Laboratory (SAL) [Sha00]. One theorem prover we have considered is the Minlog system [BBS$^+$98]. Minlog is an interactive theorem prover which allows for the creation of constructive proofs and programs to be extracted from such proofs. It would be most elegant to extract a ladder logic program from a proof.

Bibliography

[61103] IEC Standard 61131-3. *Programmable Controllers - Part 3: Programming languages*. IEC, 2003.

[ABK+02] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. Mosses, D. Sannella, and A. Tarlecki. Casl: the common algebraic specification language. *Theoretical Computer Science*, 286(2):153 – 196, 2002.

[ADK+05] Nina Amla, Xiaoqun Du, Andreas Kuehlmann, Robert P. Kurshan, and Kenneth L. McMillan. An analysis of sat-based model checking techniques in an industrial environment. In Dominique Borrione and Wolfgang Paul, editors, *Correct Hardware Design and Verification Methods*, volume 3725 of *Lecture Notes in Computer Science*, pages 254–268. Springer Berlin / Heidelberg, 2005.

[ADS+06] P. Abdulla, J. Deneux, G. Stalmarck, H. Argen, and Ove. Akerlund. Designing Safe, Reliable Systems Using SCADE. *Lecture Notes in Computer Science*, 4313/2006:115–129, 2006.

[Ake78] S. Akers. Binary decision diagrams. *Computers, IEEE Transactions on*, C-27(6):509 –516, 1978.

[BBS+98] H. Benl, U. Berger, H. Schwichtenberg, M. Seisenberger, and W. Zuber. Proof theory at work: Program development in the minlog system. *Automated Deduction - A Basis for Applications, volume II: Systems and Implementation Techniques of Applied Logic Series*, pages 41–71, 1998.

[BC04] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004.

[BCC+99] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *DAC '99: Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 317–320, New York, NY, USA, 1999. ACM.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillian, D. L. Dill, and L. J. Hwang. Symbolic model checking: 1020 states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[Bjö05] M. Björk. A first order extension of stålmarck's method. In Geoff Sutcliffe and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3835 of *Lecture Notes in Computer Science*, pages 276–291. Springer Berlin / Heidelberg, 2005.

[Bjö09] M. Björk. First order stålmarck. *Journal of Automated Reasoning*, 42:99–122, 2009. 10.1007/s10817-008-9115-4.

[Bor98] A. Boralv. Case study: Formal verication of a computerized railway interlocking. *Formal Aspects of Computing*, 10:338 – 360, 1998.

[Bry86] K. L. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, Volume 35, 1986.

[Bus98] S. R. Buss. *Handbook of Proof Theory*, volume 137 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1998.

[BW96] R. Bollig and I. Wegener. *Improving the Variable Ordering of OBDDs is NP-Complete*, volume Volume 45. IEEE Transcations on Computers, 1996.

[CBRZ01] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19:7–34, 2001. 10.1023/A:1011276507260.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[CESS08] K. Claessen, N. Een, M. Sheeran, and N. Sorensson. Sat-solving in practice. In *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, pages 61–67, May 2008.

[CGJ+01] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 176–194, London, UK, 2001. Springer-Verlag.

[CGP99] E. M. Clarke, O. Grumberg, and P. A. Peled. *Model Checking*. MIT Press, 1999.

[CLRS01] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[CMSW99] P. Caspi, C. Mazuet, R. Salem, and D. Weber. Formal design of distributed control systems with Lustre. In *Proc. Safecomp99*, pages 396–409. Springer-Verlag, 1999.

[Cri87] A. Cribbens. Solid-state interlocking (ssi): an integrated electronic signalling system for mainline railways. *Electric Power Applications, IEE Proceedings B*, 134(3):148 –158, 1987.

[CS03] K. Claessen and N. Sörensson. New techniques that improve MACE-style nite model nding. *CADE-19 Workshop on Model Computation, Miami, FL*, 2003.

[DBCB04] S. Dajani-Brown, D. Cofer, and A. Bouali. Formal verification of an avionics sensor voter. 3253:5 – 20, 2004.

[DDM06] B. Dutertre and L. De Moura. The YICES SMT solver. *Tool paper at http://yices.csl. sri. com/tool-paper. pdf*, 2006.

[D90] M. DAgostino. Investigations into the complexity of some propositional calculi, D. Phil. Dissertation. *PRG Technical Monographs 88, Programming Research Group, Oxford University*, 1990.

[EN04] N. Eén and Sörensson N. An extensible sat-solver. *Theory and Applications of Satisfiability Testing, Lecture Notes in Computer Science, Springer Berlin / Heidelberg*, 2919:333–336, 2004.

[Est09] Esterel Technologies. Scade Language Primer. 2009.

[FH98] W. Fokkink and P. Hollingshead. Verification of interlockings: from control tables to ladder logic diagrams. In J. F. Groote, S. P. Luttik, and J. J. van Wamel, editors, *Proceedings of the 3rd Workshop on Formal Methods for Industrial Critical Systems - FMICS'98*, pages 171–185, 1998.

[GvVK95] J.F. Groote, S.F.M. van Vlijmen, and J.W.C. Koorn. The safety guaranteeing system at station Hoorn-Kersenboogerd. *Computer Assurance. COMPASS 95. 'Systems Integrity, Software Safety and Process Security'. Proceedings of the Tenth Annual Conference on*, pages 57 –68, 1995.

[Hag08] G. Hagen. *Verifying safety properties of Lustre programs: an SMT-based approach*. PhD thesis, The University of Iowa, 2008.

[HP99] A. Haxthausen and J. Peleska. Formal development and verification of a distributed railway control system. *IEEE Transactions on Software Engineering*, 26:687–701, 1999.

[HT08] G. Hagen and C. Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *FMCAD '08: Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, pages 1–9, Piscataway, NJ, USA, 2008. IEEE Press.

[Inv] Invensys rail webpage, last accessed in November 2010. http://www.Invensysrail.com.

[Jam10] P. James. SAT-Based Model Checking and its Applications to Train Control Systems. *Swansea University*, 2010.

[JES07] P. Jackson, B. Ellis, and K. Sharp. Using SMT solvers to verify high-integrity programs. In *AFM '07: Proceedings of the second workshop on Automated formal methods*, pages 60–68, New York, NY, USA, 2007. ACM.

[Kan08] K. Kanso. Formal Verification of Ladder Logic. *Swansea University*, 2008.

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

[LT00] P. Letouzey and L. Théry. Formalizing Stålmarck's algorithm in Coq. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1869 of *Lecture Notes in Computer Science*, pages 388–405. Springer Berlin / Heidelberg, 2000. 10.1007/3-540-44659-1_24.

[Mar09] V. M. Marek. *Introduction to Mathematics of Satisfiability*. Studies in Informatics. Chapman and Hall/CRC, 2009.

[McM92] K. L. McMillian. Symbolic model checking: An approach to the state explosion problem. *Carnegie Mellon University, Pittsburgh, PA, USA*, 1992. UMI Order No. GAX92-24209.

[Mey92] B. Meyer. Applying d̈esign by contract: *Computer*, 25(10):40–51, 1992.

[Min] Minisat webpage, last accessed in October 2010. http://minisat.se/Main.html.

[M.S01] M.Schumann, J. Automated Theorem Proving in Software Engineering. 2001.

[Nor01] J. Nordström. Stalmarcks method versus resolution: A comparative theoretical study, 2001.

[Pel08] R. Pelánek. Fighting state space explosion: Review and evaluation. In *In Proc. of Formal Methods for Industrial Critical Systems, FMICS08*, 2008.

[RSS95] S. Rajan, N. Shankar, and M. Srivas. An integration of model checking with automated proof checking. In *Proceedings of the 7th International Conference on Computer Aided Verification*, pages 84–97, London, UK, 1995. Springer-Verlag.

[Sha00] N. Shankar. Combining theorem proving and model checking through symbolic analysis. In Catuscia Palamidessi, editor, *CONCUR 2000 Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin / Heidelberg, 2000.

[Smu69] R. S. Smullyan. *First Order Logic*. Springer-Verlag, Berlin, 1969.

[SS00] M. Sheeran and G. Stålmarck. A Tutorial on Stålmarck's Proof Procedure for Propositional Logic. *Form. Methods Syst. Des.*, 16(1):23–58, 2000.

[SSS00] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In *FMCAD '00: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 108–125, London, UK, 2000. Springer-Verlag.

[Stå94] G. Stålmarck. System for determining propositional logic theorems by applying values and rules to triplets that are generated from boolean formula. *US Patent*, (5,276,897), 1994.

[YBO+98] B. Yang, R. E. Bryant, D. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. Ranjan, and F. Somenzi. *A Performance Study of BDD-Based Model Checking*, volume 1522/1998. Springer Berlin / Heidelberg, 1998.

Appendix A

Proof Systems

A.1 Gentzen's Sequent Calculus

These rules and propositions were taken from Stålmarck [SS00].
Definition 32 (Sequent)**.** *Every line in sequent calculus proof is called a sequent and takes the following form.*

$$A_1, \ldots, A_k \vdash B_1, \ldots, B_l$$

In this text we are using the symbol $\vdash$ *to represent the sequent arrow.* [1] *The following formula captures the meaning of the above sequent.*

$$\bigwedge_{i=1}^{k} A_i \supset \bigvee_{j-1}^{l} B_j$$

Intuitively you can read a sequent as: If the conjunction of all of the A_is is true then one of the B_js in the disjunction must be true.

Definition 33 (Gentzen's Sequent Calculus PK)**.**

Axiom

$$A \vdash A$$

Structural Rules

$$\textit{(Thinning)}\ \frac{\Gamma \vdash \Delta}{\Gamma, \Theta \vdash \Delta, \Lambda} \qquad \textit{(Cut)}\ \frac{\Gamma, A \vdash \Delta \qquad \Gamma \vdash \Delta, A}{\Gamma \vdash \Delta}$$

[1] The sequent arrow is sometimes represented using the symbol $\rightarrow$

Operational Rules

$$(Or_L)\ \frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta} \quad (Or_R)\ \frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \vee B}$$

$$(And_L)\ \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \quad (And_R)\ \frac{\Gamma \vdash \Delta, A \qquad \Gamma \vdash \Delta, B}{\Gamma, \vdash \Delta, A \wedge B}$$

$$(Imp_L)\ \frac{\Gamma \vdash \Delta, A \qquad \Gamma, B \vdash \Delta}{\Gamma, A \to B \vdash \Delta} \quad (Imp_R)\ \frac{\Gamma, A \vdash \Delta, B}{\Gamma \vdash \Delta, A \to B}$$

$$(Neg_L)\ \frac{\Gamma \vdash \Delta, A}{\Gamma, \neg A \vdash \Delta} \quad (Neg_R)\ \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \Delta, \neg A}$$

Proposition 1 (Sub-formula Principle)**.** *If a PK-proof P does not contain any application of the cut rule, then All of the formulas occurring in P must be a sub-formula of some formula in the end sequent of P.*

Having the *sub-formula principle* allowed Stålmarck to place bounds on the size of proofs created by his algorithm.

Proposition 2 (Removing Thinning)**.** *If we allow axioms of the form* $\Gamma, A \vdash A, \Delta$ *then it is possible to remove the thinning rule as it is redundant and is no longer of any use.*

Removing the thinning rule gives us a proof system that is essentially the same as that by Kleene [**?**]. This proof system has the advantage that it is invertible i.e., if a sequent below the line of an inference is valid then the sequents above the line are also valid.

A.2 Smullyan's Semantic Tableaux

The Analytic tableaux form a refutation proof system. You begin a proof by assuming your propositional formula is false. Then a tree is constructed using the rules. The formula is deconstructed into its constituent sub-formulae using the rules *And*, *Not-Or* and *Not-Impl*. Case distinction on the formula is performed by the rules *Or*, *Not-And* and *Impl*. The application of a case distinction rule causes a branch to occur in the proof tree. If each branch of the tree contains a contradiction then the formula is refuted.

Definition 34 (Semantic Tableaux)**.**

$$(And)\ \frac{A \wedge B}{\begin{array}{c} A \\ B \end{array}} \quad (Not - Or)\ \frac{\neg(A \vee B)}{\begin{array}{c} \neg A \\ \neg B \end{array}}$$

$$(Or)\ \frac{A \vee B}{A | B} \quad (Not - And)\ \frac{\neg(A \wedge B)}{\neg A | \neg B}$$

$$(Impl)\ \frac{A \to B}{\neg A | B} \quad (Not - Impl)\ \frac{\neg(A \to B)}{\begin{array}{c} A \\ \neg B \end{array}}$$

$$(Not - Not)\ \frac{\neg\neg A}{A}$$

A.3 Propagation Rules for Stålmarck's Tautology Checker

In the following the proper rules used in Stålmarck's tautology checking algorithm are presented from [Nor01].

Definition 35 (Formula Equivalence Rules)**.**

$$\frac{}{P \equiv P} \tag{A.1}$$

$$\frac{P \equiv Q}{Q \equiv P} \tag{A.2}$$

$$\frac{P \equiv Q \qquad Q \equiv R}{P \equiv R} \tag{A.3}$$

$$\frac{P \equiv \bot}{P' \equiv \top} \tag{A.4}$$

$$\frac{P \equiv Q \qquad Q \equiv \top}{P \equiv \top} \tag{A.5}$$

$$\frac{P \equiv Q \qquad Q \equiv \bot}{P \equiv \bot} \tag{A.6}$$

$$\frac{P \equiv Q}{P' \equiv Q'} \tag{A.7}$$

$$\frac{P \equiv \top}{P' \equiv \bot} \tag{A.8}$$

$$\frac{P \equiv \top \qquad Q \equiv \top}{P \equiv Q} \tag{A.9}$$

$$\frac{P \equiv \bot \qquad Q \equiv \bot}{P \equiv Q} \tag{A.10}$$

$$\frac{P \equiv P'}{\bot} \tag{A.11}$$

Definition 36 (Propagation Rules)**.** *Rules for conjunction*

$$\frac{P \wedge Q \equiv \top}{P \equiv \top} \qquad \frac{P \wedge Q \equiv \top}{Q \equiv \top} \tag{A.12}$$

$$\frac{P \wedge Q \equiv P'}{P \equiv \top} \qquad \frac{P \wedge Q \equiv Q'}{Q \equiv \top} \tag{A.13}$$

$$\frac{P \wedge Q \equiv P'}{Q \equiv \bot} \qquad \frac{P \wedge Q \equiv Q'}{P \equiv \bot} \tag{A.14}$$

$$\frac{P \equiv \top}{P \wedge Q \equiv Q} \qquad \frac{Q \equiv \top}{P \wedge Q \equiv P} \tag{A.15}$$

$$\frac{P \equiv \bot}{P \wedge Q \equiv \bot} \qquad \frac{Q \equiv \bot}{P \wedge Q \equiv \bot} \tag{A.16}$$

$$\frac{P \equiv Q}{P \wedge Q \equiv P} \qquad \frac{P \equiv Q}{P \wedge Q \equiv Q} \tag{A.17}$$

$$\frac{P \equiv Q'}{P \wedge Q \equiv \bot} \tag{A.18}$$

Rules for disjunction

$$\frac{P \vee Q \equiv \bot}{P \equiv \bot} \qquad \frac{P \vee Q \equiv \bot}{Q \equiv \bot} \tag{A.19}$$

$$\frac{P \vee Q \equiv P'}{P \equiv \bot} \qquad \frac{P \vee Q \equiv Q'}{Q \equiv \bot} \tag{A.20}$$

$$\frac{P \vee Q \equiv P'}{Q \equiv \top} \qquad \frac{P \vee Q \equiv Q'}{P \equiv \top} \tag{A.21}$$

$$\frac{P \equiv \top}{P \vee Q \equiv \top} \qquad \frac{Q \equiv \top}{P \vee Q \equiv \top} \tag{A.22}$$

$$\frac{P \equiv Q}{P \vee Q \equiv P} \qquad \frac{P \equiv Q}{P \vee Q \equiv Q} \tag{A.23}$$

$$\frac{P \equiv Q'}{P \vee Q \equiv \top} \tag{A.24}$$

Rules for implication

$$\frac{P \to Q \equiv \bot}{P \equiv \top} \tag{A.25}$$

$$\frac{P \to Q \equiv \bot}{Q \equiv \bot} \tag{A.26}$$

$$\frac{P \to Q \equiv P}{P \equiv \top} \tag{A.27}$$

$$\frac{P \to Q \equiv P}{Q \equiv \top} \tag{A.28}$$

$$\frac{P \to Q \equiv Q'}{P \equiv \bot} \tag{A.29}$$

$$\frac{P \to Q \equiv Q'}{Q \equiv \bot} \tag{A.30}$$

$$\frac{P \equiv \top}{P \to Q \equiv Q} \tag{A.31}$$

$$\frac{Q \equiv \top}{P \to Q \equiv \top} \tag{A.32}$$

$$\frac{P \equiv \bot}{P \to Q \equiv \top} \tag{A.33}$$

$$\frac{Q \equiv \bot}{P \to Q \equiv P'} \tag{A.34}$$

$$\frac{P \equiv Q'}{P \to Q \equiv P'} \qquad \frac{P \equiv Q'}{P \to Q \equiv Q} \tag{A.35}$$

Rules for bi-implication

$$\frac{P \leftrightarrow Q \equiv \top}{P \equiv Q} \tag{A.36}$$

$$\frac{P \leftrightarrow Q \equiv \bot}{P \equiv Q'} \tag{A.37}$$

$$\frac{P \leftrightarrow Q \equiv P}{Q \equiv \top} \qquad \frac{P \leftrightarrow Q \equiv Q}{P \equiv \top} \tag{A.38}$$

$$\frac{P \leftrightarrow Q \equiv P'}{Q \equiv \bot} \qquad \frac{P \leftrightarrow Q \equiv Q'}{P \equiv \bot} \tag{A.39}$$

$$\frac{Q \equiv \top}{P \leftrightarrow Q \equiv P} \qquad \frac{P \equiv \top}{P \leftrightarrow Q \equiv Q} \tag{A.40}$$

$$\frac{Q \equiv \bot}{P \leftrightarrow Q \equiv P'} \qquad \frac{P \equiv \bot}{P \leftrightarrow Q \equiv Q'} \tag{A.41}$$

$$\frac{P \equiv Q}{P \leftrightarrow Q \equiv \top} \tag{A.42}$$

$$\frac{P \equiv Q'}{P \leftrightarrow Q \equiv \bot} \tag{A.43}$$

Appendix B

Concrete Railway Model

In this chapter we will present the work produced in attempt to provide a concrete model of the railway (See Chapter 5).

B.1 Railway Components

In the following we have tried to model components of the railway with the intention that they could be verified individually and then recombined into different configurations.

Track Segment 1: Node was used to model a straight piece of track. We have simplified the topological aspect somewhat since the track is generally set up for trains travelling in one direction. We have assumed that a train will not travel the wrong way down a track.

```
node Track_Segment1(TrainIn, RedLight, GreenLight, WhiteLight : bool)
returns(TrackOccupied, TrainOut, Crash: bool)
var



let

automaton
initial state EMPTY

let

TrackOccupied = false;
TrainOut = false;
Crash = false;

tel
until
if (TrainIn) restart OCCUPIED;
```

```
state OCCUPIED
let
TrackOccupied = true ;
TrainOut = false;
Crash = false;


tel
until
if ((GreenLight or WhiteLight) and not RedLight) restart TRAINLEAVE;
if (TrainIn) restart CRASH;

state TRAINLEAVE
let

TrackOccupied = true;
TrainOut = true;
Crash = false;

tel
until
if (TrainIn) restart CRASH;
if (TrainOut) restart EMPTY;

state CRASH
let

TrackOccupied = true;
Crash = true;
TrainOut = false;

tel

returns .. ;

tel
```

Track Segment 2: Originally we had modelled certain junctions using this component and of track using this component. we decided however that we would like to capture all possible ways a train can move across a junction we therefore converted all track segments in junctions to using Track Segment 3.

```
node Track_Segment2(TrainIn1, TrainIn2, RedLight, GreenLight,
                    WhiteLight, PointsNorm, PointsRev : bool)
returns(TrackOccupied, TrainOut1, TrainOut2, Crash: bool)
var
```

```
let


automaton
initial state EMPTY
let

TrackOccupied = false;
TrainOut1 = false;
TrainOut2 = false;
Crash = false;

tel
until
if (TrainIn1 or TrainIn2) restart OCCUPIED;

state OCCUPIED
let
TrackOccupied = true;
TrainOut1 = false;
TrainOut2 = false;
Crash = false;


tel
until
if ((GreenLight or WhiteLight) and not RedLight) restart TRAINLEAVE;
if (TrainIn1 or TrainIn2) restart CRASH;

state TRAINLEAVE
let

TrackOccupied = true;
TrainOut1 =  if (PointsRev) then true else false;
TrainOut2 = if (PointsNorm) then true else false;
Crash = false;

tel
until
if (TrainIn1 or TrainIn2) restart CRASH;
if (TrainOut1 or TrainOut2) restart EMPTY;

state CRASH
let
```

```
TrackOccupied = true;
Crash = true;
TrainOut1 = false;
TrainOut2 = false;
tel

returns .. ;

tel
```

Track Segment 3: This is the node that was used to model junctions in the track. The just has been modelled in such a way that the direction of travel along with the points dictate how a train exits the track.

```
node Track_Segment3(TrainIn1, TrainIn2, TrainIn3, RedLight,
GreenLight, WhiteLight, PointsNorm , PointsRev : bool)
returns(TrackOccupied, TrainOut1, TrainOut2, TrainOut3 , Crash: bool)
var

Direction : bool;

let

Direction = false -> if (TrainIn1) then true else (if (TrainIn3) then false
else pre Direction);

automaton
initial state EMPTY
let

TrackOccupied = false;
TrainOut1 = false;
TrainOut2 = false;
TrainOut3 = false;

Crash = false;

tel
until
if (TrainIn1 or TrainIn2 or TrainIn3) restart OCCUPIED;

state OCCUPIED
let
TrackOccupied = true ;
TrainOut1 = false;
Crash = false;
```

```
TrainOut2 = false;
TrainOut3 = false;


tel
until
if ((GreenLight or WhiteLight) and not RedLight) restart TRAINLEAVE;
if (TrainIn1 or TrainIn2 or TrainIn3) restart CRASH;

state TRAINLEAVE
let

TrackOccupied = true;
TrainOut1 = if (Direction) then false else true ;
Crash = false;
TrainOut2 = if (Direction and PointsRev) then true else false ;
TrainOut3 = if (Direction and PointsNorm) then true else false;

tel
until
if (TrainIn1 or TrainIn2 or TrainIn3) restart CRASH;
if (TrainOut1 or TrainOut2 or TrainOut3) restart EMPTY;

state CRASH
let

TrackOccupied = true;
Crash = true;
TrainOut1 = false;
TrainOut2 = false;
TrainOut3 = false;

tel

returns .. ;

tel



node Route(RouteCall, RouteSet, PointsLocked,
 LightsSet, W_track_clear, G_track_clear: bool)
returns(RouteSelected, DrivePL, DriveG, DriveW, DriveR: bool)
let
automaton
initial state STATE1
unless
```

```
if (RouteCall) restart STATE2;
let

RouteSelected = false;
DrivePL = false;
DriveG = false;
DriveW = false;
DriveR = true;

tel
state STATE2
unless
if (not RouteSet) restart STATE1;
if (RouteCall and PointsLocked and LightsSet) restart STATE3;
let


RouteSelected = false;
DrivePL = true;
DriveG = if (W_track_clear and G_track_clear)
then true
else false;
DriveW = if (W_track_clear and not G_track_clear)
then true
else false;
DriveR = if ( (W_track_clear and G_track_clear)
                      or (W_track_clear and not G_track_clear))
then false
else true;


tel
state STATE3
unless
if (not RouteSet) restart STATE1;
let

RouteSelected = true;
DrivePL = true;
DriveG = if (W_track_clear and G_track_clear)
then true
else false;
DriveW = if (W_track_clear and not G_track_clear)
then true
else false;
DriveR = if ( (W_track_clear and G_track_clear)
                      or (W_track_clear and not G_track_clear))
```

```
then false
else true;
tel
returns .. ;


tel
```

The following is the *SCADE* node that was used to model a point. It contains a finite state machine that models the 4 possible states a point can be in: Normal and free , Normal and locked, Reverse and free, Reverse and locked. One further state that could be added in future is the Unknown state. This is where the point is in neither Reverse or Normal but in some indeterminate state.

```
node Point(Normal, Reverse, Occupied : bool)
returns(NLock, NFree, RLock, RFree: bool)
let

automaton
  initial state NORMAL_FREE
unless
if (Occupied) restart NORMAL_LOCK;

if (Reverse and not Normal and not Occupied) restart REVERSE_FREE;
let
NLock = false;
RLock = false;
NFree = true ;
RFree = false ;
tel
until


state NORMAL_LOCK
unless
if (not Occupied) restart NORMAL_FREE;
let
NLock = true;
RLock = false;
NFree = false ;
RFree = false ;

tel

state REVERSE_FREE
unless
if (Occupied) restart REVERSE_LOCK;
```

```
if (not Reverse and Normal and not Occupied) restart NORMAL_FREE;
let

NLock = false;
RLock = false;
NFree = false;
RFree = true ;



tel



state REVERSE_LOCK
unless
if (not Occupied) restart REVERSE_FREE;
let
NLock = false;
RLock = true;
NFree = false;
RFree = false;


tel

returns .. ;

tel
```

Pointif is a model of a point using if statements instead of the finite state machines.

```
node Pointif(Normal, Reverse, Occupied : bool)
returns(NLock, NFree, RLock, RFree: bool)
let


NLock =  (Occupied) ->
 ( (pre NFree and Occupied) or (pre NLock and Occupied));

RLock = false ->  (pre RFree or pre RLock) and Occupied ;
NFree = ( (Normal and not Reverse) or (not Normal and not Reverse)
                                   or (Normal and Reverse)) and not Occupied
 -> ( ((pre RFree and not Reverse and Normal) or (pre NFree and
 (not Reverse or (Reverse and Normal))or pre NLock)) and not Occupied);

RFree =  (not Occupied and (Reverse and not Normal)) ->
```

```
 ( (pre NFree and not Normal and Reverse and not Occupied ) or
( (pre RFree and (not Normal and (notOccupied or not Reverse)
                                or (Normal and Reverse)) )  and not Occupied)
or (pre RLock and not Occupied)) ;


tel


node PointEquiv(Normal, Reverse, Occupied: bool)
returns(Equivalent , NLock1, NFree1, RLock1, RFree1,
              NLock2, NFree2, RLock2, RFree2 : bool)



let


NLock1, NFree1, RLock1, RFree1 =  Point(Normal, Reverse, Occupied);
NLock2, NFree2, RLock2, RFree2 =  Pointif(Normal, Reverse, Occupied);

Equivalent =((NLock1 and NLock2) or (not NLock1 and not NLock2)) and
((NFree1 and NFree2) or (not NFree1 and not NFree2)) and
((RFree1 and RFree2) or (not RFree1 and not RFree2)) and
((RLock1 and RLock2) or (not RLock1 and not RLock2));

tel


node Point2(Normal, Reverse, Occupied : bool)
returns(NLock, NFree, RLock, RFree: bool)
let

automaton
  initial state INITIAL
  unless
   if (false -> Occupied) restart NORMAL_LOCK;
if (false -> Normal) restart NORMAL_FREE;
if (false -> Reverse) restart REVERSE_FREE;
if (false -> not Reverse and not  Normal and not Occupied) restart NORMAL_FREE;
let

NLock = false;
RLock = false;
NFree = true ;
RFree = false ;

tel
```

```
until


state NORMAL_FREE
unless
if (Occupied) restart NORMAL_LOCK;

if (Reverse and not Normal and not Occupied) restart REVERSE_FREE;
let
NLock = false;
RLock = false;
NFree = true ;
RFree = false ;
tel
until


state NORMAL_LOCK
unless
if (not Occupied) restart NORMAL_FREE;
let
NLock = true;
RLock = false;
NFree = false ;
RFree = false ;

tel

state REVERSE_FREE
unless
if (Occupied) restart REVERSE_LOCK;
if (not Reverse and Normal and not Occupied) restart NORMAL_FREE;
let

NLock = false;
RLock = false;
NFree = false;
RFree = true ;



tel



state REVERSE_LOCK
unless
```

```
if (not Occupied) restart REVERSE_FREE;
let
NLock = false;
RLock = true;
NFree = false;
RFree = false;


tel

returns .. ;
tel


node Pointif2(Normal, Reverse, Occupied : bool)
returns(NLock, NFree, RLock, RFree: bool)
let


NLock =  false ->
 ( (pre NFree and Occupied) or (pre NLock and Occupied));

RLock = false ->  (pre RFree or pre RLock) and Occupied ;
NFree = true
 -> ( ((pre RFree and not Reverse and Normal) or (pre NFree and
 (not Reverse or (Reverse and Normal))or pre NLock)) and not Occupied);

RFree =  false ->
 ( (pre NFree and not Normal and Reverse and not Occupied ) or
( (pre RFree and (not Normal and
(not Occupied or not Reverse) or (Normal and Reverse)) )  and not Occupied)
or (pre RLock and not Occupied)) ;
tel


node PointEquiv2(Normal, Reverse, Occupied: bool)
returns(Equivalent, Equivalent2, Equivalent3, NLock1,
 NFree1, RLock1, RFree1, NLock2, NFree2,
 RLock2, RFree2, NLock3, NFree3, RLock3, RFree3,
NLock4, NFree4, RLock4, RFree4 : bool)
let


NLock1, NFree1, RLock1, RFree1 =  Point2(Normal, Reverse, Occupied);
NLock2, NFree2, RLock2, RFree2 =  Pointif2(Normal, Reverse, Occupied);
NLock3, NFree3, RLock3, RFree3  =  Point(Normal, Reverse, Occupied);
NLock4, NFree4, RLock4, RFree4  =  Pointif(Normal, Reverse, Occupied);
```

```
Equivalent =((NLock1 and NLock2) or (not NLock1 and not NLock2)) and
((NFree1 and NFree2) or (not NFree1 and not NFree2)) and
((RFree1 and RFree2) or (not RFree1 and not RFree2)) and
((RLock1 and RLock2) or (not RLock1 and not RLock2));

Equivalent2 = true -> ((NLock1 and NLock3) or (not NLock1 and not NLock3)) and
((NFree1 and NFree3) or (not NFree1 and not NFree3)) and
((RFree1 and RFree3) or (not RFree1 and not RFree3)) and
((RLock1 and RLock3) or (not RLock1 and not RLock3));

Equivalent3 = true -> ((NLock2 and NLock4) or (not NLock2 and not NLock4)) and
((NFree2 and NFree4) or (not NFree2 and not NFree4)) and
((RFree2 and RFree4) or (not RFree2 and not RFree4)) and
((RLock2 and RLock4) or (not RLock2 and not RLock4));


tel
```

B.2 Signals

Signals were modelled using the finite state machines in *SCADE*. The signals and the aspects they contain were modelled separately. A signal can be though of as a device which controls the aspects it contains. They have an Initial state in which the the red aspect is driven. Subsequent states depend on the value of the inputs.

```
node Light3Aspect(Red, White , Green: bool)
returns(LightsSet, R_O_D , W_O_D, G_O_D

: bool)
var
G_O_A,  G_O_R, W_O_A, W_O_R, R_O_A, R_O_R,
G_I_A, G_I_D, G_I_R, W_I_A, W_I_D, W_I_R, R_I_A, R_I_D,  R_I_R,
G_S_A, G_S_D, G_S_R, W_S_A, W_S_D, W_S_R, R_S_A, R_S_D,  R_S_R : bool;

let

G_S_A = false -> (pre G_I_A);
G_S_D = false -> (pre G_I_D);
G_S_R = false -> (pre G_I_R);
W_S_A = false -> (pre W_I_A);
W_S_D = false -> (pre W_I_D);
W_S_R = false -> (pre W_I_R);
```

```
R_S_A = false -> (pre R_I_A);
R_S_D = false -> (pre R_I_D);
R_S_R = false -> (pre R_I_R);

automaton
initial state INITIALISE
let
G_I_A = false;
G_I_D = false;
G_I_R = false;
W_I_A = false;
W_I_D = false;
W_I_R = false;
R_I_A = true;
R_I_D = true;
R_I_R = false;
LightsSet = false;

tel
until
if (R_O_A and R_O_D and Red) resume RED;
if (R_O_A and R_O_D and White) resume WHITE;
if (R_O_A and R_O_D and Green) resume GREEN;
state RED

let
automaton
initial state SETAVAIL
unless
if (R_S_A and not R_S_D) resume TRANSITIONSTATE;
let
G_I_A = G_S_A;
G_I_D = G_S_D;
G_I_R = G_S_R;
W_I_A = W_S_A;
W_I_D = W_S_D;
W_I_R = W_S_R;
R_I_A = true;
R_I_D = R_S_D;
R_I_R = R_S_R;
LightsSet = false;
tel

state TRANSITIONSTATE
unless
if (R_S_A and R_S_D) resume LIGHTSSET;
let
```

```
G_I_A = G_S_A;
G_I_D = false;
G_I_R = G_S_R;
W_I_A = W_S_A;
W_I_D = false;
W_I_R = W_S_R;
R_I_A = true;
R_I_D = true;
R_I_R = R_S_R;
LightsSet = false;


tel
state LIGHTSSET
let
G_I_A = false;
G_I_D = false;
G_I_R = G_S_R;
W_I_A = false;
W_I_D = false;
W_I_R = W_S_R;
R_I_A = true;
R_I_D = true;
R_I_R = R_S_R;
LightsSet = true;
tel

returns .. ;

tel
until
if (Green and LightsSet) resume GREEN;
if (White and LightsSet) resume WHITE;

state WHITE
let

automaton
initial state SETAVAIL
unless
if (W_S_A and not W_S_D) resume TRANSITIONSTATE;

let
G_I_A = G_S_A;
G_I_D = G_S_D;
G_I_R = G_S_R;
W_I_A = true;
```

```
W_I_D = G_S_D;
W_I_R = W_S_R;
R_I_A = R_S_A;
R_I_D = R_S_D;
R_I_R = R_S_R;
LightsSet = false;
tel
state TRANSITIONSTATE
unless
if (R_S_A and R_S_D) resume LIGHTSSET;
let
G_I_A = G_S_A;
G_I_D = false;
G_I_R = G_S_R;
W_I_A = true;
W_I_D = true;
W_I_R = W_S_R;
R_I_A = R_S_A;
R_I_D = false;
R_I_R = R_S_R;
LightsSet = false;


tel
state LIGHTSSET
let
G_I_A = false;
G_I_D = false;
G_I_R = G_S_R;
W_I_A = true;
W_I_D = true;
W_I_R = W_S_R;
R_I_A = false;
R_I_D = false;
R_I_R = R_S_R;
LightsSet = true;
tel

returns .. ;

tel
until
if ((Red and LightsSet)or (not Red and not White
            and not Green and LightsSet)) resume RED;
if (Green and LightsSet) resume GREEN;

state GREEN
```

```
let

automaton
initial state SETAVAIL
unless
if (R_S_A and not R_S_D) resume TRANSITIONSTATE;
let
G_I_A = true;
G_I_D = G_S_D;
G_I_R = G_S_R;
W_I_A = W_S_A;
W_I_D = W_S_D;
W_I_R = W_S_R;
R_I_A = R_S_A;
R_I_D = R_S_D;
R_I_R = R_S_R;
LightsSet = false;
tel

state TRANSITIONSTATE
unless
if (R_S_A and R_S_D) resume LIGHTSSET;
let
G_I_A = true;
G_I_D = true;
G_I_R = G_S_R;
W_I_A = W_S_A;
W_I_D = false;
W_I_R = W_S_R;
R_I_A = R_S_A;
R_I_D = false;
R_I_R = R_S_R;
LightsSet = false;


tel
state LIGHTSSET
let
G_I_A = true;
G_I_D = true;
G_I_R = G_S_R;
W_I_A = false;
W_I_D = false;
W_I_R = W_S_R;
R_I_A = false;
R_I_D = false;
R_I_R = R_S_R;
```

```
LightsSet = true;
tel

returns .. ;



tel
until
if ((Red and LightsSet) or
          (not Red and not White and not Green and LightsSet )) resume RED;
if (White and LightsSet) resume WHITE;
returns .. ;

G_O_A, G_O_D, G_O_R = SignalAspect(G_I_A , G_I_D , G_I_R);
W_O_A, W_O_D, W_O_R = SignalAspect(W_I_A , W_I_D , W_I_R);
R_O_A, R_O_D, R_O_R = SignalAspect(R_I_A , R_I_D , R_I_R);


-- Safety Conditions for a 3 Aspect Signal
-- ConLight = not (G_O_D and W_O_D  or
       -- G_O_D and R_O_D or W_O_D and R_O_D);
--  Onelight =  G_O_D or W_O_D or R_O_D;

tel


node Light2Aspect(Red,Green: bool)
returns(LightsSet, R_O_D, G_O_D

: bool)
var
G_O_A, G_O_R, R_O_A, R_O_R,
G_I_A, G_I_D, G_I_R, R_I_A, R_I_D,  R_I_R,
G_S_A, G_S_D, G_S_R, R_S_A, R_S_D,  R_S_R : bool;

let


G_S_A = false -> (pre G_I_A);
G_S_D = false -> (pre G_I_D);
G_S_R = false -> (pre G_I_R);

R_S_A = false -> (pre R_I_A);
R_S_D = false -> (pre R_I_D);
R_S_R = false -> (pre R_I_R);
```

```
automaton
initial state INITIALISE
let
G_I_A = false;
G_I_D = false;
G_I_R = false;
R_I_A = true;
R_I_D = true;
R_I_R = false;
LightsSet = false;

tel
until
if (R_O_A and R_O_D and Red) resume RED;
if (R_O_A and R_O_D and Green) resume GREEN;
state RED

let
automaton
initial state SETAVAIL
unless
if (R_S_A and not R_S_D) resume TRANSITIONSTATE;
let
G_I_A = G_S_A;
G_I_D = G_S_D;
G_I_R = G_S_R;
R_I_A = true;
R_I_D = R_S_D;
R_I_R = R_S_R;
LightsSet = false;
tel

state TRANSITIONSTATE
unless
if (R_S_A and R_S_D) resume LIGHTSSET;
let
G_I_A = G_S_A;
G_I_D = false;
G_I_R = G_S_R;
R_I_A = true;
R_I_D = true;
R_I_R = R_S_R;
LightsSet = false;


tel
state LIGHTSSET
```

```
let
G_I_A = false;
G_I_D = false;
G_I_R = G_S_R;
R_I_A = true;
R_I_D = true;
R_I_R = R_S_R;
LightsSet = true;
tel

returns .. ;

tel
until
if (Green and LightsSet) resume GREEN;
state GREEN
let

automaton
initial state SETAVAIL
unless
if (R_S_A and not R_S_D) resume TRANSITIONSTATE;
let
G_I_A = true;
G_I_D = G_S_D;
G_I_R = G_S_R;
R_I_A = R_S_A;
R_I_D = R_S_D;
R_I_R = R_S_R;
LightsSet = false;
tel

state TRANSITIONSTATE
unless
if (R_S_A and R_S_D) resume LIGHTSSET;
let
G_I_A = true;
G_I_D = true;
G_I_R = G_S_R;
R_I_A = R_S_A;
R_I_D = false;
R_I_R = R_S_R;
LightsSet = false;


tel
state LIGHTSSET
```

```
let
G_I_A = true;
G_I_D = true;
G_I_R = G_S_R;
R_I_A = false;
R_I_D = false;
R_I_R = R_S_R;
LightsSet = true;
tel

returns .. ;



tel
until
if ((Red and LightsSet) or
          (not Red and not Green and LightsSet )) resume RED;
returns .. ;

G_O_A, G_O_D, G_O_R = SignalAspect(G_I_A , G_I_D , G_I_R);
R_O_A, R_O_D, R_O_R = SignalAspect(R_I_A , R_I_D , R_I_R);

--  ConLight = not (G_O_D and R_O_D);
--   Onelight =  G_O_D or R_O_D;
```

Currently the fixed red constantly outputs a boolean stream containing the value true. Further behaviour could be modelled at a later stage such as failure and reporting.

```
node FixedRed()
returns(Red : bool)
var

let

Red = true;


tel
```

The following is the *SCADE* model for a signal aspect.

```
node SignalAspect(a,d,r : bool) returns (Avail, Driven, Report: bool)
let



automaton
initial state STATE_1
```

```
unless
if (not a) restart STATE_5;
if (d) restart STATE_2;
if (r) restart STATE_4;
let

Avail = true;
Driven = false;
Report = false;

tel

state STATE_2
unless
if (not d) restart STATE_1;
if (not a) restart STATE_6;
if (r) restart STATE_3;
let

Avail = true;
Driven = true;
Report = false;


tel


state STATE_3
unless
if (not a) restart STATE_8;
if (not d) restart STATE_4;
if (not r) restart STATE_2;
let

Avail = true;
Driven = true;
Report = true;


tel

state STATE_4
unless
if (not a) restart STATE_7;
if (d) restart STATE_3;
if (not r) restart STATE_1;
let
```

```
Avail = true;
Driven = false;
Report = true;

tel

state STATE_5
unless
if (a) restart STATE_1;
if (r) restart STATE_7;
let


Avail = false;
Driven = false;
Report = false;



tel

state STATE_6
unless
if (a) restart STATE_2;
if (r) restart STATE_8;
let

Avail = false;
Driven = true;
Report = false;

tel

state STATE_7
unless
if (a) restart STATE_4;
if (not r) restart STATE_5;
let

Avail = false;
Driven = false;
Report = true;

tel
```

```
state STATE_8
unless
if (a) restart STATE_3;
if (not r) restart STATE_6;
let


Avail = false;
Driven = true;
Report = true;


tel
returns .. ;

-- AspectSafe = true -> Avail or pre Avail or
               (not Avail and not pre Avail and
               (not Driven  and not pre Driven) or (Driven and pre Driven));


tel
```

B.3 Route Controller

```
node RouteController(v1204_1_R, v1204_2_R, v1206_R, v1205_R, v1203_R, v1201_R,
 v1204_1_RS, v1204_2_RS, v1206_RS, v1205_RS, v1203_RS, v1201_RS : bool)
returns (v1204_1_A, v1204_2_A, v1206_A, v1205_A, v1203_A , v1201_A,
v1204_1_C, v1204_2_C, v1206_C, v1205_C, v1203_C, v1201_C,
v1204_1_S, v1204_2_S, v1206_S, v1205_S, v1203_S, v1201_S,
ConNP1, ConNP2, ConNP3, ConNP4 , ConNP5
: bool)

let


-- automaton for conflicting routes

automaton
initial state INIT
unless
    if (v1204_1_R) restart ROUTESET2;
if (v1204_2_R) restart ROUTESET1;
if (v1206_R) restart ROUTESET1;
if (v1205_R) restart ROUTESET3;
```

```
let
-- All routes are available initially
v1204_1_A = true ;
v1204_2_A = true ;
v1206_A = true ;
v1205_A = true ;

-- No routes have been called in the intial state

v1204_1_C = false ;
v1204_2_C = false ;
v1206_C = false ;
v1205_C = false ;

-- No routes are selected in the intial state

v1204_1_S = false ;
v1204_2_S = false ;
v1206_S = false ;
v1205_S = false ;

tel


state ROUTESET1
unless
if (not v1206_R  and not v1204_1_R) restart INIT;
let
--
v1204_1_A = if (v1204_2_R) then true else false;
v1204_2_A = false ;
v1206_A = if (v1206_R) then true else false;
v1205_A = false ;

-- The requested route is called
v1204_1_C = if (v1204_2_R) then true else false;
v1204_2_C = false ;
v1206_C = if (v1206_R) then true else false;
v1205_C = false ;


-- if statement for 1206

       v1206_S = if (not v1206_RS) then false else true;

-- if statement for 1204_2
```

```
v1204_1_S = if (not v1204_2_RS) then false else true;

v1204_2_S = false ;
v1205_S = false ;

tel

state ROUTESET2
unless
if (not v1204_2_R) restart INIT;
let

v1204_1_A = false ;
v1204_2_A = true ;
v1206_A = false ;
v1205_A = false ;

-- The requested route is called

v1204_1_C = false ;
v1204_2_C = true ;
v1206_C = false ;
v1205_C = false ;

-- automaton for 1204_1


automaton
initial state NOT_SEL
unless
if (v1204_2_RS) restart SEL;
let

v1204_2_S = false;

tel
state SEL
let

v1204_2_S = true;


tel
returns .. ;
```

```
v1206_S = false;
v1204_1_S = false;
v1205_S = false;


tel

state ROUTESET3
unless
if (not v1205_R) restart INIT;
let

v1204_1_A = false ;
v1204_2_A = false ;
v1206_A = false ;
v1205_A = true ;

-- The requested route is called

v1204_1_C = false ;
v1204_2_C = false ;
v1206_C = false ;
v1205_C = true ;

-- automaton for 1205

automaton
initial state NOT_SEL
unless
if (v1205_RS) restart SEL;
let

v1205_S = false;

tel
state SEL
let

v1205_S = true;


tel
returns .. ;

v1204_1_S = false;
v1204_2_S = false;
```

```
v1206_S = false;

tel
returns .. ;

--- Routes with no conflicts are always available



v1203_A = true;
v1201_A = true;

-- flows for for route 1203




v1203_C = if (v1203_R) then true else false;
v1203_S = if (v1203_RS) then true else false;




-- flows for route 1201



v1201_C = if(v1201_R) then true else false;
v1201_S = if (v1201_RS) then true else false;


--Safety Conditions for Conflicting Routes

ConNP1 = not( v1204_1_S and v1205_S);
ConNP2 = not( v1204_1_S and v1204_2_S);
ConNP3 = not( v1204_2_S and v1205_S);
ConNP4 = not( v1204_2_S and v1206_S);
ConNP5 = not( v1206_S and v1205_S);


tel
```

B.4 Railway Segment Model

The following contains the *SCADE* model for concrete railway example. Below is a list of variable suffixes.

- Track Segments
  - TO : Track Occupied
  - O : Train Out
  - CR : Crash
- Signals
  - RED : Red aspect is showing on the signal
  - GRE : Green aspect is showing on the signal
  - WHI : White aspect is showing on the signal
  - LS : Lights set
- – Points
  - NL : Points locked normal
  - RL : Points locked reverse
  - NF : Points free and normal
  - RF : Points Free and reverse
- Routes
  - R: Route Requested
  - RS : Route Selected
  - RC : Route Called
  - S : Route Set
  - A : Available
  - DP : Drive points , caused by a route being called.
  - DR : Drive Red
  - DG :
  - DW :
  - WTC : The track segments required for a white light are clear.
  - GTC : The track segments required for a green light are clear.

Each track component has a unique identifier.

Component Type	Component Identifiers
Track	1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008, 1009, 1010, 1011
Points	1101, 1102, 1103, 1104
Signals	1201, 1202, 1203, 1204, 1205, 1206, 1207 , 1208, 1209, 1210

```
node AbstractRailway(v1204_1_R, v1204_2_R, v1206_R , v1205_R,
                           v1203_R, v1201_R, TrainIn : bool)

returns(sPLRS_1206, sPLRS_1204_1, sPLRS_1204_2,
sPLRS_1205,  sNPA_1206, sNPA_1204, sNPA_1205, sNPATR_1206,
       sNPATR_1204_1, sNPATR_1204_2, sNPATR_1205, nocrash, sTI1001  : bool)
var
v1205_WTC, v1204_1_WTC, v1204_2_WTC, v1206_WTC, v1203_WTC , v1201_WTC : bool;
v1205_GTC, v1204_1_GTC, v1204_2_GTC, v1206_GTC, v1203_GTC , v1201_GTC : bool;

 v1204_1_A, v1204_2_A,
 v1206_A, v1205_A, v1204_1_S, v1204_2_S, v1206_S, v1205_S ,
v1101_NF , v1101_NL , v1101_RF, v1101_RL, v1102_NF , v1102_NL ,
    v1102_RF, v1102_RL ,v1103_NF, v1103_NL, v1103_RF , v1103_RL,
v1104_NF, v1104_NL, v1104_RF , v1104_RL,

v1204_1_RS, v1204_1_DP, v1204_1_DG , v1204_1_DW , v1204_1_DR,
v1204_2_RS , v1204_2_DP, v1204_2_DG , v1204_2_DW , v1204_2_DR,
v1206_RS , v1206_DP, v1206_DG, v1206_DW, v1206_DR,
v1205_RS , v1205_DP, v1205_DG , v1205_DW , v1205_DR,
v1203_RS , v1203_DP, v1203_DG, v1203_DW, v1203_DR,
v1201_RS , v1201_DP, v1201_DG, v1201_DW, v1201_DR,
v1204_2_RC, v1204_LS, v1206_LS, v1205_RC, v1205_LS, v1203_RC,
v1203_LS, v1201_RC, v1201_LS,  v1201_RED, v1203_RED, v1204_RED,
v1205_RED, v1206_RED, v1204_1_RC, v1206_RC, v1203_S,
    v1201_S, v1203_A,  v1201_A, v1201_GRE, v1201_WHI,
v1001_TO , v1001_O, v1002_TO , v1002_O, v1003_TO, v1003_O,
v1204_GRE, v1004_TO , v1004_O_2,
v1006_TO, v1006_O, v1205_GRE,  v1005_TO, v1005_O_2,
v1007_TO, v1007_O, v1008_TO, v1008_O, v1009_TO, v1009_O,
v1010_TO, v1010_O_2 , v1010_O_3, v1011_TO, v1011_O,
v1009_O_2, v1004_O,  v1010_O, v1005_O, v1203_WHI, v1203_GRE, v1206_GRE,
v8255_1_D, v1208_D, v1209_D, v8257_2_D ,
      v1001_CR, v1002_CR, v1003_CR, v1004_CR, v1011_CR, v1005_CR,
v1006_CR,  v1010_CR, v1009_CR, v1008_CR, v1007_CR , v1005_O_3, v1004_O_3, v1009_O_3


: bool;
let
```

```
-- East Bound Line


-- Track 1001
v1001_TO , v1001_O, v1001_CR =
      Track_Segment1( TrainIn, v1201_RED, v1201_GRE, v1201_WHI );

-- Track 1002
-- Since there is no light on this segment
-- any train can proceed to the next piece of track
-- hence green and white are set to true untill i find a way of modelling this.

v1002_TO , v1002_O, v1002_CR =
      Track_Segment1(v1001_O , false, true , true);

-- Track 1003
v1003_TO, v1003_O, v1003_CR =
     Track_Segment1(v1002_O, v1204_RED, v1204_GRE, false);

-- Track 1004
v1004_TO, v1004_O , v1004_O_2, v1004_O_3, v1004_CR =
  Track_Segment3(v1003_O, v1009_O_2 , v1005_O_3,
                   false, true, false, v1101_NL , v1101_RL);

-- Track 1005
v1005_TO, v1005_O, v1005_O_2 , v1005_O_3, v1005_CR =
      Track_Segment3(v1006_O, false, v1004_O_3,
                  false , true, false , v1103_NL , v1103_RL);

-- Track 1006
v1006_TO, v1006_O, v1006_CR = Track_Segment1(v1005_O, v1205_RED, v1205_GRE, false);

-- West Bound Line
-- Track 1007
v1007_TO, v1007_O, v1007_CR = Track_Segment1(v1008_O, false, true, false);

-- Track 8002
-- Unclear if this track segment is part of the scheme.
-- v8002_TO, v8002_O, v8002_CR =
-- Track_Segment1(v1007_O, v1202_RED , v1202_GRE, v1202_WHI);

-- Track 1008
v1008_TO, v1008_O, v1008_CR =
      Track_Segment1(v1009_O_3, v1203_RED, v1203_GRE , v1203_WHI);

-- Track 1009
```

```
v1009_TO, v1009_O, v1009_O_2 , v1009_O_3, v1009_CR =
       Track_Segment3( v1010_O, v1004_O_2,
       false ,  false , true, false, v1102_NL , v1102_RL);

-- Track 1010
v1010_TO, v1010_O, v1010_O_2 , v1010_O_3, v1010_CR =
       Track_Segment3(v1009_O, v1005_O_2,
       v1011_O , false, true, false, v1104_NL, v1104_RL);

-- Track 1011
v1011_TO, v1011_O, v1011_CR  = Track_Segment1(v1010_O_3, v1206_RED, v1206_GRE, false);


-- Points
-- Left Points 1101/2

v1101_NF , v1101_NL , v1101_RF, v1101_RL =
      Point( (v1204_1_DP or v1205_DP or v1205_DP), v1204_2_DP, v1004_TO );

      v1102_NF , v1102_NL , v1102_RF, v1102_RL =
  Point( (v1204_1_DP or v1205_DP or v1205_DP), v1204_2_DP, v1009_TO );

-- Right Points 1103/4


v1103_NF, v1103_NL, v1103_RF , v1103_RL =
      Point((v1204_1_DP or v1204_2_DP or v1206_DP), v1205_DP, v1005_TO);

      v1104_NF, v1104_NL, v1104_RF , v1104_RL =
      Point((v1204_1_DP or v1204_2_DP or v1206_DP), v1205_DP,  v1010_TO);



-- Routes
-- Route 1204(1) (Ends 8255)
-- Locks Points Normal
-- 1101/2
-- 1103/4
-- Tracks Clear Green : 1004, 1005 , 1006
-- Tracks Clear White

v1204_1_WTC = false;
v1204_1_GTC = v1004_TO and v1005_TO and v1006_TO;
v1204_1_RS, v1204_1_DP, v1204_1_DG , v1204_1_DW , v1204_1_DR =
       Route(v1204_1_RC, (false -> pre v1204_1_S),
      (false -> pre v1101_NL and pre v1102_NL and pre v1103_NL and pre v1104_NL),
       v1204_LS, v1204_2_WTC , v1204_2_GTC);
```

```
-- Route 1204(2) (Ends 8257)
-- Locks Points Normal
-- 1103/4
-- Locks Points Reverse
-- 1101/2
-- Tracks Clear Green : 1004 , 1009 , 1010 ,1011
-- Tracks Clear White

v1204_2_WTC = false;
v1204_2_GTC = v1004_TO and v1009_TO and v1010_TO and v1011_TO;
v1204_2_RS , v1204_2_DP, v1204_2_DG , v1204_2_DW , v1204_2_DR =
      Route( v1204_2_RC, (false -> pre v1204_2_S) ,
     (false -> pre v1101_RL and pre v1102_RL and pre v1103_NL and pre v1104_NL),
      v1204_LS, v1204_2_WTC, v1204_2_GTC);

-- Route 1206 (Ends A1203)
-- Locks Points Normal
-- 1103/4
-- 1101/2
-- Tracks Clear Green : 1010 , 1009 , 1008 , 1007
-- Tracks Clear White :

v1206_WTC = false;
v1206_GTC = v1010_TO and v1009_TO and v1008_TO and v1007_TO;
v1206_RS , v1206_DP, v1206_DG, v1206_DW, v1206_DR =
             Route(v1204_2_RC, (false -> pre v1204_2_S) ,
    (false -> pre v1101_NL and pre v1102_NL and pre v1103_NL and pre v1104_NL),
 v1206_LS, v1206_WTC, v1206_GTC);


-- Route 1205 (Ends A1203)
-- Locks Points Normal
-- 1101/2
-- Locks Points Reverse
-- 1103/4
-- Tracks Clear Green : 1005 , 1010 , 1009 , 1008 , 1007
-- Tracks Clear White

v1205_WTC = false;
v1205_GTC = v1005_TO and v1010_TO and v1009_TO and v1008_TO and v1007_TO;
v1205_RS , v1205_DP, v1205_DG , v1205_DW , v1205_DR =
   Route(v1205_RC, (false -> pre v1205_S) ,
      (false -> pre v1101_NL and pre v1102_NL and pre v1103_RL and pre v1104_RL),
      v1205_LS, v1205_WTC, v1205_GTC);
```

```
-- Routes Without Points
-- Route 1203
-- Tracks Clear Green : 1007, 8002
-- Tracks Clear White : 8004

v1203_WTC = false;
v1203_GTC = v1007_TO;
v1203_RS , v1203_DP, v1203_DG, v1203_DW, v1203_DR
       = Route(v1203_RC,
      (false -> pre v1203_S), true, v1203_LS, v1203_WTC , v1203_GTC );

-- Route 1201
-- Tracks Clear Green : 1002
-- Tracks Clear White : 1003

v1201_GTC = v1002_CR;
v1201_WTC = v1003_CR;
v1201_RS , v1201_DP, v1201_DG, v1201_DW, v1201_DR =
       Route(v1201_RC,(false -> pre v1201_S), true, v1201_LS, v1201_WTC , v1201_GTC);

-- Signals
-- A1201  (3 Aspect)

v1201_LS , v1201_RED, v1201_WHI, v1201_GRE =
       Light3Aspect(v1201_DR, v1201_DW, v1201_DG);

-- A1202 (3 Aspect)
-- There is no control table entry for this signal
-- it may not actually be part of the scheme.
--v1202_LS , v1202_RED, v1202_WHI, v1202_GRE =
--       Light3Aspect(v1202_DR , v1202_DW, v1202_DG);

-- A1203 (3 Aspect)

v1203_LS , v1203_RED, v1203_WHI, v1203_GRE =
       Light3Aspect(v1203_DR , v1203_DW, v1203_DG);

-- 1204 (2 Aspect with Route Indicator and RS)

v1204_LS , v1204_RED, v1204_GRE = Light2Aspect(v1204_2_DR or
       v1204_1_DR , ((v1204_1_DG and v1204_1_DW) or (v1204_2_DG and v1204_2_DW) ));

-- 1205 (2 Aspect with RS)

v1205_LS , v1205_RED , v1205_GRE =
      Light2Aspect(v1205_DR, (v1205_DG and v1205_DW ));
```

```
-- 1206 (2 Aspect with RS)

v1206_LS , v1206_RED , v1206_GRE =
      Light2Aspect(v1206_DR, (v1206_DG and v1206_DW ));

-- Fixed Reds  1207 , 1208 , 1209 , 1210

v1207_D = FixedRed();
v1208_D = FixedRed();
v1209_D = FixedRed();
v1210_D = FixedRed();

v1204_1_A, v1204_2_A, v1206_A, v1205_A, v1203_A, v1201_A,
      v1204_1_RC, v1204_2_RC, v1206_RC, v1205_RC, v1203_RC,
      v1201_RC, v1204_1_S, v1204_2_S, v1206_S, v1205_S, v1203_S, v1201_S
= RouteController(v1204_1_R, v1204_2_R, v1206_R , v1205_R, v1203_R,
         v1201_R , v1204_1_RS,  v1204_2_RS, v1206_RS , v1205_RS, v1203_RS , v1201_RS);


-- Safety Conditions
-- Points Locked when Route Set
sPLRS_1206 = (not v1206_RS or (v1101_NL and v1102_NL and v1103_NL and v1104_NL));
sPLRS_1204_1 = (not v1204_1_RS or (v1101_NL and v1102_NL and v1103_NL and v1104_NL));
sPLRS_1204_2 = (not v1204_2_RS or (v1101_RL and v1102_RL and v1103_NL and v1104_NL));
sPLRS_1205 = ( not v1205_RS or (v1101_NL and v1102_NL and v1103_RL and v1104_RL));

-- No proceed aspect if route not set
sNPA_1206 = (not v1206_GRE or v1206_RS);
sNPA_1204 = (not v1204_GRE or #(v1204_1_RS , v1204_2_RS));
    sNPA_1205 = (not v1205_GRE or v1205_RS);


-- No proceed aspect if train in route

sNPATR_1206 = (not v1206_GRE or (v1010_TO and v1009_TO and v1008_TO and v1007_TO));
sNPATR_1204_1 = (not (v1204_GRE and v1204_1_RS) or
                       (v1004_TO and v1005_TO and v1006_TO));
sNPATR_1204_2 = (not (v1204_GRE and v1204_2_RS) or
                      (v1004_TO and v1009_TO and v1010_TO and v1011_TO));
sNPATR_1205 = (not v1205_GRE or
                    (v1005_TO and v1010_TO and v1009_TO and v1008_TO and v1007_TO));

-- Crashes can not occur

nocrash = not v1004_CR;
```

```
-- Trains coming into a piece of track should occupy it

sTI1001   = true -> (not pre TrainIn) or v1001_TO;


tel
```

B.5 Modular Verification

B.5.0.1 Topological Verifcation

```
node ModularVerification1(TrainIn : bool)
returns (sTrainIn, sTrainOcc, vTRACK1_TO , vTRACK1_O, vTRACK1_CR ,
vTRACK2_TO , vTRACK2_O, vTRACK_CR  : bool)

let

vTRACK1_TO , vTRACK1_O, vTRACK1_CR = Track_Segment1( TrainIn, false, false, true );
vTRACK2_TO , vTRACK2_O, vTRACK_CR = Track_Segment1(vTRACK1_O , false, false , true);


-- Below are the safety conditions to verify that
-- two segments of track behave properly when connected together

-- Trains Coming into a piece of track should occupy it

sTrainIn   = true -> (not pre TrainIn) or vTRACK1_TO;

-- If a train leaves a piece of track  it should cease
             -- to occupy this piece of track and occupy the next

sTrainOcc = true -> (not pre vTRACK1_O) or
            (vTRACK2_TO and (not pre TrainIn or vTRACK1_TO));


tel


node ModularVerification2(TrainIn, Testpoint: bool)
returns (vTRACK_TO, vTRACK_O, vTRACK_O_2 , vTRACK_O_3, vTRACK_CR, sTP : bool)


let

vTRACK_TO, vTRACK_O, vTRACK_O_2 , vTRACK_O_3, vTRACK_CR =
```

```
       Track_Segment3(TrainIn , false,
       false ,  false , true, false , Testpoint , not Testpoint);


-- Points must influence which direction the train goes on the track.

sTP =  not vTRACK_O_2 or  not Testpoint and not vTRACK_O_3 or Testpoint;

tel


node ModularVerification3(TrainIn, RED , GREEN, WHITE : bool)
returns ( vTRACK1_TO , vTRACK1_O, vTRACK1_CR ,sRed : bool)

let

vTRACK1_TO , vTRACK1_O, vTRACK1_CR = Track_Segment1( TrainIn, RED, GREEN, WHITE );

-- Trains dont leave the track if a Red light is showing

sRed = not RED or vTRACK1_TO;

tel
```

B.5.1 Route Verification

– Route Verification

```
node ModularVerification4(RouteSet, PointsLocked, LightsSet,
                          W_track_clear, G_track_clear : bool)
returns(RouteSelected , DrivePoints, DriveGreen ,
         DriveWhite , DriveRed, SafetyRed : bool)
let

-- Route(RouteCall, RouteSet, PointsLocked, LightsSet, W_track_clear, G_track_clear)

 RouteSelected , DrivePoints, DriveGreen , DriveWhite , DriveRed =
      Route(false, RouteSet, PointsLocked, LightsSet, W_track_clear, G_track_clear);

-- If the route is never called The red light and only the red light should be driven.

SafetyRed = DriveRed and not DriveGreen and not DriveWhite;


tel
```

```
node ModularVerification5(RouteCall, RouteSet , PointsLocked , LightsSet,
                          W_track_clear , G_track_clear : bool)
returns( RouteSelected , DrivePoints, DriveGreen ,
         DriveWhite , DriveRed, SafetyRed : bool)
let



-- If the Route is called by but the track is not clear
     -- then the red aspect should be driven.

      -- There are two possibilities for the formalisation of this safety condition.
      -- RouteSelected , DrivePoints, DriveGreen , DriveWhite , DriveRed =
      --   Route(RouteCall, RouteSet , PointsLocked , LightsSet , false , false );
      -- SafetyRed = not RouteCall or (DriveRed and not DriveGreen and not DriveWhite);

RouteSelected , DrivePoints, DriveGreen , DriveWhite , DriveRed =
        Route(RouteCall, RouteSet , PointsLocked ,
              LightsSet , W_track_clear , G_track_clear );

      SafetyRed =  not( RouteCall and not W_track_clear and not G_track_clear)or
                   (DriveRed and not DriveGreen and not DriveWhite);

tel

node ModularVerification6(RouteCall, RouteSet , PointsLocked , LightsSet : bool)
returns(RouteSelected , DrivePoints, DriveGreen , DriveWhite ,
                                  DriveRed, SafetyCond : bool)

let


RouteSelected , DrivePoints, DriveGreen , DriveWhite , DriveRed =
       Route(RouteCall, true , PointsLocked , LightsSet , true , true );

-- If the track is clear and the Route is set then the Green light should show.


SafetyCond =  true -> (not pre RouteCall) or
               DriveGreen and not DriveWhite and not DriveRed;

tel


node ModularVerification7(RouteCall, RouteSet , PointsLocked , LightsSet : bool)
returns(RouteSelected , DrivePoints, DriveGreen , DriveWhite ,
                                  DriveRed, SafetyCond : bool)
```

```
var

foo : bool;
let

foo = false -> if (pre foo or pre RouteCall) then true else false;


RouteSelected , DrivePoints, DriveGreen , DriveWhite , DriveRed =
       Route(RouteCall, foo , PointsLocked , LightsSet , true , true );

SafetyCond =  true -> (not pre RouteCall) or
       DriveGreen and not DriveWhite and not DriveRed;

tel
```