SAT-based Model Checking and its applications to Train Control Systems

Phillip James

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

February 2010

Declaration

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

Signed	....	(candidate)
Date	....	

Statement 1

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

Signed	....	(candidate)
Date	....	

Statement 2

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

Signed	....	(candidate)
Date	....	

Abstract

Formal verification of railway control software has been identified to be one of the "grand challenges" [Jac04] of Computer Science. In this thesis, we demonstrate the successful application of various SAT-based model checking techniques to verify train control systems.

Starting with a propositional model for a control system, more specifically an interlocking, we show how execution of the system can be modelled via a finite automaton. We give both bounded and unbounded algorithms to perform SAT-based model checking over such an automaton, commenting on the advantages and disadvantages of each. In order to tackle the state space explosion problem, we propose slicing. We then give the correctness of this method with respect to our modelling approach.

The result of the thesis is a verification tool that combines the algorithms considered within the thesis. The tool has been applied to two real world interlocking systems and a discussion of the results is given.

Acknowledgements

Firstly, I wish to thank Dr. Markus Roggenbach for his guidance throughout the project. Without his patience and continuous support this project would not have been possible. I look forward to the prospect of continuing to work under his supervision. I also thank his close friend Erwin R. Catesbeiana (Jr) for keeping us both on track.

I would like to thank my family and close friends for their support throughout my university education. I especially thank Emma Thom for the many times where her support has been very much needed.

Thanks is also extended to my second supervisor Dr. Anton Setzer, and to my examiners Prof. Achim Jung and Dr. Neal Harman. Their comments and suggestions have helped substantially improve this project.

I would like to thank our industrial partners Invensys for funding the project, and thank Swansea University's Department of Computer Science for giving me the opportunity to complete this work.

Finally, I would like to acknowledge all the support from my close colleagues within the Computer Science, including Liam O'Reilly, Temeshgen Kahsai, Matthew Gwynne, Fredrik Nordvall Forsberg, Karim Kanso, Jennifer Pearson and Tom Owen.

Table of Contents

Chapter 1

Introduction

Contents

Railways are a prominent example of critical systems. Failures in the control systems for railways can have disastrous consequences. The growing pressures on railway systems to provide both safe and efficient operation of the railway network has lead to an increased interest into the applications of Computer Science. Since the early 1980's, following an act of law allowing the use of microprocessors in the design of safety critical systems [KR01], there has been a steadily increasing application of Computer Science within the railway industry. From advanced routing systems [ZKvH01, MO07] through to automatic train protection systems [Sim94] Computer Science now plays a role in many aspects of modern railway systems.

1.1 Overview of Formal Methods in the Railway Domain

In recent years there has been a large amount of interest [Bjo09, KMS08, WR03, Win02, HP00, INE09, BG00] in the application of formal methods, including formal verification of systems, within the railway domain. Here we shall concentrate particularly on the formal verification of railway computer systems with regards to safety properties.

Early work including [BFG$^+$98, BAB$^+$95] explored the possible application of traditional modelling and verification methods to railway systems. These approaches used traditional techniques such as process algebra and often the results illustrated the high complexity involved in verification of complex systems, such as railway systems. This is exactly why the verification problem for railway systems has been described as a "grand challenge" of Computer Science [Jac04]. Recent advances in verification methods, including the introduction of satisfiability (SAT) solvers [BHvMW09] to model checking [SSS00, ES03, ADK$^+$05], have meant that more recent attempts at verifying railway systems have in fact been successful [KMS08, Kan08, WR03]. This project, in co-operation with Invensys [inv09], an internationally established company[1], explores the application of such modern verification processes to real world railway systems, leading to the production of a verification tool for interlockings [KR01] programmed using ladder logic [IEC03].

1.2 Project Aims and Approach

In this project, due to recent successes [Kan08, KMS08], we explore SAT [BHvMW09] based verification methods and how such methods scale up with application to real world railway control software. We explore a series of verification techniques including inductive verification [SSS00], bounded model checking [BK08, CGP99] and temporal induction [SSS00, ES03], applying each method to yield the verification of Westrace interlockings [wes09]. Along with these methods we propose techniques to reduce the complexity of verification, including program slicing [Tip95, FH98] and comment on the possibilities of applying functional dependency removal [JB04]. Finally these techniques have been combined into a verification tool for use by Invensys.

The project continues and extends a successful project by Kanso [Kan08], where SAT based inductive verification gave impressive results when applied to the verification of a single Westrace interlocking system [wes09]. Kanso develops a parser for ladder logic programs, the language used to describe such interlocking systems. Using the parser, Kanso translates such ladder logic programs into a propositional formulae based model. Finally Kanso translates given safety conditions into propositional formulae and verifies that these safety conditions are valid within the propositional ladder logic model. The approach taken by Kanso motivates our work and for this reason we shall revisit the approach in more depth in Chapter 4.

The approach by Kanso, was not without its problems. The main problem highlighted in the work by Kanso, was the problem of unreachable system states. This problem gives arise to the need for manual analysis of the counter examples to decide whether or not the given counter example is reachable by the system. For inclusion into the standard development process of interlockings, Invensys requires further automation of the verification. For these reasons, the main aim of this project is to explore the feasibility of SAT based model checking techniques. Such techniques are able to exclude the problem of unreachable states and in the case that a safety condition does not hold, not only do they allow the production of counter examples, but also traces to how these counter examples occur.

To achieve this goal, we use the modelling language of propositional logic. We model both the interlocking system requiring verification, and the safety condition, before formulating the problem into a satisfiability problem [BHvMW09]. This satisfiability problem is then discharged to a SAT solver [BHvMW09, min09, par09]. The result of the verification process is either that the system is safe, or that a counter example trace is produced to a violating system state. A further result of this project, is the verification of a new

[1]e.g., within Australia, Germany and the U.K.

train station. That is the verification tool we give has been applied to verify not only the interlocking verified by Kanso in [Kan08], but also to the verification of another slightly more complex interlocking system. First reports on this research have been presented at BCTCS'09 [Jam09a], Swansea Science Day [JK09] and at the postgraduate workshop VINO [Jam09b]. Furthermore, preliminary results of the work have been published in PPL'09 [JIR09] and CALCO'jnr [JR09].

Previously, various formal methods have been applied to the area of verification of railways, including algebraic specification, e.g. [Bjo09], process algebraic modelling and verification, e.g., [Win02, PGHD04], and also model oriented specification, where, for example the B method has been used in order to verify part of the Paris Metro railway [BG00] in terms of both safety and liveness properties. There are also several international research projects [INE09, Tra09] devoted to the challenge of formal verification within the railway domain.

Importantly for us, approaches have also been taken towards the verification of ladder logic programs [ZRK03, FH98]. Ladder logic is the language used by Invensys to program their Westrace interlockings. Firstly in [ZRK03] a ladder logic program is translated into a timed automaton and automatically verified against given properties using the model checker Uppaal [upp09]. Here the problem of state space explosion is highlighted as the main restriction of the approach when considering larger ladder logic programs. In [FH98] an inductive verification approach is taken to verify ladder logic interlockings, the issue of very large formulae arising from the verification approach is highlighted, and finally some possible solutions are proposed.

1.3 Thesis Outline

In Chapter 2, we introduce the reader to the terminology and history of railway systems. We give an overview of interlocking systems and introduce the operational structure of the Westrace interlocking. Finally we gave a detailed introduction to ladder logic programs.

Chapter 3 introduces the modelling approach we use to model ladder logic diagrams. On the syntactical level we use the language of propositional logic to describe ladder logic programs, and on the semantic level we use finite automaton to allow us to reason about such ladder logic programs. Chapter 3 concludes with an overview on the modelling of safety conditions, and an introduction to the satisfiability problem for propositional logic.

In Chapter 4, we discuss the field of verification. We define exactly what it means to verify that a ladder logic program respects a given safety condition. Continuing from this, we pay attention to the work by Kanso [Kan08, KMS08], from which this project takes its base. This includes a discussion of inductive verification using a SAT solver, along with both the successes and the failures of such an approach to verification. Finally an example application of inductive verification is given, based on a small example ladder logic program.

In Chapter 5, the verification approach of Model Checking is reviewed. The algorithms which are explored in this project are introduced, along with an discussion of how they overcome the problems obtained from using inductive verification. That is, the problem of unreachable states and counter example trace generation. Once again, the chapter closes with an example application of the proposed algorithms to a small ladder logic program.

Chapter 6 extends the algorithms and approaches given in Chapter 5 to be unbounded. A discussion is held into how inclusion checks can be formulated in propositional logic, to ensure that the reachable state space of a ladder logic program is fully verified. This involves a discussion of loop freedom within automaton, and how this plays a role determining inclusion. A inclusion check is then given, along with a correctness proof for the inclusion check. Finally the verification approach of temporal induction is introduced, and the application of both the explicit inclusion check and temporal induction is illustrated on an example ladder logic program.

Chapter 7 begins by introducing the research area of program slicing. It is shown how program slicing is used within the debugging of systems, and then an algorithm for slicing ladder logic programs is given. This algorithm is used to reduce the propositional formula size needed to represent such a ladder logic program. The application of slicing is shown via means of an example, and a correctness proof is given showing that the application of slicing preserves validity of the ladder logic program, with respect to a given safety condition.

In Chapter 8, we give an overview of the verification tool that has been created for use by Invensys. We show how the tool is an extended and improved version of the tool given in [Kan08]. We then comment on the problems that had to be overcome to enhance the tool to allow us to verify a new train station. We give a discussion of the software engineering principles that have been followed in the development of the tool. Finally some possible future improvements that could be made to the tool are given.

Chapter 9 gives the verification results achieved using our tool. The results of verification of two train stations are given, along with a comparison of all proposed verification techniques. In this chapter, we see the successes of the project in terms of meeting its goals to give a verification tool that is able to produce counter example traces and also ignore unreachable system states. Finally the results of applying our slicing technique to the verification process is discuss.

In Chapter 10 we draw the thesis to a close. To do this we comment on the main practical and theoretical results of the project, and propose some possible areas of future work.

Chapter 2

Railway Signalling

Contents

The birth of railways as we know them, dates back to 1804, when Richard Trevithick introduced a steam locomotive guided by rails [Bur02]. The locomotive was used to transport iron in the South Wales valley of Merthyr Tydfil, and was the first such installation anywhere in the world. Since this introduction, railway design has seen many advances, leading to modern high speed railways seen across the world today. This project focuses on computerised equipment currently used within railways. For this reason, we introduce the reader to some basic railway components and terminology visible in nearly all modern railways. Continuing from this, we give a brief history of railway signalling systems, motivating the use of formal methods within the design of such systems. Finally we look at the details of interlocking systems, concentrating on the Westrace Interlocking system [wes09] designed by Invensys.

2.1 Railway Components

Here, we introduce the main railway terminology we will use throughout the following chapters. To do this we give a small example of a typical section of railway (Figure 2.1) and discuss the main features that occur within it. More details on all the topics discussed below can be found in [KR01].

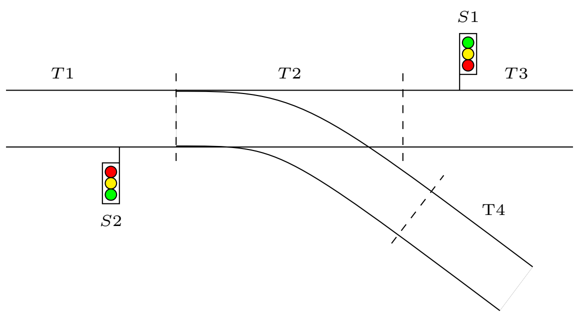


Figure 2.1: An example railway.

2.1.1 Track Segments

A track segment is usually an alpha-numeric string that is used within track plans to break the overall railway down into small areas of track. Figure 2.1, shows four track segments, namely $T1$, $T2$, $T3$ and $T4$. Track segments are not fixed to a particular size, and often some track segments represent very long sections of the railway and others very short.

2.1.2 Signals

Signals are primarily used to inform train drivers of possible problems on the railway ahead, and what actions they should take to prevent an accident. Usually signals are placed along the side of the track, as shown in Figure 2.1. Within the railway domain, the indication that is given to a driver by a signal is known as an "aspect". Originally different aspects were displayed using different physical orientations of the signal, although most modern signals make use of coloured lights. The signals $S1$ and $S2$ in Figure 2.1 both have three aspects, namely:

- Red – Indicating to the driver that he should stop.
- Yellow[1] – Indicating to the driver that he should proceed with caution.
- Green – Indicating to the driver that he may proceed.

These three colours are the main colours used in varying types of signal across the world, although some signals also make use of white aspects. The signals in Figure 2.1 are known as three aspect signals and along with these both two aspect and four aspect signals are commonly used. Usually, four aspect signals are used on higher speed lines and two aspect signals on lower speed lines. The different number of aspects allow for a different variety of messages to be portrayed to the driver, which in turn gives the driver a safe breaking distance. The details of such signalling schemes will not be of interest for the remainder of the document, so for further information the reader is encouraged to see [KR01].

[1]Notice this is not amber like in other transportation systems.

2.1.3 Points

The term "point" is used within railways to represent a junction or split in the rails. In track segment $T2$ of Figure 2.1, there is a point that splits the track into two. A point is traditionally controlled via a lever, allowing the operator to decide which path the oncoming train takes. In our example in Figure 2.1, if the point is set to allow travel from track segment $T1$ through to track segment $T3$, then it is said to be in "normal" position. Whereas if it was set to allow travel between $T1$ and $T4$, then it is said to be in "reverse" position.

2.1.4 Routes

The last topic of railways we study is that of routes through the railway. By route, we mean the physical path a train may take along the tracks of a railway. For example, in Figure 2.1, assuming trains can only travel from left to right, there are two possible routes that a train may take, the choice of which is dependant on the positioning of the point. The operation of deciding and setting which route a train should take is known as route setting. Route setting can be performed by a human signaller or automatically by given railway systems. The operation of setting a route involves changing various signals along the track, and also setting the position of many points. Clearly then, route setting needs to be monitored carefully to avoid trains being derailed or crashing.

2.2 History of Signalling

Since railways first began in the early 19^{th} century, there has always been some form of signalling technique to control the operation of trains. Even though these techniques did not initially include fixed signals, for example policemen giving signals to train drivers using coloured flags, the notion of signalling has played a key role in shaping the current design of railways around the world.

The principle task of signalling within the railway is to ensure the safe operation of the railway, for example a signalling system should not allow two trains to collide, nor should it allow one train to follow another at a dangerously close distance. Another key role that signalling must fulfil, is with regards to the throughput of the railway. As the demand for quicker train services grows, so does the need for more complex signalling algorithms. From this point onwards we shall only consider aspects of signalling regarding safety, although the reader should be aware that much of the complexity of modern interlocking systems is caused by the need for ever more efficient signalling systems. We shall now give a brief historical account of railway signalling and its main concepts.

The earliest form of a signalling system within railways consisted of policemen patrolling each individual station. As mentioned above, the policemen would indicate to passing drivers whether or not it was safe to proceed using different coloured flags (or oil lamps by night). Policemen would use their own timing method, typically a pocket watch, to

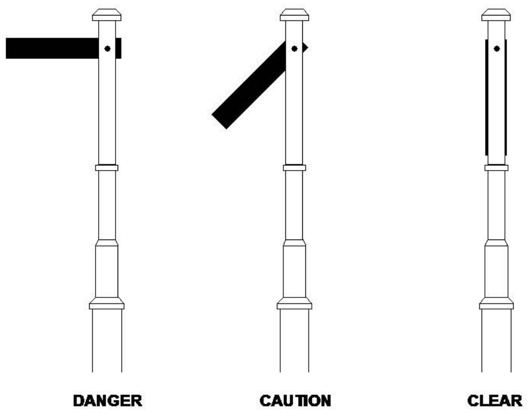


Figure 2.2: Illustration of the aspects of Semaphore signals.

decide when it was safe for a train to depart from the station. This scheme worked relatively successfully given that there was no communication between adjacent stations, and that timings between stations were not synchronised in any way. Many people often argue [KR01] that the lack of accidents during the early years of railways, was probably due to the low speed of trains at the time.

Around 1842, flags were beginning to be replaced by fixed signals that indicated whether or not a train should proceed. These signals came in many forms, the most popular of which was the *'semaphore'* signal. The semaphore signal has three different aspects:[2] one aspect for proceed, another aspect for proceed with caution and a final aspect for stop. These aspects are illustrated in Figure 2.2. Quickly after the introduction of semaphore signals, it was realised that many signals could be operated from a single location. Through the use of wires, each signal was connected to a lever, meaning that a policeman no longer had to manually walk around changing signals but could just control all the levers from a central point. This central location became known as the *'signal box'*, and the controllers of the signal box became known as signallers.

The next major design step came in the late 1850's, The idea to design the levers within signal boxes in such a way that they were physically locked in position unless it was safe for the signaller to move the lever was introduced. This feature of signalling is known as an *'interlocking'* and is one of the main components used within signalling today. Interlocking systems will be discussed further in Section 2.3.

Around the same period as interlockings were being introduced, simple Morse code like communication techniques were installed between adjacent stations. With this communication and interlockings becoming well established, railways in general managed to achieve rather high safety levels even though the speed of trains was increasing.

Continuing the trend to obtain better safety levels, the next major development was the introduction of the so called *'track circuit'* in the early 1870's. A track circuit is an electronic circuit that could detect whether or not a train was occupying the current track segment.

[2]Aspect refers to the physical state being shown by a signal.

Using such track circuits along a length of railway allowed automatic operation of signals via these electrical circuits. Very quickly electrical circuits became widely used within the railway. This included the introduction of electronic light based signals. In the early 1920's, coloured lights were introduced for signalling. Like the traditional traffic lights seen on many road networks, different colours of light were used to indicate to the driver of a train information about what was ahead of them on the track. Over the next ten years, electrical circuits became even more widely used including to assist in the manual manoeuvring of levers. Eventually, levers to control signals were removed from the railway and signallers would change a signal or point by simply pressing a button on a central control panel.

A trend of small adjustments to railway systems took place over the next twenty five years. Then in the mid 1980's, a very important law was changed to allow the use of electronic components in the control of safety critical devices [KR01]. This meant that well established mechanical interlockings could be replaced by microprocessor based interlockings, known as solid state interlockings or SSI's [Lea91]. This change brought with it a vast range of new features for the signaller. One of these features included automatic route setting by interlockings, where the interlocking would automatically set signals and change track positions depending on information provided to it from a railway timetable.

Currently new trends in railway systems often rely on the train itself sending information about its location and velocity to the interlocking and other safety systems. This enables trains to communicate with the signalling systems directly, meaning technology such as automatic warning systems (AWS) [Noc85] and automatic braking systems (ATP) [KR01] can effectively aid train drivers in the safe running of the railway. With this kind of direct communication between a train and the signalling system, the need for physical signals warning drivers is quickly becoming redundant. There are even modern railways where the need for train drivers has already become redundant, for example the Copenhagen Metro is completely automated, and this trend seems set to continue in the future.

2.3 Interlocking Systems

The main objective of an interlocking system is to ensure safety within the railway. This entails ensuring that certain rules are followed whilst computing if the future state of the railway is safe. For example, an interlocking should not allow a signals aspect to be changed to "Proceed" if not all of the specified safety conditions are met. The interlocking is also responsible for bringing the railway into a fail safe state if a problem is detected during operation. This, in most railways, means bringing all trains to a stop by setting all signals to show the red aspect.

The interlocking itself enforces a set of rules which cannot be broken when performing a task to change the state of the railway. Figure 2.3 illustrates the typical use of an interlocking as a safety layer. The interlocking interfaces with both the physical track layout and the human (or computerised) controller. The controller will issue a request, such as to change a signal's aspect to green, then the interlocking will use the set of rules and current track information to determine whether it is safe for the operation to be permitted. If it is safe

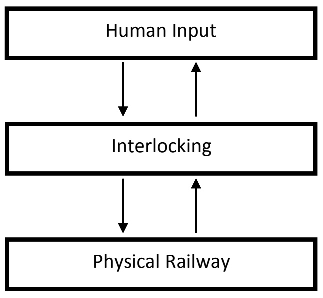


Figure 2.3: An interlocking is a safety layer.

then the interlocking will change the physical layout of the track, informing the controller of the change. Whereas if it is unsafe to complete the operation the interlocking will not allow the physical track layout to be changed, and will report back to the controller that the operation will yield an unsafe situation.

The earliest interlockings consisted of many mechanical levers, connected to signals, track points and other railway entities. The levers were physically arranged so that a lever could only be moved into a new position if it was safe to move the corresponding track entity. Over the last thirty years or so, more and more computer based interlockings, such as the Westrace interlocking, have appeared on mainstream railways. Modern interlockings are now vastly more complex than the original mechanical interlockings. Interlockings now obtain information not only from what signallers are entering into the system, but also from electronic relays on track circuits and information from trains themselves about their position and velocity etc.

Mainly for historical reasons, the underlying safety properties required to be upheld by the railway are recorded into large tables known as control tables [Lea91]. The control tables are designed by experienced engineers who manually design the safety properties to be ensured by the interlocking. During the design process of such control tables, a continual test pattern is followed to ensure the designs are correct. Once this process has been completed to a level that is high enough to be deemed safe, the control tables are encoded into a Westrace interlocking[3] using the ladder logic language [IEC03]. Given that this process is completed by humans, there is always room for human error. This is exactly why we shall explore the formal verification of Westrace interlockings.

2.4 The Implementation of Westrace Interlockings

We shall now look more closely into the implementation of a Westrace interlocking. Firstly we discuss the overall control structure of the Westrace, before briefly discussing the physical parts of the interlocking. Finally we give a detailed definition of the control program run by a Westrace interlocking, showing a small example of such a program.

[3]The reader should note that different interlockings such as the VPI [vpi09] use different implementation languages.

2.4.1 Westrace Control Structure

As discussed above, the Westrace interlocking communicates with both the system signaller and the physical track layout. Here we look at how and when these communication processes take place, noting how certain information is manipulated and returned by the interlocking.

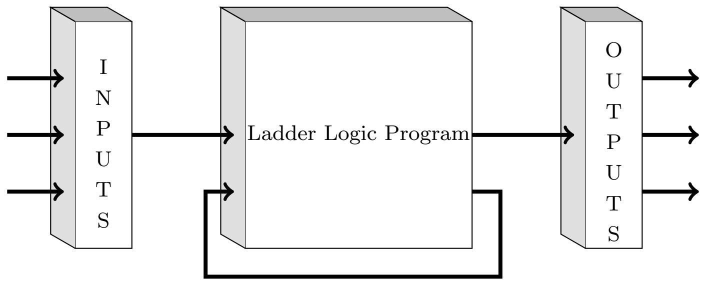


Figure 2.4: Control cycle of a Westrace interlocking.

Figure 2.4 shows the typical control process taken by a Westrace interlocking. It shows the following three main stages of operation.

1. **Reading of Inputs.** – The first stage involves reading input values from various sources. The input reading process is undertaken by a specialised "I/O module". Inputs may include requests from signallers and details from physical track sensors. It is also possible for input values to be defined as remembered values from the previous execution of the ladder logic program.

2. **Internal Processing.** – The second stage involves computing new values for output variables. This task uses the variables that have been read in stage one. These variables are then run through a ladder logic program. We will discuss this ladder logic program in more depth in Section 2.5.

3. **Committing of Outputs.** – Finally, all calculated outputs are passed back to the "I/O Module" to be committed to various sources. Here we note that some outputs may actually be remembered by the interlocking, ready to be used within the next execution of the control cycle. In this stage, commands to change the physical track layout may be issued and information may be passed back to the signaller.

These three stages of execution are executed repeatedly[4] until a problem is detected. If no problem is detected then the operations are repeated indefinitely. This control loop is similar to the common control loop executed by many safety critical applications and often found within control theory [Lev96]. As a notational remark, from this point onwards, we shall use the words "iteration" and "execution" interchangeably, to represent a single execution of steps one through to three of the Westrace control cycle.

[4]For example they could be placed within a simple "while-loop" construct.

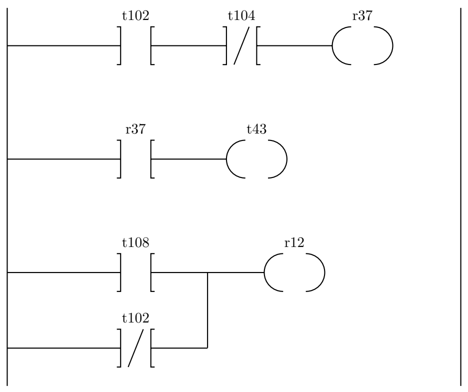


Figure 2.5: An example section of a ladder logic program.

2.5 Westrace Ladder Logic Programs

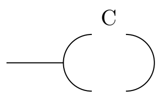
The second step of the control cycle for a Westrace interlocking involves the execution of a ladder logic program [IEC03]. Here we shall define precisely what a ladder logic program is. To do this we firstly show an example of a ladder logic program, introducing the main concepts of ladder logic. Then in the next chapter we shall introduce a theoretical modelling of ladder logic.

Ladder logic is graphical language described in IEC standard 61131 [IEC03] and is often used for the programming of Programmable Logic Controllers [Bol06] (or PLC's). It gets its name from its "ladder" like graphical appearance. The actual ladder logic programs used by Invensys in the Westrace interlocking only uses a subset of the features of full ladder logic, and from this point onwards the reader should note that whenever ladder logic is used we are in fact referring to this subset. Figure 2.5 shows part of a simple ladder logic program.

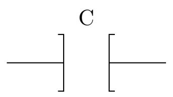
The ladder logic diagrams themselves contain a series of horizontal bars, these bars are known as "rungs". Each rung of a ladder logic program contains a series of constructs that contain variables which can have values 1 or 0[5]. The constructs that can be included on a rung are:

- **Coils** – These are used to represent values that are either output by the program, or used internally later in the program. Coils always occur as the right most construct of a rung. The value of a coil is determined by the variables occurring previously in the rung and the shape of the rung.
- **Open Contacts** – These are used to simply represent the value of the variable within them.
- **Closed Contacts** – These are used to represent a negation of the value of the variable within them.

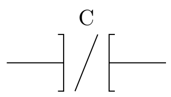
[5]or closed and open as they are commonly referred to in engineering.



(a) A coil.



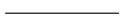
(b) An open contact.



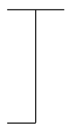
(c) A closed contact.

Figure 2.6: Main constructs of Ladder Logic.

Along with these constructs, the shape of each rung also carries meaning, it is used to determine the value of the corresponding coil. Using a propositional logic setting [HR04], a horizontal line connecting two contacts has the logical meaning of an "And" operation, and a vertical line connecting two constructs has the logical meaning of an "Or" operation. A value is now calculated for each coil by starting at the left hand side of each rung, and calculating, using these constructs and logical operations, the resultant 0 or 1 value.



(a) Horizontal connection representing conjunction.



(b) Vertical connection representing disjunction.

Figure 2.7: Possible rung connectors of Ladder Logic.

In Section 3 we show how the semantics of ladder logic programs can be captured completely using propositional formulae.

2.5.1 A Pelican Crossing Example

We now introduce an example ladder logic program which we use as a running example from this point onwards. An illustration of a simple pelican crossing system is given in Figure 2.8. Note that the choice of using a pelican crossing as an example has been motivated by work in [Kan08].

A pelican crossing is a computerised system found on many road networks throughout the world. The basic idea is that a pelican crossing allows pedestrians to safely cross a flow of traffic. To this end, a pelican crossing consists of the following components: four traffic lights - two for pedestrians, two for the traffic, where for simplicity we assume that all these traffic lights can only show red or green. The pedestrians traffic lights emit an audio signal when they show green and have an input button which a pedestrian can press in order to request the green signal.

In order to program our system, we use the following Boolean variables, distinguished into input, output, and state variables. There is only one input variable, namely *pressed.* This variable becomes true if a pedestrian presses the button at either pedestrian light. We use the suffix g to indicate that a traffic light shows green, and the suffix r to indicate that a traffic light shows red. There are four traffic lights, namely *pla* and *plb* for pedestrians, and *tla* and *tlb* for traffic. Thus, overall there are eight output variables for lights, namely *plag*, *plar*, *plbg*, *plbr*, *tlag*, *tlar*, *tlbg*, and *tlbr*. When one of these variable is true, the corresponding light is on. There is one output variable *audio*. When *audio* is true then the audio signal is sounding. Finally there are two state variables, *req* which "remembers" the value of *pressed*, and *crossing* which indicates that pedestrians may cross the road.

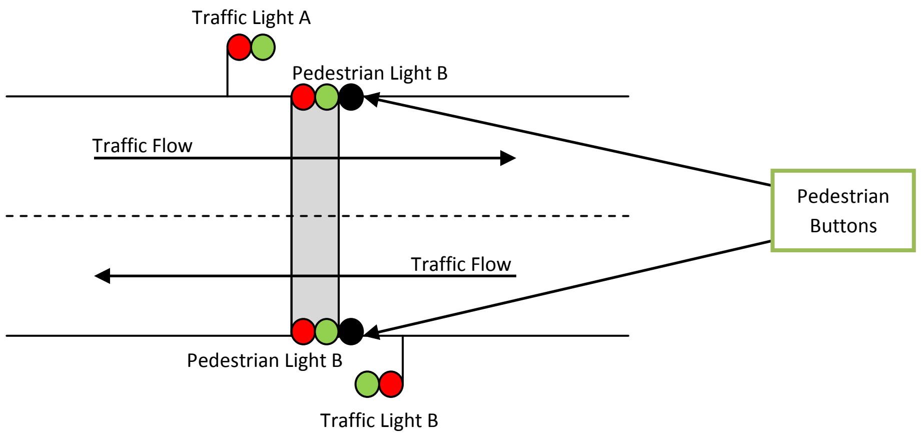


Figure 2.8: An example pelican crossing.

A ladder logic program to control such a pelican crossing system is given in Figure 2.9. This example illustrates further, the constructs of ladder logic. The ladder logic program in Figure 2.9 contains, in total, twelve rungs. We can see, that each rung is made up of a series of closed and open contacts, and that these contacts are connected using vertical and horizontal connections. Finally, the program also illustrates that all rungs must end with a coil.

The first line of Figure 2.9, can be read as: if there was a request *req* and in the last control cycle and pedestrians were not allowed to cross the road, then at the end of the current cycle pedestrians will be allowed to cross the road. Its second line says: In the next cycle *req* will be true if a pedestrian pressed the button before starting this cycle (indicated by *pressed*) and in the previous cycle there was no request. The remainder of the program can be read similarly.

Throughout the remainder of this document we shall see how we can model and verify this ladder logic program using various techniques.

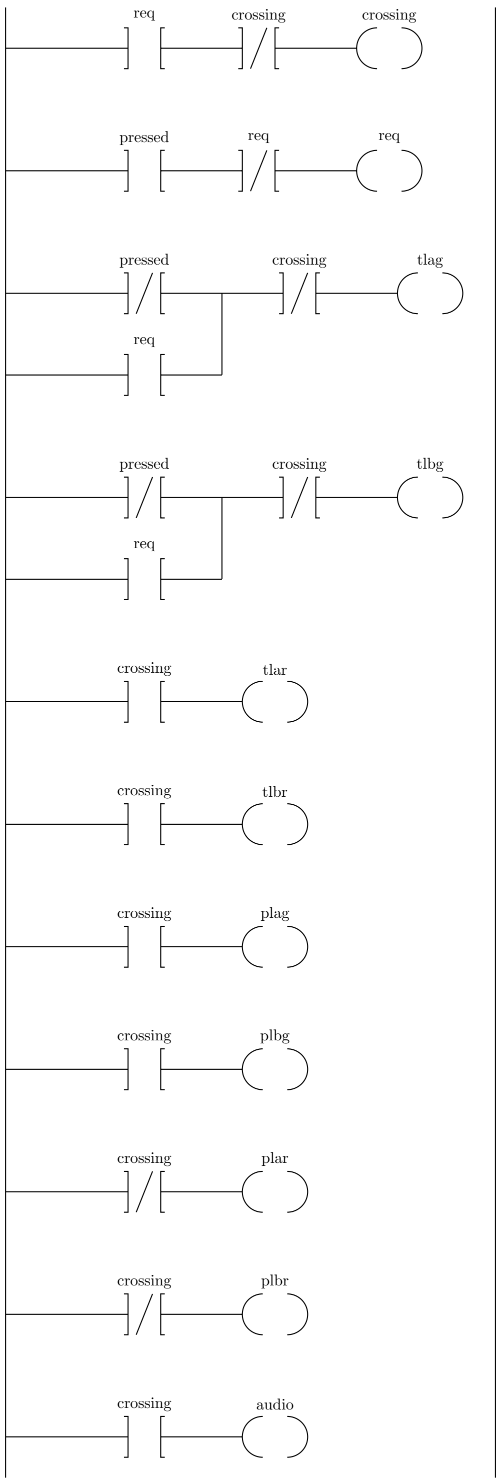


Figure 2.9: An example ladder logic program to control a pelican crossing.

Chapter 3

Modelling Interlockings and SAT

Contents

In this chapter, we discuss how we model Westrace interlockings. This will involve a discussion of propositional logic, before showing how the various parts of ladder logic can be described using propositional logic. We then continue to explain how we can use such propositional formulae to gain an automaton theoretic modelling. We will discuss which properties we would like to show hold for such interlockings, commenting on how we can represent such properties using first order logic. Finally we will introduce some basic background information on SAT solvers, giving an overview of the main algorithms used within successful tools such as MiniSat [min09] and zChaff [cha09].

3.1 Propositional Logic

To allow us to model ladder logic programs, we need a mathematical language which captures all of the features of ladder logic and allows us to reason about such ladder logic programs. For this language, we have chosen propositional logic following successful approaches of [Kan08, KMS08, FH98]. We firstly introduce propositional logic before continuing to show how we can use propositional logic to model ladder logic programs.

We define the syntax of propositional formulae relative to a given set of variables.

Definition 3.1 (Propositional Formulae)**:** Given a set of variables V, we define the set of propositional formulae $PROP$ over $\wp(V)$ as

- $\top \in PROP$.

- $\bot \in PROP$.
- $v \in PROP$ for $v \in V$.
- $\neg\psi \in PROP$ for $\psi \in PROP$.
- $\psi \wedge \phi$ for $\psi, \phi \in PROP$.
- $\psi \vee \phi$ for $\psi, \phi \in PROP$.

For convenience, we shall adopt the following commonly used notations:

$$\psi \Rightarrow \phi \text{ to represent } \neg\psi \vee \phi \text{ and,}$$
$$\psi \Leftrightarrow \phi \text{ to represent } (\psi \Rightarrow \phi) \wedge (\phi \Rightarrow \psi).$$

Now we have a notion of propositional formulae, we define inductively a function $vars : PROP \to V$, which gives us the set variables that appear in a propositional formulae.

Definition 3.2 (Variables of a formula)**:** Given a propositional formula $\psi \in PROP$ we define $vars : PROP \to \wp(V)$ inductively as,

- $vars(\top) = vars(\bot) = \emptyset$.
- $vars(v) = \{v\}$ for $v \in V$.
- $vars(\neg\psi) = vars(\psi)$ for $\psi \in PROP$.
- $vars(\psi \wedge \phi) = vars(\psi) \cup vars(\phi)$ for $\psi, \phi \in PROP$.
- $vars(\psi \vee \phi) = vars(\psi) \cup vars(\phi)$ for $\psi, \phi \in PROP$.

For a propositional formulae, we shall use the standard notions of a valuation and satisfaction. That is:

Definition 3.3 (Valuation)**:** A valuation for a variable set V is a map $\mu : V \to \{0, 1\}$.

Then for satisfaction we have:

Definition 3.4 ($\models$)**:** Given a set of variables V, a valuation $\mu : V \to \{0, 1\}$ and a propositional formulae $\psi, \phi \in PROP$. We define

- $\mu \models \top$.
- $\mu \not\models \bot$.
- $\mu \models v$ iff $\mu(v) = 1$ for $v \in V$.
- $\mu \models \neg\psi$ iff $\mu \not\models \psi$.
- $\mu \models \psi \wedge \phi$ iff $\mu \models \psi$ and $\mu \models \phi$.
- $\mu \models \psi \vee \phi$ iff $\mu \models \psi$ or $\mu \models \phi$.

Later, in Section 3.5, satisfaction of propositional formulae will be the key topic. Satisfaction of such propositional formulae will be the basis of the verification techniques we use to verify ladder logic programs.

3.2 Modelling Ladder Logic in Propositional Logic

Work has already been completed to translate a given ladder logic program into a propositional formula representation [Kan08]. Here we define exactly the syntactical structure of the resultant propositional formula gained from this translation. First, we shall define some variable sets corresponding to the different types of variables that can occur in a ladder logic program.

The first set of variables which we will consider are the coils of the ladder logic program. These are the variables that can be assigned to within the program and in some sense, the variables that hold the values output by a ladder logic program.

Definition 3.5 (Coils of a ladder logic program)**:** We define C to be the set of all coils occurring in a ladder logic program P.

Considering our pelican crossing example ladder logic program in Figure 2.9, we can read off the following set of coils:

$$C = \{crossing, req, tlag, tlbg, tlar, tlbr, plag, plbg, plar, plbr, audio\}.$$

The next set of variables which are used within the ladder logic program are the set of variables read as inputs in step one of the control cycle.

Definition 3.6 (Input variables of a ladder logic program)**:** We define I to be the set of all variables given as inputs to a ladder logic program P.

Again considering our example ladder logic program, we can read off the following set of inputs:

$$I = \{pressed\}.$$

Notice that in contrast to coils, inputs to ladder logic programs have no explicit representation in the program. This is because the inputs are provided to the ladder logic program by the "I/O module" and hence are only referred to in the ladder logic program. Also note that once input have been set by the "I/O module" they are not updated for the remainder of that execution cycle.

Before defining the full set of variables that a ladder logic program contains, we firstly give a small notational definition that will make future definitions easier to interpret.

Definition 3.7 (*prime* and *unprime*)**:** Given a universe of variable names U we define two bijective functions, $prime$, $unprime : U \to U$ such that for all $x \in U$

$$unprime(prime(x)) = x \text{ and}$$

$$prime(unprime(x)) = x.$$

Remark 3.8: For some variable x, we often write x' to represent $prime(x)$, and for some set of variables $V = \{x_1, \ldots, x_n\}$ we write V' to represent $\{x'_1, \ldots, x'_n\}$.

Finally we define the full set of variables used by a ladder logic program,

Definition 3.9 (Variables of a ladder logic program)**:** The set of variables V of a ladder logic program P is defined as a finite set $V = I \cup C \cup C'$ where $I \cap (C \cup C') = \emptyset$.

Remark 3.10: From this point onwards, when we refer to the set of variables V, we always mean the set of variables for a ladder logic program via the given definition.

To illustrate such a variable set, consider once more our pelican crossing example in Figure 2.9, were we have the following variable set:

$$V = I \cup C \cup C'$$

where

- $I = \{pressed\}$,
- $C = \{crossing, req, tlag, tlbg, tlar, tlbr, plag, plbg, plar, plbr, audio\}$,
- $C' = \{crossing', req', tlag', tlbg', tlar', tlbr', plag', plbg', plar', plbr', audio'\}$.

Here, the set of variables C represents values remembered by the ladder logic program from its previous execution. This follows from the control cycle of a ladder logic program given in Figure 2.4.

Using the structure of a ladder logic program, we can define the format of a rung in propositional logic:

Definition 3.11 (Rung)**:** A rung (over V) is a propositional formula $R \in PROP$ with the structure $R \equiv c' \Leftrightarrow \psi$ where $c' \in C'$, $\psi \in PROP$ over V, and $c' \notin vars(\psi)$.

An example of such a rung is the first line of Figure 2.9 where we have the rung

$$crossing' \Leftrightarrow (req \wedge \neg crossing).$$

Using our definition of a rung, we can describe an entire ladder logic program as an order dependant series of rungs represented as propositional formula. The following definition captures the requirement that the order of rungs matters within ladder logic programs.

Definition 3.12 (Ladder Logic Propositional Formulae)**:** A ladder logic formula ψ_P (relative to a set of input variables I and a set of state variables C) is a propositional formula

$$\psi_P \equiv ((c'_1 \Leftrightarrow \psi_1) \wedge (c'_2 \Leftrightarrow \psi_2) \wedge \cdots \wedge (c'_n \Leftrightarrow \psi_n))$$

for some $n \geq 0$, such that:

- for all $1 \leq i \leq n : c'_i \in C'$. I.e., c_i is an output coil.
- for all $1 \leq i, j \leq n$: if $i \neq j \Rightarrow c'_i \neq c'_j$. I.e., output coils are uniquely defined.
- for all $1 \leq i \leq n$:, $vars(\psi_i) \subseteq I \cup \{c'_1, \dots c'_{i-1}\} \cup \{c_i, \dots, c_n\}$. I.e., Variables occurring on a rung must have been previously defined either as inputs or coils.

Note that here we use the convention that $\{c'_1, \dots, c'_0\} = \emptyset$. If $n = 0$, as usual $\psi \equiv True$. The empty program will prove to be useful later, in the context of slicing.

Remark 3.13: For a ladder logic propositional formula

$$\psi_P \equiv ((c'_1 \Leftrightarrow \psi_1) \wedge (c'_2 \Leftrightarrow \psi_2) \wedge \cdots \wedge (c'_n \Leftrightarrow \psi_n)) \in PROP_{LL}$$

we will use the notation

$$\psi_p \equiv [R_1, R_2, \dots, R_n] \in PROP_{LL}$$

where $R_i \equiv c'_i \Leftrightarrow \psi_i$, for $1 \leq i \leq n$, for some $n \geq 0$, and the list notation is used to represent the conjunction of rungs.

Given this formulation, we can now produce a propositional formula representing the pelican crossing ladder logic program. The resultant propositional formula is shown in Figure 3.1.

$$\begin{array}{l}
[crossing' \iff (req \wedge \neg crossing), \\
req' \iff (pressed \wedge \neg req), \\
tlag' \iff ((\neg crossing') \wedge (\neg pressed \vee req')), \\
tlbg' \iff ((\neg crossing') \wedge (\neg pressed \vee req')), \\
tlar' \iff crossing', \\
tlbr' \iff crossing', \\
plag' \iff crossing', \\
plbg' \iff crossing', \\
plar' \iff (\neg crossing'), \\
plbr' \iff (\neg crossing'), \\
audio' \iff crossing']
\end{array}$$

Figure 3.1: An example ladder logic program.

Here we notice that there is exactly one list entry corresponding to each rung of the original ladder logic program. Also we notice that the assignment to a coil gets translated into an equivalence between the coil and all constructs occurring on the corresponding rung. We can see that all output coils are primed and only assigned to once. The last point we notice is that all primed variables referred to on the left hand side of a rung have previously been defined in the ladder logic program. This shows that our modelling approach exactly captures the nature of ladder logic programs used by Invensys.

3.3 Representation via Automata

In the previous section, we introduced a method of modelling the syntax a ladder logic program using a propositional formula. Given that such a ladder logic program is iterated continually, we can model the behaviour of the system by considering various variable values after each execution. Therefore, the propositional formula we obtain via our modelling approach corresponds to a transition function between states of the interlocking system. For this reason we introduce an automaton theoretical approach to modelling that allows us to capture the operation of a Westrace interlocking.

To allow us to formulate an automaton representation of a Westrace interlocking, we shall first extend the variable sets and notions introduced earlier. We use the two functions *prime* and *unprime* we introduced earlier. These functions enable us to speak about two versions of variables. One set of these variables, namely the unprimed set, will correspond to the variables before an iteration of the ladder logic program. The other set, namely the primed set, will correspond to the new values of the variables after an iteration of the ladder logic program. To motivate this consider the simple automaton in Figure 3.2.

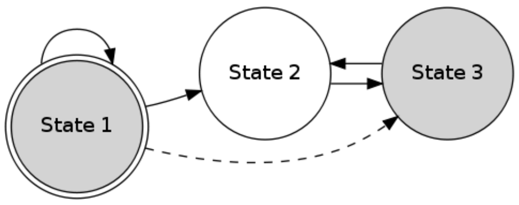


Figure 3.2: A simple automaton, illustrating the need for primed variables.

If we wish to reason about the dashed transition of the automaton given in Figure 3.2, then we need to be able to speak about two sets of state variables. One set being the state variables used as an input to the transition, and one set representing the output state of the transition. Similarly, if we wish to reason about the execution of a ladder logic program, then we need to reason about all variables that occur in that ladder logic program. Given that we want our ladder logic propositional formula to represent a transition between system states, we need to be able to reason about such pairs of states. That is one state that defines the inputs and remembered coils that will be used in our transition, and one state that defines the results of the transition given these inputs. To allow us to speak about such pairs of states in our automaton, we need to introduce the notion of a paired valuation. A paired valuation will simply be a combination of two single valuations.

Definition 3.14 (Paired Valuations)**:** Given a set of inputs I and a set of coils C and valuations $\mu, \mu' : (I \cup C) \to \{0,1\}$ we define $\mu \fatsemi \mu' : (I \cup C \cup I' \cup C') \to \{0,1\}$ where

$$\mu \fatsemi \mu'(x) = \begin{cases} \mu(x) & \text{if } x \in I \cup C \\ \mu'(unprime(z')) & \text{if } z' \in I' \cup C', x = z' \end{cases}$$

Notice that since μ and μ' give values to the same variable set, i.e., $I \cup C$, the definition of paring valuations allows us to change the vocabulary for μ' so that we can speak about the variables $I' \cup C'$. This in turn allows us to speak about pairs of states in our automaton. This pairing function allows us to define the following automaton representing runs of a Westrace interlocking system.

Definition 3.15 (Automaton)**:** Given a ladder logic propositional formula ψ_P over V, define an automaton

$$A(\psi_P) = (S, I_s, \rightarrow)$$

where

- $S = \{\mu \mid \mu : I \cup C \rightarrow \{0,1\}\}$ is the set of states,
- $\mu \rightarrow \mu'$ if $\mu \mathbin{⨟} \mu' \models \psi_P$ defines the transitions, and
- $I_s = \{\mu' \mid \exists \mu : \mu \models \neg I, \mu \mathbin{⨟} \mu' \models \psi_P\}$ gives the set of initial states, where $\neg I$ expands to $\bigwedge_{i \in I} \neg i$ for all $i \in I$.

Remark 3.16: Notice here that ψ_P does not impose any conditions on the values in I'. Thus we gain a non deterministic automaton. Also notice that the automaton $A(\psi_P)$ is finite as it has $2^{|I \cup C|}$ states, where I and C are both finite.

Also, we have the following theorem, which indicates that we can always make a transition from a state in our automaton.

Theorem 3.17: Given an Automaton $A(\psi_P)$ for some ladder logic program P, if μ is a state in $A(\psi_P)$, then there exists a μ' such that $\mu \mathbin{⨟} \mu' \models \psi_P$.

Proof. Let $\mu : I \cup C \rightarrow \{0,1\}$ be a valuation. Define a sequence of valuations $\nu_0, \nu_1, \ldots, \nu_n$ such that,

$$\nu_j : I \cup C \cup \{c'_1, c'_2, \ldots, c'_n\} \rightarrow \{0,1\}$$

for $0 \leq j \leq n$, where

$$\begin{array}{rcll} \nu_j(c) & = & \mu(c) & \text{for } c \in C, \\ \nu_j(i) & = & \mu(i) & \text{for } i \in I, \end{array}$$

and

$$\begin{array}{rcll} \nu_{j+1}(c'_k) & = & \nu_j(c'_k) & \text{for } 1 \leq k \leq j, c_k \in C \\ \nu_{j+1}(c'_{j+1}) & = & \begin{cases} 0 & \text{if } \nu_j \not\models R_{j+1} \\ 1 & \text{if } \nu_j \models R_{j+1} \end{cases} & \text{for } c_{j+1} \in C \end{array}$$

Finally, define $\mu'(c) = \nu_n(c')$ for $c \in C$ and $\mu'(i)$ as arbitrary for $i \in I$. $\square$

Now we have defined our automaton, we shall also define the useful notion of reachability in an automaton.

Definition 3.18 (Reachable)**:** A pair of valuations $\mu \mathbin{⨟} \mu'$ is reachable with respect to an automaton $A(P) = (S, I, \rightarrow)$ for some ladder logic program P, if there exists a series of transitions $\mu_0 \rightarrow \mu_1 \rightarrow \cdots \rightarrow \mu \rightarrow \mu'$ such that $\mu_0 \in I$.

The usefulness of this notion will become more apparent later in the document, but as an example automaton gained via this definition, consider the automaton we gain from our pelican crossing propositional formulae shown in Figure 3.3.

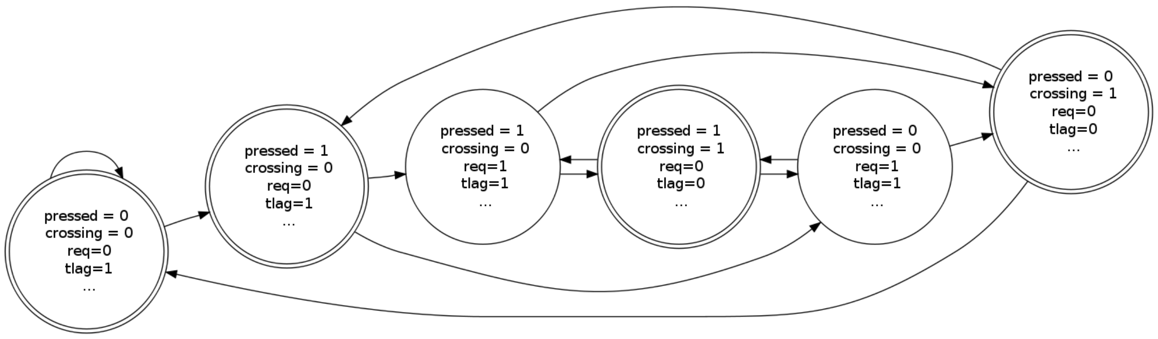


Figure 3.3: An automaton theoretic modelling of the ladder logic program for a pelican crossing.

This automaton shows all the reachable states of the pelican crossing example ladder logic program. Initial states are represented using double a double circle. Here we have omitted unreachable states as in the following chapters we will not be interested in the unreachable states. Also the diagram would be much larger (2^{12} states!) if all states were included.

3.4 Safety Conditions

Before we begin to discuss verification techniques, we first need to introduce how we will model properties which we would like to show hold in our system. Here we will only be interested in safety properties and not liveness properties. That is using the classification of [MP91] we will only consider properties that say "something bad will not happen", and not properties that say "something good will eventually happen". Previously, work has been completed [Fok96] to formulate classes of safety conditions for a certain railway. Here we shall not try to propose classes of formulae, but instead introduce a general scheme which the safety condition we consider will follow.

In [Kan08], a language was defined for use by Invensys. This language enables the engineers at Invensys to speak about safety conditions in a fairly informal, non-specific manner. A tool was implemented to translate safety conditions in this informal language into concrete conditions in propositional logic. The tool will be covered in more depth in Section 4.3. Here we define the general format of such a propositional safety condition for a given ladder logic program.

Definition 3.19 (Safety Condition)**:** A safety condition φ for a ladder logic propositional formula ψ_P over variables $V = I \cup C \cup C'$, is defined as:

$$\varphi \in PROP$$

such that for all $v \in vars(\varphi)$, it holds that $v \in vars(\psi_P)$.

This definition is rather intuitive, and allows for a safety condition to range over pairs of states in our automaton. That is, a safety condition can speak about both input and output values of a ladder logic program. Using our pelican crossing ladder logic program as an example, a safety condition we may want upheld is that “A traffic light always shows a single aspect”. This would then be captured by the following propositional formula:

$$(tlag1 \vee tlar1) \wedge not(tlag1 \wedge tlar1) \wedge (tlbg1 \vee tlbr1) \wedge not(tlbg1 \wedge tlbr1)$$

Remark 3.20: Notice that this particular safety condition does not speak about any values from the set C'.

The remainder of this document will now concentrate on showing how we can prove such formulae hold.

3.5 Satisfiability

Here, we shall introduce the Boolean Satisfiability, commenting mainly on the history of SAT and some approaches to solving the problem.

A well known problem in theoretical Computer Science is the Boolean Satisfiability (SAT) problem [BHvMW09] for propositional logic. The problem can be stated as, given a propositional formula φ, does there exist a valuation μ for φ such that

$$\mu \models \varphi.$$

The combinatorial complexity of this problem makes it one of the most famous problems in the $P = NP$ question posed by Stephen Cook in 1971 [Coo71]. The practical and theoretical implications, one of which will be discussed later, of finding a polynomial time algorithm for the SAT decision problem, means that many algorithmic approaches [GPFW96] have been applied to try to solve the problem.

The first such non-trivial approach to solving the SAT problem was given by Davis and Putnam (DP algorithm) in [DP60]. Shortly after this in 1962, the well known DPLL (Davis-Putnam-Logemann-Loveland) algorithm was introduced in [DLL62]. This algorithm is based on the theme of backtracking, and was an extension to the DP algorithm. The backtracking approach was the main algorithmic method towards solving the SAT problem up until the introduction of CDCL (conflict driven clause learning) algorithms [MSLM09] in the 1990’s. Finally, the DPLL algorithm was enhanced further to include a so called “look ahead” feature, and now this algorithmic approaches is commonly referred to as look ahead sat solving [HvM09].

All of the algorithms we have mentioned above are so called “complete algorithms”, that is they always return a “satisfiable” or “unsatisfiable” for a given problem. Given that no polynomial time algorithm for the SAT problem has been found yet, this means that many of these algorithms could take years to complete. For this reason, newer approaches have also included incomplete algorithms, with the main approach to this problem using a local search approach [KSS09]. These algorithms may try to solve a given problem for a given period of time, then simply give up and return a "I don't know" result. Due to the nature of the problems which we will use a SAT solver for, these kind of incomplete algorithms and solvers will be of no use to us. Therefore, we shall mainly use the CDCL solver MiniSat [min09, ES04], the look-ahead solver OKsolver [Kul08] and the model finder Paradox[1] [par09].

Now we have introduced the SAT problem and some SAT solvers, in next chapters we shall introduce how we can use SAT solvers for the verification of ladder logic programs.

[1]Which happens to use Minisat as the underlying proof tool.

Chapter 4

A Survey of Verification Approaches for Ladder Logic

Contents

In this chapter, we introduce exactly what it means to verify a ladder logic program. We do this by first introducing the general verification problem, then showing how we concretely plan to verify ladder logic. Following this, we introduce some related work, including a detailed account of work by Kanso [Kan08, KMS08]. In this work, Kanso applies an inductive verification approach, proposing and developing tools which we have extended further.

4.1 Verification

Ensuring that a system is correct is a vital part of modern Computer Science. The need for both correct hardware and software grows greater and greater as the use of control systems expands. Whether it be verification of interfaces on medical devices [Thi09], or verification of electronic payment systems [KR09], the use of formal methods and verification is becoming more predominant everyday. The main limitation in the verification of such systems, is caused by the ever growing size and complexity of computer systems, leading to verification problems that just can not be solved feasibly with current techniques. The general goal of verification, is to show that a computer system does or does not exhibit certain properties. Therefore verification techniques provide us with a way to reason about system behaviour, and often gives us new sights into how a system actually behaves.

4.1.1 Classical Verification

Verification of computer systems can be split into two main fields, verification of software [Kro09] and verification of hardware [Kro99]. In the former, it is usually the code from the software that is used as a model, or a model may be constructed for the system from a users point of view. Whereas the latter, usually consists of modelling the given hardware device using a certain modelling language such as CSP [Hoa85], Casl [Mos04] or some form of timed logic [PP06].

Overall the general verification process can usually be summarised using the following three steps:

1. Construct a model of the software or hardware to be verified.
2. Using the original system specification (usually given in natural language), derive properties that the system should/should not fulfil.
3. Perform a verification technique to check if the model of the system meets the desired properties.

In step one of the process, a common technique is to construct some form of automaton theoretic model [PP06]. Then the verification problem is an analysis over the states of the constructed automaton. This is in line with the modelling approach we have taken for ladder logic in Chapter 3. Next, the properties to be upheld by the system are specified using some form of logical language. Typically, properties are specified using a temporal logic [MP91] allowing aspects such as time to be captured within the conditions. Here, as shown in Chapter 3, we have use propositional logic simply because it is a powerful enough language to capture all of the properties Invensys are interested in. Finally, the third step of the process, the verification step, is usually completed using tools such as model checkers or theorem provers. In this project we will use the highly successful technique of SAT-based model checking [Kan08, GRV08, BC00, SSS00]. We will concentrate on such techniques for the remainder of this document.

4.2 Verification of Ladder Logic

We now show how the three main steps of verification outlined above, can be applied to ladder logic. To do this we shall illustrate with examples how we can check to see if a given safety condition holds in our automaton model of a ladder logic program. Finally, we will look into some related approaches to the verification of ladder logic.

In Chapter 3, we have shown how we can model a ladder logic program as a propositional formula, which in turn is used as a transition function for an automaton. We have also explained the format of safety conditions and how they can also be modelled using propositional formulae. Therefore given a ladder logic program P, and a safety condition φ, we want that φ is upheld in all reachable system states, that is:

Definition 4.1 (Verification Problem)**:** Given a ladder logic program P and given a safety condition φ, we say that

$$A(\psi_P) \models \varphi$$

iff φ holds for all reachable states in $A(\psi_P)$. Here

- ψ_p is a propositional formula modelling the ladder logic program P.
- $A(\psi_P)$ is the automaton constructed following the methods of Chapter 3 using ψ_P as a transition function.
- φ is a safety condition.

To show this, it suffices to show that for every reachable pair of states $\mu; \mu'$ in $A(\psi_P)$ we have:

$$\mu; \mu' \models \varphi.$$

To illustrate this example consider the following automaton constructed from our pelican crossing ladder logic program.

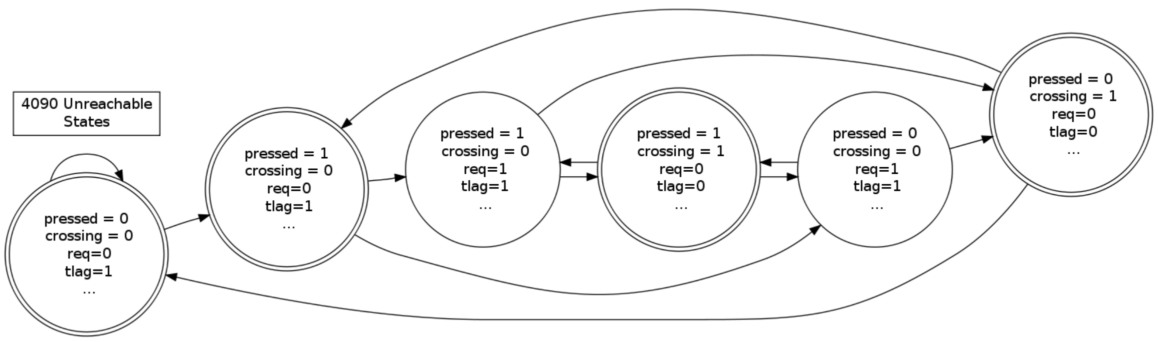

4090 Unreachable States
pressed = 0 crossing = 0 req=0 tlag=1 ...
pressed = 1 crossing = 0 req=0 tlag=1 ...
pressed = 1 crossing = 0 req=1 tlag=1 ...
pressed = 1 crossing = 1 req=0 tlag=0 ...
pressed = 0 crossing = 0 req=1 tlag=1 ...
pressed = 0 crossing = 1 req=0 tlag=0 ...

Figure 4.1: An automaton showing reachable states for our pelican crossing example.

Here we have 6 states that are reachable. This means that as long as the interlocking starts correctly in an initial state, then the interlocking system will never reach a state other than these 6. Therefore, to show that a safety condition holds for an entire ladder logic program, we have to show that it holds in all reachable states of our automaton. Algorithms and techniques to find all such pairs of reachable states and to check them against the safety property will be discussed in Chapters 5 and 6.

4.2.1 Related approaches

We now move on to discuss some previous approaches to the verification of ladder logic programs. We will discuss three approaches, the last of which we shall direct most attention to, as the tools developed in this approach will form the basis of our approach.

The earliest approach which we will look into is that of Wan Fokkink and Paul Hollingshead in [FH98]. In this article, Fokkink and Hollingshead describe the role of control tables within railway signalling, and show how such control tables are encoded into ladder logic programs. They show, in an approach similar to [Kan08] and ours, how a ladder logic program can be modelled using a propositional formula. They then consider how certain invariants can be used to strengthen the propositional model. Here we shall not focus on such invariants, but just comment later about some invariants that could be used to improve the verification process. Finally, a so called slicing algorithm is presented (without proof of correctness) and it is shown how such an algorithm can be used to reduce the complexity of verification. In chapter 7, we will discuss a similar slicing algorithm, giving both an implementation of such an algorithm and a correctness proof that the slicing algorithm is correct with regards to our verification method.

The next approach to verification of ladder logic programs we comment on is that of Zoubek et. al. in [ZRK03]. In this work, a ladder logic program is modelled using a finite state timed automaton, and then the Uppaal [upp09] model checker is run on the automaton with a given property to be verified. We have chosen a different modelling approach, as the ladder logic programs used by Invensys do not explicitly use time. Therefore, we do not need to add the complexity of time to our automaton model. The paper then continues to give a detailed description of example verification scenarios, and shows how the approach outlined can be successfully applied.

Finally we will now cover, in some detail, a third approach to verification.

4.3 Verification of Westrace Interlockings – Kanso '08

Recently, in 2008, a highly successful approach to SAT-based verification of ladder logic programs was given by Kanso in [Kan08, KMS08]. Also working in co-operation with Invensys, Kanso proposed and implemented a tool to perform induction based verification of ladder logic programs for Westrace interlockings. The approach, which we will now outline, was successfully applied to the verification of a small real world interlocking.

4.3.1 The Modelling Approach and Verification Process

The modelling approach taken by Kanso, was highly similar to the modelling approaches we have already mentioned. A ladder logic program was modelled as a propositional formula, and safety conditions were modelled using first order formulae. Here we note that Kanso implicitly used the notion of a finite automaton, refraining from an explicit construction of an automaton corresponding to a ladder logic program. In his thesis, Kanso splits the problem of verification into two main parts, both of which are illustrated in Figure 4.2.

- **Problem 1 – Verification Method**
  The first problem that is tackled by Kanso, is to come up with a suitable automatic verification technique. To solve this problem, Kanso, using Haskell [Hut07], implements a parser for ladder logic programs. This parser, given a ladder logic file used by Invensys, creates a propositional formula in an abstract syntax form. From this point on we shall use ψ_P to represent the formula gained from parsing a ladder logic

program. Next, the formula ψ_P is combined with a given safety condition φ, the generation of which we will comment on later, to give some formulae to be checked for satisfiability. Using our automaton modelling, the formulae produced by Kanso correspond to the following inductive verification process.

1. Base Case: $\mu \fatsemi \mu' \models \varphi$ for all μ^0, μ, μ' where $\mu^0 \models \neg I$, $\mu^0 \fatsemi \mu \models \psi_P$ and $\mu \fatsemi \mu' \models \psi_P$. That is, given the initial conditions of the ladder logic program are met, after two execution cycles of the ladder logic program, we have that the safety condition is upheld.
2. Induction Step: For all μ, μ', μ'' with $\mu; \mu' \models \psi_P$ holds: If $\mu; \mu' \models \varphi$ and $\mu'; \mu'' \models \psi_P$ then $\mu'; \mu'' \models \varphi$. That is, given that the safety condition holds for some execution of the ladder logic program, then we have that the safety condition is still upheld after the next iteration of the ladder logic program.

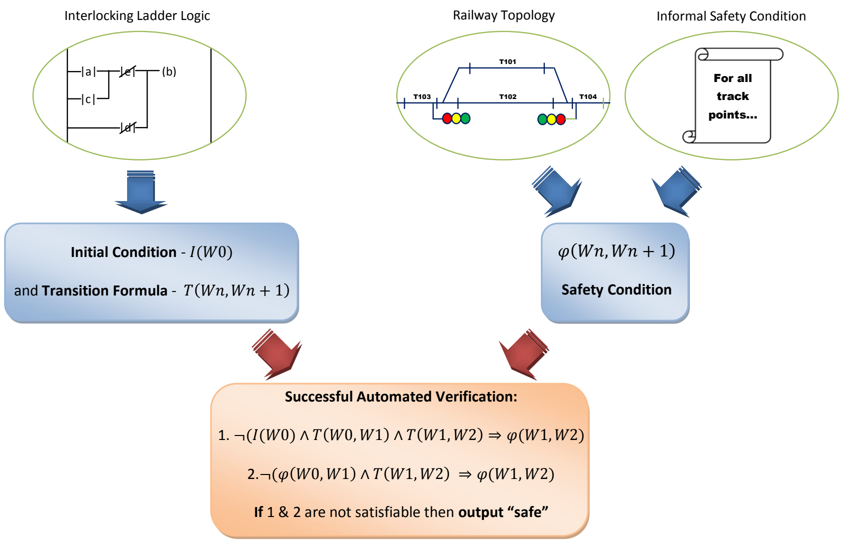

Interlocking Ladder Logic
Railway Topology
Informal Safety Condition
a
e
(b)
c
d
T101
T103
T102
T104
For all track points...
Initial Condition - $I(W0)$
and Transition Formula - $T(Wn, Wn+1)$
$\varphi(Wn, Wn+1)$
Safety Condition
Successful Automated Verification:
1. $\neg(I(W0) \wedge T(W0,W1) \wedge T(W1,W2) \Rightarrow \varphi(W1,W2)$
2.$\neg(\varphi(W0,W1) \wedge T(W1,W2) \Rightarrow \varphi(W1,W2)$
If 1 & 2 are not satisfiable then output "safe"

Figure 4.2: Illustration of the modelling approach taken by Kanso.

Obviously if these two conditions are shown to hold, the the ladder logic program will have been verified, via induction, with respect to the given safety condition. To show these formula hold, Kanso uses a SAT solver, namely the OKsolver [Kul08], as the underlying proof tool. As we have seen, a SAT solver returns whether or not a formula is satisfiable. Hence if we were to formulae corresponding to the above conditions to a SAT solver, then as long as there is at least one pair of states satisfying each formula, the SAT solver will return satisfiable. However this result is of course useless to us as there still may exist other pairs of states where the safety condition is violated. To solve this issue, the formulae involved are negated. This means that if the SAT solver now returns satisfiable, then a pair of states that violate the safety condition has been found. Therefore to verify a property successfully, we want the SAT solver to return unsatisfiable. Finally, if a counter example was found, i.e., one of the negated formulae is satisfiable, then Kanso's tool would give a pictorial situation illustrating the counter example. Figure 4.2 illustrates the approach taken by Kanso. The propositional formulae presented in the diagram are those relating to the conditions stated above. For more details on these formulae, see Section 5.2.

- **Problem 2 – Generating Safety Conditions** The second problem which Kanso solves, is that of generating safety conditions. Kanso provides Invensys with a tool into which they can enter informal safety conditions. The informal safety conditions are required to follow certain syntax rules outlined by Kanso. This provides an easy way for the engineers at Invensys to describe safety properties, and also allows the safety condition to be translated into a logical form. An example of such a safety condition would be, "For all points p (normal(p))", which expresses that all points specified in the track plan should be in normal position. Once such a safety condition has been entered, the tool automatically translates the condition into a first order logic formula, before translating this first order logic formula down into a series of propositional formulae. During this process, all variables are instantiated to contain actually names of variables within the ladder logic program. This is possible as the first order formulae are always considered over a finite model given by the railway topology. For example, in the above safety condition "For all points p" will be instantiated with the name of each point occurring in the railway topology. The final series of obtained propositional formulae can then be used as safety conditions for problem 1.

The combination of these two problems yields a full tool for the verification of ladder logic programs. Again, Figure 4.2 shows the internal structure of the final tool.

Overall, the merits of the approach by Kanso are clear:

1. The verification of a small interlocking ladder logic program is within the range of only minutes.
2. In his thesis [Kan08], Kanso gives a series of safety conditions that have been verified, either successfully, or with the generation of a counter example situation.
3. The whole process is automated into a single tool.

Even though there was a high level of success, the approach taken by Kanso could be improved further. One problem with the tools created, is that in places they are coupled to the single interlocking that was being verified. The second problem is that the tools were creates using a varied code base, consisting of several languages such as Haskell, Prolog and Java. This makes the maintenance and debugging processes highly difficult. The problems given above are mainly based around how the verification software that has been implemented, and with time could all be successfully solved, e.g., making the software generic is possible through various extensions to the existing tools, but now we shall move our attention to another, more fundamental issue with this approach.

4.3.2 Unreachable States and Error Traces

Using the inductive scheme presented by Kanso, the verification process is open to errors due to unreachable states. If we consider the inductive step used in the verification process,

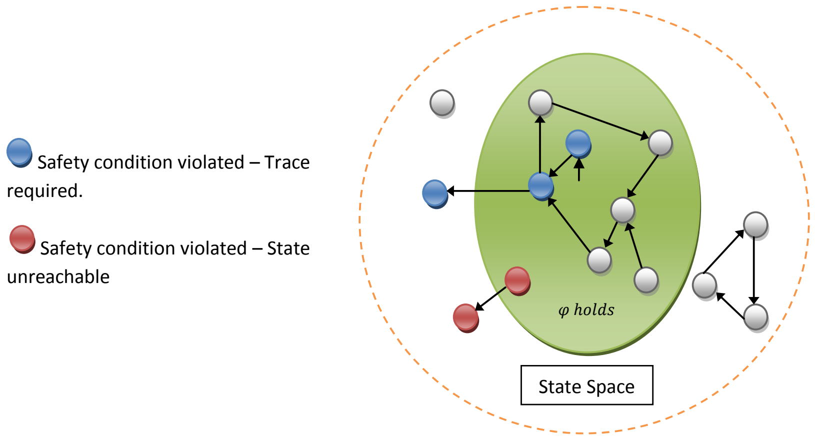

Safety condition violated – Trace required.
Safety condition violated – State unreachable
φ holds
State Space

Figure 4.3: Illustration of state space and violating states.

Induction Step: For all μ, μ', μ'' with $\mu; \mu' \models \psi_P$ holds: If $\mu; \mu' \models \varphi$ and $\mu'; \mu'' \models \psi_P$ then $\mu'; \mu'' \models \varphi$.

we see that it considers all states where the safety condition φ holds. It does not consider whether or not these states are reachable from an initial configuration of the system. This can be seen more clearly in Figure 4.3. This leads to the problem that sometimes the verification process returns a pair of states which violate the safety condition, but this result is incorrect. This is because these states can never be reached in a concrete system run. To solve this problem, either a different verification algorithm will be required, namely one that does not take an inductive approach, or certain invariants could be added to the verification process. We shall not consider invariants any further here, as it is apparent that strong enough variants are simply unknown to the engineers at Invensys. Another fundamental problem with this inductive approach, is that when a valid counter example is found, the state of the system for the counter example can be output by the software, but no error trace to how this state occurred can be given. Thus, in order to exclude such violations, the engineers have to study these examples to quite a detail. In practice, they excluded all of them. However, this turned out to be a rather time consuming and also error prone process. Hence, producing such an error trace is something that the system engineers at Invensys are highly interested in as it would enable them to find the cause of the problem much more easily. To solve the problem of producing an error trace, again a different verification approach will be needed. A more iterative approach to verification is what we will now discuss in the following chapters.

4.4 An Example of Inductive Verification

To illustrate the concept of inductive verification and the problems it poses, we shall now give an example verification of the pelican crossing ladder logic program. For purposes of illustration we have used the CASL specification language [Mos04], along with HETS, the Heterogeneous Tool Set [HET09]. Here we note that the final tool we provide for the verification of ladder logic does not use CASL or HETS, hence we will not discuss the tools any further. The full CASL specifications of all examples given are included for reference in the appendix.

Considering our pelican crossing example, we have parsed the ladder logic into a propositional formula, and specified this propositional formula as a transition in CASL. The specification for the transition function is given in Figure 4.4.

spec TRANSITION[STATE0][STATE1] =
$\quad$ $crossing1 \Leftrightarrow req0 \wedge \neg\, crossing0$
$\quad$ $req1 \Leftrightarrow pressed0 \wedge \neg\, req0$
$\quad$ $tlag1 \Leftrightarrow \neg\, crossing1 \wedge (\neg\, pressed0 \vee req1)$
$\quad$ $tlbg1 \Leftrightarrow \neg\, crossing1 \wedge (\neg\, pressed0 \vee req1)$
$\quad$ $tlar1 \Leftrightarrow crossing1$
$\quad$ $tlbr1 \Leftrightarrow crossing1$
$\quad$ $plag1 \Leftrightarrow crossing1$
$\quad$ $plbg1 \Leftrightarrow crossing1$
$\quad$ $plar1 \Leftrightarrow \neg\, crossing1$
$\quad$ $plbr1 \Leftrightarrow \neg\, crossing1$
$\quad$ $audio1 \Leftrightarrow crossing1$
end

Figure 4.4: Transition function specified in CASL.

From Figure 4.4 we can see that the specification of the transition function is parametrised. This means that the transition specification can be used as a transition between any pair of states in our automaton. Also specified in CASL, as shown in Figure 4.5, is the initial conditions which are used in the definition of our initial states. This specification describes that initially all variables in the ladder logic program should be set to false.

Finally, it remains to specify the safety condition which we would like to be upheld. Here we use the same safety condition as given in Chapter 3, namely that for the traffic lights in our pelican crossing example, we have that "A traffic light always shows a single aspect".

spec INITIAL[STATE0] =
$\quad$ $\neg\, pressed0$
end

Figure 4.5: Initial conditions specified in CASL.

To show that this safety condition is upheld, we attempt to use the inductive verification approach. To do this, we have specified both the conditions, i.e., the base case and inductive step, used by Kanso. The CASL specification of the base case is given in Figure 4.6, and the CASL specification of the inductive step is given in Figure 4.7. Here we note that instead of representing state variables using unprimed and primed versions, we use 1 and 0 respectively.

The specification of the base case condition in Figure 4.6 shows that we firstly import specifications of the variables to be used within verification conditions, i.e., *State*0 and *State*1. It then shows that we instantiate the initial conditions of the system. That is we set all of the variables in *State*0 to be false. Next the specification shows that we instantiate the transition specification with the variable set *State*0 and *State*1. This represents a transition in our automaton from *State*0 to *State*1. Similarly we see this transition between *State*1 and *State*2. Finally, we see a *then %implies* statement which tells us that the axioms stated after this are to be proven from the given specifications. After this statement comes the specification of the required safety condition.

```
spec KansoConditionOne =
     State0
and  State1
and  State2
and  Initial[State0]
then Transition[State0][State1]
and  Transition
     [State1 fit
      req0 ↦ req1 crossing0 ↦ crossing1 pressed0 ↦ pressed1
      tlag0 ↦ tlag1 tlbg0 ↦ tlbg1 tlar0 ↦ tlar1 tlbr0 ↦ tlbr1
      plag0 ↦ plag1 plar0 ↦ plar1 plbg0 ↦ plbg1 plbr0 ↦ plbr1
      audio0 ↦ audio1]
     [State2 fit
      req1 ↦ req2 crossing1 ↦ crossing2 pressed1 ↦ pressed2
      tlag1 ↦ tlag2 tlbg1 ↦ tlbg2 tlar1 ↦ tlar2 tlbr1 ↦ tlbr2
      plag1 ↦ plag2 plar1 ↦ plar2 plbg1 ↦ plbg2 plbr1 ↦ plbr2
      audio1 ↦ audio2]
then %implies
     (tlag1 ∨ tlar1) ∧ ¬ (tlag1 ∧ tlar1) ∧ (tlbg1 ∨ tlbr1)
     ∧ ¬ (tlbg1 ∧ tlbr1)
end
```

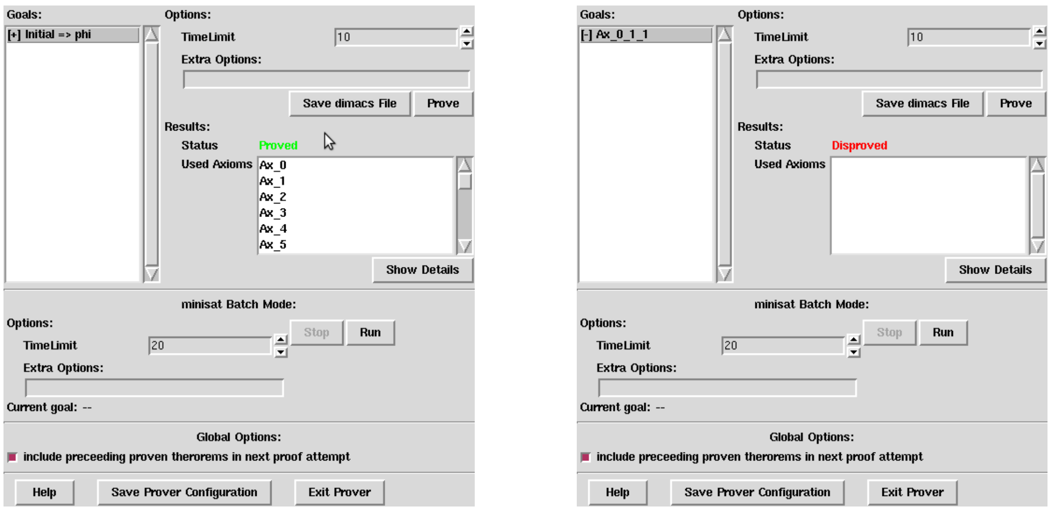
Figure 4.6: Base case verification condition specified in CASL.

In the same style as the base case specification, Figure 4.7 shows the specification of the inductive verification step. In this specification, we notice that there is no longer an instantiation of the initial specification. Instead, there is a specification that describes that the state represented by the variables in *State*0 is a safe one. Then, it is specified that there is a transition from this safe state into a new state *State*1 via instantiation of the transition specification. Finally, we once again see a *then %implies* statement which is followed by the a specification that the safety condition must hold in the new state.

Using the features of HETS, we can now use the MiniSat solver to try to prove these two condition. Figure 4.8 shows that the base case verification succeeds, whilst verification of the inductive step fails. The screenshots in Figure 4.8, show the proof windows in Hets. The state of each proof is indicated by the "Status" field. The Figure shows that, firstly the base case condition has been proved, where as secondly, the inductive step has been disproved.

spec KANSOCONDITIONTWO =
STATE0
and STATE1
and STATE2
then $(tlag0 \vee tlar0) \wedge \neg\, (tlag0 \wedge tlar0)$
$\wedge\, (tlbg0 \vee tlbr0) \wedge \neg\, (tlbg0 \wedge tlbr0)$
then TRANSITION[STATE0][STATE1]
and TRANSITION
[STATE1 **fit**
$req0 \mapsto req1\ crossing0 \mapsto crossing1\ pressed0 \mapsto pressed1$
$tlag0 \mapsto tlag1\ tlbg0 \mapsto tlbg1\ tlar0 \mapsto tlar1\ tlbr0 \mapsto tlbr1$
$plag0 \mapsto plag1\ plar0 \mapsto plar1\ plbg0 \mapsto plbg1\ plbr0 \mapsto plbr1$
$audio0 \mapsto audio1$]
[STATE2 **fit**
$req1 \mapsto req2\ crossing1 \mapsto crossing2\ pressed1 \mapsto pressed2$
$tlag1 \mapsto tlag2\ tlbg1 \mapsto tlbg2\ tlar1 \mapsto tlar2\ tlbr1 \mapsto tlbr2$
$plag1 \mapsto plag2\ plar1 \mapsto plar2\ plbg1 \mapsto plbg2\ plbr1 \mapsto plbr2$
$audio1 \mapsto audio2$]
then %implies
$(tlag1 \vee tlar1) \wedge \neg\, (tlag1 \wedge tlar1) \wedge (tlbg1 \vee tlbr1)$
$\wedge\, \neg\, (tlbg1 \wedge tlbr1)$
end

Figure 4.7: Base case verification condition specified in CASL.



Goals: [+] Initial => phi
Options:
TimeLimit 10
Extra Options:
Results:
Status Proved
Used Axioms Ax_0 Ax_1 Ax_2 Ax_3 Ax_4 Ax_5
minisat Batch Mode:
Options:
TimeLimit 20
Extra Options:
Current goal: --
Global Options:
include preceeding proven therorems in next proof attempt

Goals: [-] Ax_0_1_1
Options:
TimeLimit 10
Extra Options:
Results:
Status Disproved
Used Axioms
minisat Batch Mode:
Options:
TimeLimit 20
Extra Options:
Current goal: --
Global Options:
include preceeding proven therorems in next proof attempt

Figure 4.8: Inductive verification approach fails - The base case verification succeeds, where as the inductive step verification fails.

This result implies that there is a problem with the ladder logic program for our pelican crossing example. However a simple manual analysis of the automaton in Figure 4.1 shows that the safety condition does in fact hold in all reachable states of the automaton[1]. Therefore, the failure of the verification is due to an unreachable state - although the program is correct, Kanso's approach is incapable of proving this.

To improve the verification process, we have to take into account the notion of reachability. Over the next two chapter we will introduce and apply verification approaches to solve this issue. The approaches will also result in a counter example trace being produced if the verification of a given property fails.

[1]This will also be shown via model checking in Chapter 6.

Chapter 5

Bounded Model Checking

Contents

In this chapter, we introduce the main verification technique which we have applied to verify ladder logic programs for interlockings. We begin by introducing the well known field of model checking via an historical overview. We discuss why model checking is required and comment on the success of model checking approaches. We continue to describe a SAT based bounded model checking algorithm, showing how it can be applied to our verification problem via an example application. Finally we state some limitations of bounded model checking as a verification technique.

5.1 The Origin of Model Checking

Model checking [BK08, CGP99] is a formal verification technique that arose from the need to verify concurrent programs [Cla08]. Up until the introduction of model checking in 1981 [CE81] the main technique of verify a system was to construct hand written proofs the properties were upheld by the system in question. Obviously the scalability of such an approach came into question, and in [CE81] Edmund Clarke and Allan Emerson propose a solution to this using a much more model theoretic and mechanised approach. This approach basically consisted of given a finite transition system M, and given a formula F (usually a temporal formula), systematically check for each state s in the given model if $s \models F$. Of course if this can be shown true for all states in the finite transition system, then $M \models F$. Clarke and Emerson also wanted that if there was a state s such that $s \not\models F$, then the model checking process would return a counter example trace. In [CE81], the first algorithm was proposed to solve this problem. Here we will not look further into this algorithm, but will instead see an algorithm based on SAT solving in Section 5.3.

The advantages of using such a model checker soon become apparent, the biggest of which being that:

- The process can be fully automated.
- If a property does not hold, then a counter example trace can be generated.
- The model checker process required no complicated proofs to be written.

Most interesting for us is the ability to fully automate the process, and to create not only counter examples but counter example traces. Both of these properties fulfil the requirements of out industrial partner. Given these advantages, much interest was and still is being taken into the area of model checking. The biggest drawback of model checking, arises from the state space explosion problem [CGJ$^+$01]. Consider as an example our pelican crossing example. The ladder logic program contains only 11 variables, but the automaton contains 2^{12} states. Now consider a typical interlocking ladder logic program for a small railway station. Such a ladder logic program contains around 300 to 500 variables. This causes the automaton to have up to 2^{300} and 2^{500} hundred states. Given this exponential increase in automaton size, the verification time will also increase exponentially. Therefore most of the current research in the field of model checking deals with trying to reduce the state space explosion problem, most commonly through some form of abstraction technique [KP07]. How we have applied techniques to help tackle the problem of state space explosion will be covered in Chapter 7.

5.1.1 Bounded Model Checking

As a result of the state space explosion problem, the full verification of complex system is often not feasible via model checking. This is the reason that in [BCCZ99] Armin Biere et. al. introduce a technique which allows a bound to be put on the model checking process. To do this, the traditional approach of using binary decision diagrams [Ake78] to represent the underlying state space of the automaton is removed and a new approach using SAT solving is proposed. Then, a bound is placed on how many transitions should be considered within the automaton when performing the verification process. This of course makes the procedure incomplete, as only states reachable within this bounded number of transitions are considered. Hence unless all states of the system are reached within this bounded number, the system may still be unsafe. A bounded model checking algorithm only ensures safety up to this given bound. However, suggestions are also given to make the procedure complete, such extension will be discuss and applied to the verification of ladder logic in Chapter 6. The paper itself outlines an entire sound basis for the use of SAT procedures in model checking, and shows how such a procedure reduces the effect of the state space explosion. The paper also illustrates that counter example generation is usually quicker using SAT based model checking over traditional model checking with binary decision diagrams. Finally, a comparison of concrete verification results between non SAT-based and SAT-based model checkers is given.

The introduction of a bound to the model checking process, along with the increased efficiency of the process due to the introduction of SAT solvers, meant that model checking became widely used within industrial applications [CESS08, ADK+05]. The main application of bounded model checking within industry is to find counter examples whilst testing systems. Given the high complexity of the interlocking systems we would like to verify, and given that we would like counter example traces to be produced, we have chosen to take such a SAT-based verification approach.

5.2 Representing State Sequences

The model checking approaches we discuss in the following sections, all rely upon the use of a SAT solver. Thus, we have to give a representation of the state sequences of the automaton under consideration using propositional logic formulae. To do this we define the following variable sets.

Definition 5.1 (Variable Sets)**:** Given a set I of input variables, and set C of coils, we define variable sets:

$$W_j = C^{(j)} \cup I^{(j)}$$

where

- $C^{(j)} = \{c^{(j)} \mid c \in C\}$ and
- $I^{(j)} = \{x^{(j)} \mid x \in I\}$

for $j \in \mathbf{Z}$.

A sequence $W_0, W_1, W_2, \ldots$ of these variable sets represents a state sequence of an automaton $A(\psi)$. For notational convenience, we use the superscript (j) to produce fresh variables. We write W^n for $W_0 \cup W_1 \cup \cdots \cup W_n$, $[W_j/(I \cup C)]$ to denote the substitution where all superscripts are removed, and $[W_{j+1}/(I' \cup C')]$ for the substitution where all superscripts are replaced by primes.

To allow us to map between states in our automaton and the corresponding variable sets, we introduce the following two functions:

Definition 5.2 (*seq*)**:** Given a valuation $\xi : W^n \to \{0, 1\}$, we define

$$seq(\xi) =< \mu_0, \mu_1, \ldots, \mu_n >$$

where

$$\begin{array}{rl} \mu_k : & I \cup C \to \{0,1\} \\ & i \mapsto \mu_k(i) = \xi(i^{(k)}) \\ & c \mapsto \mu_k(c) = \xi(c^{(k)}) \end{array}$$

for $0 \leq k \leq n$.

Here, *seq* gives a state sequence that is represented by a series of propositional variable sets. The second function we define is used for the reverse translation.

Definition 5.3 (*concat*)**:** Given a state sequence $< \mu_0, \mu_1, \dots, \mu_n >$, we define

$$concat(< \mu_0, \mu_1, \dots, \mu_n >) : W^n \to \{0, 1\}$$

- $i^{(k)} \mapsto \mu_k(i)$
- $c^{(k)} \mapsto \mu_k(c)$

for $0 \leq k \leq n$.

Using these variable sets, we can define a propositional formula for state sequences in our automaton. Such formulae will be used within the model checking algorithms we propose. Firstly, we represent a transition using the notation

$$T(W_j, W_{j+1}) \equiv \psi\,[W_j/(I \cup C)][W_{j+1}/(I' \cup C')].$$

Then we define a series of transitions:

Definition 5.4 (Series of transitions)**:** Let ψ_P be a ladder logic formula. We define the propositional formulae

$$Init \equiv (\bigwedge_{i \in I^{(-1)}} \neg i) \wedge T(W_{-1}, W_0) \qquad T_n \equiv \bigwedge_{0 \leq j \leq n-1} T(W_j, W_{j+1})$$

where $n \geq 0$.

Given a ladder logic formula ψ_P, then the formula $Init \wedge T_n$ is "satisfied" exactly by all state sequences $\mu_0, \mu_1, \dots, \mu_n$ of $A(\psi_P)$. More formally: Given a state sequence $\mu_0, \mu_1, \dots, \mu_n$ we construct, using *concat*, an valuation $\xi : W_{-1} \cup W_0 \cup \dots \cup W_n \to \{1, 0\}$, where state μ_j gives the interpretation of W_j for $0 \leq j \leq n$, i.e. $\xi(i^{(j)}) = \mu_j(i)$, $i \in I$, and $\xi(c^{(j)}) = \mu_j(c)$, $c \in C$; $\xi(i^{(-1)}) = 0$, $i \in I$, and $\xi(c^{(-1)})$ such that we reach μ_0 via ψ_P. For this ξ holds: $\xi \models Init \wedge T_n$. Conversely, given a ξ with $\xi \models Init \wedge T_n$ one can decompose it, using *seq* to a state sequences $\mu_0, \mu_1, \dots, \mu_n$ of $A(\psi_P)$.

With these notations in place we can define safety at a specific point in a sequence $W_0, W_1, W_2, \dots$

Definition 5.5 (Safety at step n)**:** Let φ be a safety condition for a ladder logic formula ψ_P, i.e., $vars(\varphi) \subseteq I \cup C \cup C'$. We define the propositional formula

$$\varphi_n \equiv \varphi\,[W_{n-1}/(I \cup C)][W_n/(I' \cup C')],$$

where $n > 0$.

If this formula holds, then we know that the safety condition φ holds for the n^{th} transition. We continue by using these formulae to give a bounded model checking algorithm for ladder logic programs.

5.3 Applying Bounded Model Checking to Ladder Logic

Given the problems arising from the inductive verification approach taken by Kanso (see Section 4.3), we now present algorithms to solve these issues. The algorithms we present use the automaton theoretic modelling we have given in Chapter 3, where to gain the transition function we make use of the ladder logic parser given by Kanso in [Kan08]. Like Kanso, we will also continue to use the successful underlying proof technology of a SAT solver. The algorithms we have used are adaptations of the bounded model checking algorithms given in [SSS00]. Overall we propose two main algorithms which allow the of elimination of unreachable states, and the production of full counter example traces.

5.3.1 Forwards Reachability in k Steps

The first algorithm which we give, shown in Figure 5.1, performs a forwards iteration of the state space. That is, given a automaton $A(\psi_P)$ for some transition function ψ_P, along with some safety condition φ. The algorithm will check that φ holds for up to k transitions from an initial state of the system. Within the algorithm, we make use of the following two formulae:

$$\mathit{Initial} \equiv \mathit{Init} \wedge T(W_0, W_1) \Rightarrow \varphi_1 \text{ and } \mathit{Transition}_n \equiv T_n \Rightarrow \varphi_n.$$

Here *Initial* describes that for all possible initial sates μ we have that $\mu \models \varphi$. $\mathit{Transition}_n$ describes that given any state sequence of length n in the automaton, that φ must hold throughout this state sequence. Below we give a bounded model checking algorithm based on these formulae.

```
$j \leftarrow 1$
if $\neg\mathit{Initial}$ is satisfiable return error trace
$j \leftarrow j + 1$
while $j \leq K$ do
        if $\neg\,\mathit{Transition}_j$ is satisfiable return error trace
        $j \leftarrow j + 1$
return "K-Safe"
```

Figure 5.1: K-step forwards iteration algorithm.

The first step of the algorithm is to initialise a variable j to be one. The variable j will now be used to count how many transitions have been taken in the automaton. The next step in the algorithm is to compute all possible initial states of the automaton, i.e. the state sequence encoded by *Initial*. As *Initial* also encodes that φ must hold in these states, we then check, using a SAT solver that $\neg\mathit{Initial}$ holds. In the next step the algorithm begins to iterate through the state space. At each iteration, the algorithm checks that $\neg\,\mathit{Transition}_j$ holds for the relevant j. This corresponds to checking that the safety condition φ for all possible state sequences of length j If a state is found to violate the safety condition, then a counter example trace to this state will be returned. If no such state is found, then the algorithm will continue to compute until the bound k is reached. If k is reached then the algorithm returns that the system is "k-safe".

Obviously this algorithm removes the possibility of finding unreachable states that violate the safety condition, as the only states visited by the algorithm are the states that can be reached from initial states of the system and a series of transitions defined by the ladder logic program itself. Also the iterative technique taken by the algorithm gives us the ability to produce an error trace if a state that violates the safety condition is found. The issue that the proposed approach immediately raises is that the number of calls to the sat solver in this algorithm is significantly higher than with the approach by Kanso. The inductive verification by Kanso only makes two calls to the SAT solver, one for the initial condition and one for the inductive step. Whereas the algorithm we propose makes a call for every iteration of the algorithm. If the SAT solving times are large for each problem then this could begin to cause problems, especially as k increases. To combat this, when implementing the algorithm, instead of making many calls to the SAT solver, we combine the calls. That is we make *one* call, namely "$\neg(\mathit{Transition}_j \wedge \cdots \wedge \mathit{Transition}_{j+l})$" is satisfiable, for $l > 1$ to the SAT solver.

We now give a backwards reachability algorithm which is also based on this approach.

5.3.2 Backwards Reachability in k Steps

The second algorithm we give to solve the problem of verification of ladder logic programs is a slight variation of the first algorithm. In a similar manner to the first algorithm, the new algorithm will iterate through the states a automaton, but this time in a backwards manner. That is, instead of starting the iteration process from an initial state of the automaton, the iteration process shall start in an error state. Then the algorithm will execute in a backwards manner, terminating either when a initial state of the automaton is reached, or when the the k bound is reached. This algorithm is given in Figure 5.2.

Here, the concept of starting in a violating state may seem a strange one, as usually we want to find if any violating states exist in our automaton. Although if we consider this approach as an extension to the inductive approach taken by Kanso, then things become clearer. Firstly, Kanso's inductive verification approach can be applied, then if a counter example is found, this counter example can be used as a starting state for this backwards iteration algorithm. This would enable the engineers to determine if the counter example was reachable or not, and if it was, then a counter example trace would also be provided for them by the algorithm.

To allow us to formulate an algorithm for this, we define the following,

$$\mathit{TBack}_{m,n} \equiv \bigwedge_{m \leq i \leq n-1} T(W_i, W_{i+1})$$

which represents a series of m transitions backwards from some state occurring at point n. Then we define,

$$\mathit{Violation}_n \equiv T(W_{n-1}, W_n) \Rightarrow \neg\varphi_n \text{ and}$$

$$TransitionBack_{m,n} \equiv Violation_n \wedge TBack_{m,n} \Rightarrow \neg Init.$$

where $m, n \geq 0$ Using these notions for backward transitions, we can define the following backward iteration algorithm.

```
$l \leftarrow$ violating path length
$j \leftarrow 1$
while $j \leq K$ do
    if $TransitionBack_{j,l}$ is satisfiable return error trace
    $j \leftarrow j + 1$
return "K-Safe"
```

Figure 5.2: K-step backwards iteration algorithm.

From this point onwards, we ensure that the solutions we propose and apply in Chapter 6 and Chapter 7 also apply to such a backwards approach.

5.4 Example Application of Iteration Algorithms

To show how our iterative approach to verification is successfully in both excluding unreachable states and giving counter example traces, we now give an example of its application to the pelican crossing ladder logic program. As in Chapter 4, we have used the CASL specification language [Mos04], along with HETS, the Heterogeneous Tool Set [HET09]. Also, full CASL specifications of all examples given are included for reference in the appendix.

5.4.1 Forwards Reachability - Incorrect Ladder Logic

```
spec TRANSITION[STATE0][STATE1] =
    crossing1 ⇔ req0 ∧ ¬ crossing0
    req1 ⇔ pressed0 ∧ ¬ req0
    tlag1 ⇔ ¬ crossing1
    tlbg1 ⇔ ¬ crossing1 ∧ ¬ pressed0
    tlar1 ⇔ crossing1
    tlbr1 ⇔ crossing1
    plag1 ⇔ crossing1
    plbg1 ⇔ crossing1
    plar1 ⇔ ¬ crossing1
    plbr1 ⇔ ¬ crossing1
    audio1 ⇔ crossing1
end
```

Figure 5.3: An Incorrect transition function specified in CASL.

Firstly, we shall consider the production of counter example traces when verification fails. To do this we consider an incorrect version of the pelican crossing ladder logic program as given in [Kan08]. The CASL specification of the transition function obtained from this incorrect ladder logic program is given in Figure 5.3.

```
spec ForwardsIteration =
          State0
and  State1
and  State2
then Transition[State0][State1]
and  Transition
     [State1 fit
      req0 ↦ req1 crossing0 ↦ crossing1 pressed0 ↦ pressed1
      tlag0 ↦ tlag1 tlbg0 ↦ tlbg1 tlar0 ↦ tlar1 tlbr0 ↦ tlbr1
      plag0 ↦ plag1 plar0 ↦ plar1 plbg0 ↦ plbg1 plbr0 ↦ plbr1
      audio0 ↦ audio1]
     [State2 fit
      req1 ↦ req2 crossing1 ↦ crossing2 pressed1 ↦ pressed2
      tlag1 ↦ tlag2 tlbg1 ↦ tlbg2 tlar1 ↦ tlar2 tlbr1 ↦ tlbr2
      plag1 ↦ plag2 plar1 ↦ plar2 plbg1 ↦ plbg2 plbr1 ↦ plbr2
      audio1 ↦ audio2]
      .
      .
then %implies
     (tlag1 ∨ tlar1) ∧ ¬ (tlag1 ∧ tlar1) ∧ (tlbg1 ∨ tlbr1)
     ∧ ¬ (tlbg1 ∧ tlbr1)
          .
          .
     (tlag6 ∨ tlar6) ∧ ¬ (tlag6 ∧ tlar6) ∧ (tlbg6 ∨ tlbr6)
     ∧ ¬ (tlbg6 ∧ tlbr6)
end
```

Figure 5.4: Forward iteration verification approach specified in CASL.

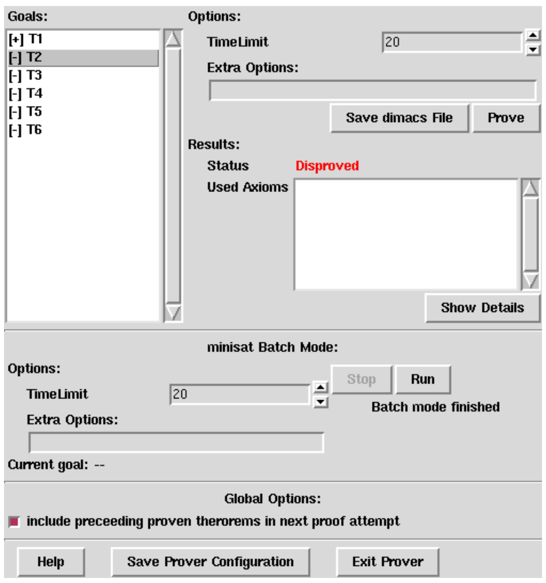
In a similar manner to Chapter 4, a specification capturing the initial conditions of the system has also been created. This can be found in the Appendix. We have also created a specification for our new iterative verification approach, a snippet of this specification is given in Figure 5.4. In this specification, we use the CASL keyword *fit*. This is used to perform a substitution of variables, to allow the reuse of the specified transition function.

This specification shows how we encode a series of transitions using the instantiation of the transition specification over different variable sets. In the example above, a total of five transitions are made, some of which have been removed to shorten the presentation of the example. Also we see that as with the inductive approach, we have imported the specification outlining the initial conditions. Finally, after the *then %implies* statement, we see a series of conditions to be proved. Here there is one safety condition referring to each state visited along the transitions taken through our automaton. Figure 5.6 shows that the verification of the second transition fails in Hets.

Given that this verification fails, we are able to construct a counter example trace. The counter example trace below has not been created using Hets, as the tool does not currently support it. The counter example trace has been generated from the model finder Paradox [par09]. This tool is the what we use later as part of our implementation of a verification process for ladder logic programs.

```
[pressed0=1, crossing0=0, req0=0, tlag0=1,
 tlbg0=1, tlar0=0, tlbr0=0, plag0=0, plbg0=0,
 plar0=1, plbr0=1, audio0=0, pressed1=1, crossing1=0,
 req1=1, tlag1=1, tlbg1=0, tlar1=0, tlbr1=0,
 plag1=0, plbg1=0, plar1=1, plbr1=1, audio1=0]
```

Figure 5.5: A generated counter example trace.


Goals:
[+] T1
[-] T2
[-] T3
[-] T4
[-] T5
[-] T6

Options:
TimeLimit 20
Extra Options:

Results:
Status Disproved
Used Axioms

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

Global Options:
include preceeding proven therorems in next proof attempt

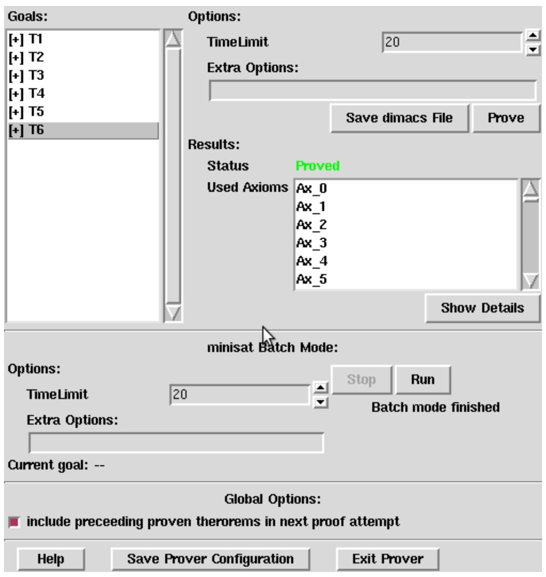
Figure 5.6: Forwards reachability - verification of incorrect ladder fails.

As we can see from the counter example trace in Figure 5.5, values have been given for each execution of the ladder logic program, starting from the initial conditions of the system, going all the way through to where the counter example has occurred, namely after one iteration. This kind of trace provides the engineers at Invensys with much needed information for the debugging process. Obviously a similar trace can also be constructed using our backwards iteration approach, only this trace would begin at an error state and end in the initial state.

5.4.2 Forwards Reachability - Correct Ladder Logic

In the previous section, we have shown how both the forward and backward iteration approaches we have proposed can successfully yield a counter example error trace when an error is found. Here we shall show that the forward iteration approach we have given can also verify a ladder logic program if the ladder logic upholds the safety condition. To do this, we will once again consider the correct ladder logic program we have been using as a running example throughout. The specification of the transition function in CASL has already been given in Figure 4.4. Also, the CASL specification of the forwards iteration approach is similar to that given in Figure 5.4, only we instantiate the new correct transition function instead. The full specification can be found in the appendix, as here we shall just show the result of the verification, and give some comments on it.

Figure 5.7 shows the successful verification of up to five transitions, i.e., six states, of the correct pelican crossing ladder logic program. Through manual meta reasoning, i.e., by considering the maximum length, loop free series of transitions required to check every state in the automaton in Figure 4.1, we have confirmed that this result implies safety of the whole system. Therefore, the iterative approach can also be used to verify correct ladder logic programs.


Goals:
[+] T1
[+] T2
[+] T3
[+] T4
[+] T5
[+] T6

Options:
TimeLimit 20
Extra Options:

Results:
Status Proved
Used Axioms Ax_0 Ax_1 Ax_2 Ax_3 Ax_4 Ax_5

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

Global Options:
include preceeding proven therorems in next proof attempt

Figure 5.7: Forwards reachability - Successful verification of correct ladder.

Obviously, the major downfall, is that to ensure the whole state space has been verified, manual meta reasoning is required. On such a small example, this is relatively easy as the automaton can be created by hand. Considering a real interlocking program then there is in the region of 2^{300} to 2^{500} possible reachable states, and hence manual reasoning about such an automaton would be impossible. For this reason, an automated approach to knowing when the full state space of a system has been checked is desired. In the next chapter, some techniques which enable this reasoning to be automated will be discussed.

Chapter 6

Unbounded Model Checking

Contents

Given the problems that we have shown bounded model checking to have (see Chapter 5), we now introduce two methods to solve these problems. Both methods will allow us to automate a check to show when the reachable state space of an automaton has been reached. The first approach we consider involves adding an explicit inclusion formulae to the verification process. Whereas the second approach will adapt the format of the iteration algorithms given in Section 5.3.1. Both approaches will require the notion of loop free paths through an automaton to be formulated. For this reason, we shall first introduce some motivation of why we need such a loop free property, before explaining how we can formulate such a property in propositional logic. Finally, we show the use of this formulation within our two approaches to gain a fully automatic verification process for Westrace interlockings.

6.1 Loop Free Paths

The property of a loop free path will be required for the complete model checking approaches we consider later. Here, we firstly define what we mean by "path" and "loop free", before motivating why the notion of a loop free path is needed. Finally we show how to formulate the property of loop freedom in propositional logic.

From this point onwards, we will use the notion of a path, by this we mean:

Definition 6.1 (Path)**:** Given an automaton A, a path R of length n is defined as a sequence of (n+1) states $\mu_0, \mu_1, \ldots \mu_n$ which are in transition relation:

$$R = \mu_0 \rightarrow \mu_1 \rightarrow \cdots \rightarrow \mu_n$$

Given this we can extend the notion of a path to that of a loop free path, that is

Definition 6.2 (Loop Free Path)**:** Given an automaton A, a loop free path R of length n is defined as:

$$R = \mu_0 \rightarrow \mu_1 \rightarrow \cdots \rightarrow \mu_n$$

where for all $0 \leq i < j \leq n$, $\mu_i \neq \mu_j$.

That is a path is loop free if all the states along the path are unique. Using these notions, we can motivate the need for loop free paths within the verification process.

6.1.1 Why Loop Freedom

The forward iteration approach to verification introduced in Section 5.3.1, is based on the process of verifying an increasing number of transitions through an automaton. When we say that we have verified, for example, up to five transitions in an automaton, we mean that we have verified all concrete paths of length five through the automaton. This will include both paths containing loops, i.e., paths containing duplicated states, and paths without loops[1]. To say that we have verified the entire reachable state space of an automaton, means that we have verified a series of transitions equal to the length of the longest loop free path occurring in the automaton. For example, in our pelican crossing automaton given in Figure 4.1, we are required to verify five transitions to cover the reachable state space. This is because the longest loop free path through the automaton has a length of five. To be able to automate a check to know when all reachable states have been verified, we of course need a method of reasoning about the longest path to occur in the automaton. The reason why we need to reason not only about the longest path, but the longest loop free path arises from the way in which we use the SAT solver as our verification tool.

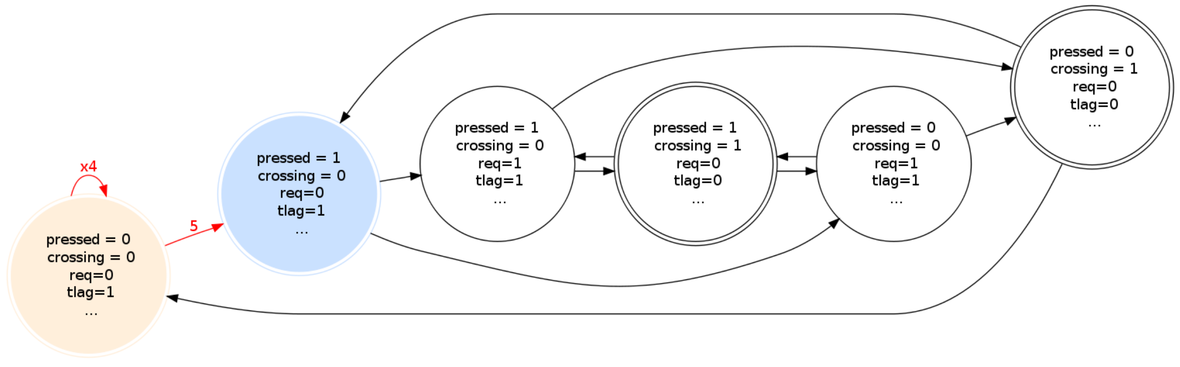


Figure 6.1: Illustration of a path of length five that may be return by a SAT solver. Obviously with such a path, not all states have been verified in this concrete instance.

[1]assuming such a path exists.

Consider the process the SAT solver takes when we give it a formula to check for satisfiability. When we call the SAT solver with our verification problem, i.e.,

$$\neg(Init \wedge Transition_k),$$

we ask is there any path of length k such that along that path, the safety condition φ is violated. Of course the SAT solver now checks every possible path for such a violation. Therefore, if we were to somehow ask the SAT solver "does k transitions verify every reachable state", it would always return a counter example to this question even if every state had been verified. The counter example would be a path of length n but the path would contain loops, and hence not every state is verified in this concrete counter example path. This situation is illustrated in Figure 6.1. What we really need to solve this problem is a way of telling the SAT solver to ignore paths with loops. This situation is illustrated in Figure 6.2, where the longest loop free path is highlighted, and we can see that this path passes through all reachable states in the automaton.

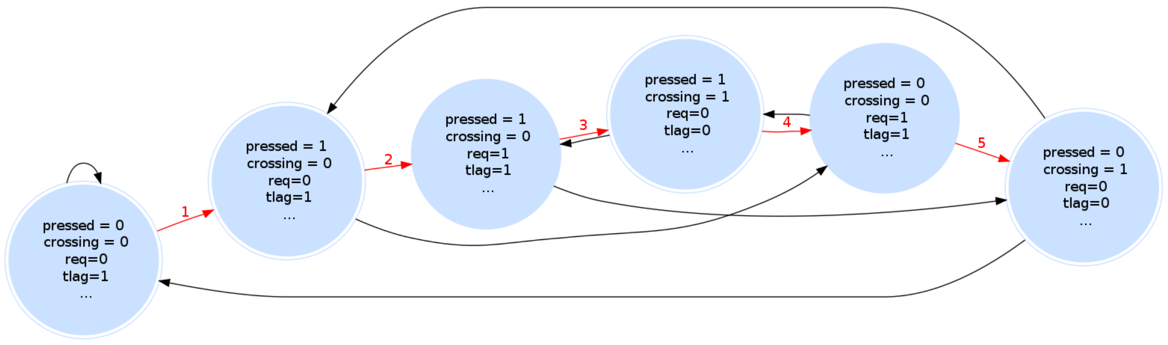


Figure 6.2: Illustration of a path of length five through the automaton. All reachable states would have been verified via this path.

Given this problem of using a SAT solver, no matter how we formulate the question "have all reachable states been verified?", we will need to include a condition telling the SAT solver to consider loop free paths only. We shall show how this can be done in the following section.

6.1.2 Formulating Loop Freedom in Propositional Logic Loop

We now give a propositional formula to express the requirement of loop freedom within a path.

For a state sequence encoded by $W_0, W_1, \ldots, W_n$ we define:

$$LF_n \equiv T_n \wedge (\bigwedge_{0 \leq k < l \leq n} \neg(W_k \Leftrightarrow W_l))$$

where $(W_k \Leftrightarrow W_l) \equiv \bigwedge_{i \in I} i^{(k)} \Leftrightarrow i^{(l)} \wedge \bigwedge_{c \in C} c^{(k)} \Leftrightarrow c^{(l)}$; $k, l, n \geq 0$.

LF_n describes the state sequences of length n of an automaton which are "loop free", i.e. the states appearing in the sequence are pairwise different. We can now continue to use this notion of path loop freedom within our automated approaches to unbounded verification.

6.2 Explicit Inclusion Check

The first approach we discuss to completely automate the model checking approach is to use an explicit inclusion check to indicate when we have verified all reachable states. To do this, we introduce the an explicit formula to check to see if inclusion has been reached. To give such a propositional formula, we would like to somehow capture the First order formula for all paths of length $n+1$ there exists a path of length n such that all the states in the longer path also occur in the shorter path. The basic way to encode such a property in propositional logic, would be to simply unfold the definitions of both the for all and exists statements. Considering that to unfold the exists statement alone would mean enumerating all possible paths the automaton could contain, this approach quickly becomes not feasible. For this reason, the approach we now present does not give a tight bound on when inclusion is reached, but instead an approximation. To define out inclusion check more efficiently than the brute force method, we can make use of the path loop freedom formulation given earlier.

Definition 6.3 (Inclusion Check)**:** Given an automaton A constructed from a transition function ψ_P, inclusion has been reached when we find an $n > 0$ such that the following holds:

$$LF_n \Rightarrow \bigvee_{0 \leq i \leq n} W_{n+1} \Leftrightarrow W_i$$

where W_{n+1}, W_i are variable sets.

If the inclusion check property evaluates to true then it describes that we have a loop free path of length n, and that adding another transition to this path means that we have to take a loop. I.e., the new state μ_{n+1} in the path is equivalent to some previous state we have already seen along the path. This is exactly the check we need to include in our call to the SAT solver. Once again, if we consider that we would like to minimise the number of calls to the SAT solver, then we have to consider implementation details of such an algorithm. Like with the forwards iteration algorithm, we can describe a series of transitions using a single formula. That is, to encode a series of $n+1$ transitions along with a loop freedom check, we can use the following formula.

$$\mathit{Inclusion}_n \equiv \neg I(W_0) \wedge T_{n+1} \wedge LF_n \Rightarrow \bigvee_{0 \leq i \leq n} (W_{n+1} \Leftrightarrow W_i).$$

This formula captures exactly the check we would like to use to see if all reachable states have been verified. This is shown by Theorem 6.4.

Theorem 6.4 (Path condition implies inclusion)**:** Let our formula encoding that inclusion has been reached hold, that is

$$\models \neg I(W_0) \wedge T_{n+1} \wedge LF_n \Rightarrow \bigvee_{0 \leq i \leq n} (W_{n+1} \Leftrightarrow W_i)$$

then we have that $B_{n+1} \subseteq B_0 \cup B_1 \ldots \cup B_n$ where $B_0 = I_s$, i. e. the initial states, and $B_{i+1} = \{\mu' \mid \mu \,\S\, \mu' \models \psi_P, \mu \in B_i\}$ for, $1 \leq i \leq n$.

Proof. Let μ_{n+1} be a state in B_{n+1}, then there exists a path

$$\mu_o \rightarrow \mu_1 \rightarrow \cdots \rightarrow \mu_n \rightarrow \mu_{n+1}$$

such that for all $0 \leq i \leq n+1$ it holds that $\mu_i \,\S\, \mu_{i+1}$ and $\mu_0 \models \neg I$. If we now combine $\mu_0, \mu_1, \ldots \mu_{n+1}$ into

$$\nu : W_0 \cup W_1 \cup \ldots \cup W_{n+1} \rightarrow \{0, 1\}$$

as our formula encoding inclusion is a tautology, we know that

$$\nu \models \neg I(W_0) \wedge T_{n+1} \wedge LF_n \Rightarrow \bigvee_{0 \leq i \leq n} (W_{n+1} \Leftrightarrow W_i).$$

Now, given that $\mu_0 \rightarrow \mu_1 \rightarrow \cdots \rightarrow \mu_{n+1}$ is a path, we know that the formulae $\neg I(W_0)$ and T_{n+1} both hold. Therefore if,

- LF_n holds, then for some $i \leq n$ we have

  $$\mu_0 \rightarrow \mu_1 \rightarrow \cdots \rightarrow \mu_i = n + 1.$$

  As a consequence of this, we have that $\mu_{n+1} \in B_i$ for some $i \leq n$. Hence $\mu_{n+1} \in B_0, B_1 \cup \ldots \cup B_n$.

- LF_n does not hold, then one of the states in the path has been repeated, which shows that the is a shorter path for μ_{n+1}. Hence $\mu_{n+1} \in B_0, B_1 \cup \ldots \cup B_n$.

□

Also, we can illustrate that there will always be a point where the inclusion check becomes true. This is given by Theorem 6.5.

Theorem 6.5 (Eventually the inclusion check becomes true)**:** There exists a n such that $\neg I(W_0) \wedge T_{n+1} \wedge LF_n \Rightarrow \bigvee_{0 \leq i \leq n} (W_{n+1} \Leftrightarrow W_i)$ becomes true.

Proof. Let n be the number of states in our automaton. Let ν be a valuation,

- if $\nu \models \neg I(W_0) \wedge T_{n+1} \wedge LF_n$ then

  $$\nu \models \bigvee_{0 \leq i \leq n} (W_{n+1} \Leftrightarrow W_i)$$

  as there are only n states in the automaton.

- if $\nu \not\models \neg I(W_0) \wedge T_{n+1} \wedge LF_n$ then the above formula holds.

□

Finally, we know that in our automaton, if we reach a fixpoint in the state space, then inclusion has been reached. That is,

Theorem 6.6 (Correctness of inclusion check)**:** Considering our automaton, define $B_0 = I_s$, i. e. the initial states, and $B_{i+1} = \{\mu' \,|\, \mu \fatsemi \mu' \models \psi_P, \mu \in B_i\}$ for, $1 \leq i \leq n$. If $B_{i+1} \subseteq B_0 \cup B_1 \cup \ldots \cup B_i$ then $B_{i+2} \subset B_0 \cup B_1 \cup \ldots \cup B_{i+1}$.

Proof. Let $\mu' \in B_{i+2}$, this implies that there exists $\mu \in B_{i+1}$ such that $\mu \fatsemi \mu' \models \psi_P$. By assumption, we know there exists a $\mu \in B_k$ for some $0 \leq k \leq i$. Thus, $\mu' \in B_{k+1}$, and hence $\mu' \in B_0 \cup B_1 \cup \ldots \cup B_{i+1}$. □

Using our defined inclusion check, we can now slightly modify the forwards iteration algorithm given in Chapter 5. Figure 6.3 shows this new algorithm.

```
$j \leftarrow 1$
if $\neg Initial$ is satisfiable return error trace
do
    $j \leftarrow j + 1$
    if $\neg Transition_j$ is satisfiable return error trace
while $Inclusion_j$ is unsatisfiable
return "Safe"
```

Figure 6.3: Unbounded forwards iteration algorithm via the use of an explicit inclusion check.

When comparing the new algorithm given in Figure 6.3 to the original bounded forwards iteration algorithm in Figure 5.1, we can see that the while loop has been changed to a do while loop. Also, the condition for exiting the loop has been changed. In this new version of the algorithm, the check to see if we have reached the given bound has been removed, and has been replaced with a set inclusion check. This means that the algorithm will continually iterate, checking a larger and larger number of states until the set of checked states reaches a fixed point. Hence now the algorithm is unbounded and only terminates if a violation is found or when the entire reachable state space has been successfully verified.

We will now see how this example can be applied to the verification of the pelican crossing ladder logic program. Before moving on, we should make a note about the size of such an inclusion check formula. The formula given above grows very quickly as the number of variables in the ladder logic program being verified increases. If we consider the formula size required to encode ladder with k rungs for n transitions, then we have a formula bounded by $O(kn)$. Now if we consider the formula required to encode loop freedom for a path of length n for the same ladder we gain a formula bounded by $O(kn^2)$. Hence we can see that the size of our overall formula for n transitions with an inclusion check has a space complexity of $O(kn^2)$. Obviously, this may cause space problems[2] as we try to verify large

[2]but not necessarily time problems as $PTIME \subset PSPACE$. This is because you obviously can't use more than $PSPACE$ in $PTIME$.

ladder logic programs over many iterations. Hence in Chapter 7, we shall discuss ways to reduce such formula size during the verification process.

6.3 Application of the Inclusion Check to Ladder Logic

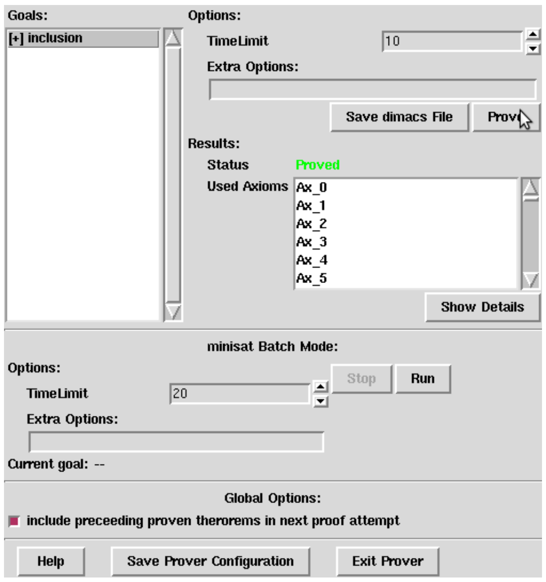
Here we give an example of how the above described inclusion check works successfully on our pelican crossing example. We will consider the correct pelican crossing program specified in Section 5.4.2. Here we shall not repeat this specification, just comment that we have added to the specification one more transition. Also we shall show here how we add a new check to be proven inside the *then %implies* section of the specification. Figure 6.4 shows the new *then %implies* specification.

```
then %implies
        (tlag1 ∨ tlar1) ∧ ¬ (tlag1 ∧ tlar1) ∧ (tlbg1 ∨ tlbr1)
        ∧ ¬ (tlbg1 ∧ tlbr1)
        .
        .
        (¬ (crossing1 ⇔ crossing0) ∨ ¬ (req1 ⇔ req0)
         ∨ ¬ (pressed1 ⇔ pressed0) ∨ ¬ (tlag1 ⇔ tlag0))
        ∧ (¬ (crossing2 ⇔ crossing0) ∨ ¬ (req2 ⇔ req0)
        .
        .
        ∧ (¬ (crossing6 ⇔ crossing5) ∨ ¬ (req6 ⇔ req5)
            ∨ ¬ (pressed6 ⇔ pressed5)
            ∨ ¬ (tlag6 ⇔ tlag5)) ⇒ ((crossing7 ⇔ crossing6)
        .
        .
            ∨ ((crossing7 ⇔ crossing1)
                ∧ (req7 ⇔ req1)
                ∧ (pressed7 ⇔ pressed1)
                ∧ (tlag7 ⇔ tlag1))
end
```

Figure 6.4: An inclusion check for the pelican crossing specified in CASL.

In Figure 6.4, we can see the new inclusion check has been added to the specification. For reasons of space, some variables and statements have been omitted from this figure. This illustrates further how large the formula becomes when adding the inclusion check, even for our simple example. Firstly, in the specification we see the safety conditions to be proven. Then below these safety conditions we see the inclusion property.

This inclusion property can be split into two parts. The first part occurring before the implication symbol represents our loop freedom formula. The second part of the inclusion property occurring after the implication, says that the new state we have reached in our automaton is equal to a state we have seen previously. That is, we must take a loop to have a path of such length.



Goals:
[+] inclusion

Options:
TimeLimit 10
Extra Options:

Results:
Status Proved
Used Axioms Ax_0 Ax_1 Ax_2 Ax_3 Ax_4 Ax_5

minisat Batch Mode:
Options:
TimeLimit 20
Extra Options:
Current goal: --

Global Options:
include preceeding proven therorems in next proof attempt

Figure 6.5: Inclusion check successfully holds.

In a similar manner to earlier, if we now try to prove that the inclusion property is true using Hets and MiniSat, we can see the result is positive, this is shown in Figure 6.5. Following from the example in Figure 6.2 of such a situation, this is exactly the result that is to be expected. Hence, the explicit inclusion check can be used as an automatic way to check that the whole state space of an automaton has been verified.

Given that the Inclusion check formula we have given here is very large, we now discuss a slightly different technique that is once again based on the principle of induction.

6.4 Temporal Induction

The principle behind temporal induction was introduced by Mary Sheeran et. al. in [SSS00]. In this work, Sheeran et. al. introduce the idea of using a SAT solver to check if safety properties are upheld within an automaton. The work introduces the the notion of using a SAT solver to perform bounded model checking, and proposes several algorithms to perform such a verification procedure. One of the algorithms proposed is that of temporal induction. The work then explores how such algorithms perform when applied to the verification of a hardware device known as an FPGA core. Here we shall discuss temporal induction, explaining the notions behind the technique.

Temporal induction is a method that is based on strengthening the inductive approach given by Kanso [Kan08]. As the name suggests, the verification method still consists of two proof steps, namely the base case and the inductive step. However, to improve on the problems arising with normal inductive verification, the base case is strengthened to say that the safety condition holds for a series of n transitions through the system, not only one. Given that we can prove such a base case, we can strengthen the inductive step to assume that we have a series of n transitions where the safety condition is upheld. Of course we need to include a loop freedom property, as otherwise the issues raised in Section 6.1.1 will take effect. To explain this more thoroughly, we shall now describe the temporal induction formulae that will specify our base case and induction step. The two formulae we present are in line to the formulae presented in [ES03].

Consider the original base case formula used by Kanso for inductive verification:

$$I(W_0) \wedge T(W_0, W_1) \wedge T(W_1, W_1) \Rightarrow \varphi(W_1, W_2).$$

Here we see that there is two transitions through the automaton. If we were to replace these transitions by a series of n transitions, then we would also have to check for all n transitions the safety condition holds. To allow us to do this we introduce the following definition:

Definition 6.7: Given an automaton A and a safety condition φ we define the formula: $\mathit{safe}_n = \bigwedge_{1 \leq j \leq n} \varphi_j$. where W_i, W_{i+1} are variable sets.

This formula expresses that the safety condition holds in a series of n transitions through the given automaton. Using this we give the following formula to represent the new base case:

$$\mathit{Base}_n \equiv \text{Init} \wedge T_n \Rightarrow \varphi_n.$$

This formula is highly similar to the formula that we prove holds when performing our forwards iteration algorithm for a bound of size n. If we prove this formula holds, then we know that a serious of n transitions from the initial states of our automaton are safe. Given we know this, and taking into account that we require the loop freedom property, we can now strengthen out induction step formulae to become:

$$\mathit{Step}_n \equiv T_{n+1} \wedge \mathit{LF}_{n+1} \wedge \mathit{safe}_n \Rightarrow \varphi_{n+1}.$$

This formula says that assuming there is a loop free path of length $n + 1$, and that the safety condition holds for the first n transitions of this path, then the safety condition also holds for the next transition in the path, namely transition $n + 1$. We can now construct a verification algorithm using these two formulae as the basis. This algorithm is similar to those presented in [SSS00, CESS08] and is known as the temporal induction algorithm.

```
$n \leftarrow 0$
while true do
    if $\neg \mathit{Base}_n$ is satisfiable return trace
    if $\neg \mathit{Step}_n$ is unsatisfiable return "Safe"
    $n \leftarrow n + 1$
```

Figure 6.6: Temporal induction algorithm.

To consider the effect of this algorithm, we have to consider the effect of the two "if" statements in it. The first "if" statement represents the check to see if our base case holds. That is it checks to see if the safety condition holds for all paths of length n in our automaton. Obviously, as we negate this formula, and then gain a satisfiable result from the SAT solver, then it means that there is a violating state occurring on some path of length n, hence we return a counter example. If there is no such satisfying assignment, then we know that the first n transitions from any initial state in our automaton are safe. The second "if" statement that occurs is concerned with the induction step. Here we ask the question, given a loop free path of length $n+1$ where the first n transitions are safe, are all the $n+1$ transitions safe. If we negate this, and the result is unsatisfiable, the it means that there is no loop free path of length $n+1$ where a violation occurs. However it may also mean that there is no loop free path of length $n+1$ and hence we have checked all reachable states. Therefore if the result is unsatisfiable, we know that we can return that the system is safe. If this result is satisfiable, it means that a violation occurs on some path of length $n+1$, and hence we need to continue to find out if this path is reachable from an initial state. This outline of the algorithm should give an intuitive understanding of why the algorithm is complete and correct. Here we show that the algorithm terminates. For completeness see [SSS00, ES03].

Theorem 6.8: For all ladder logic formulae and safety conditions, temporal induction terminates.

Proof. Let ψ be a ladder logic formula. Let φ be a safety condition. Given that the automaton $A(\psi)$ is finite, we know that for some k all state sequences longer than k include a state twice. Thus, the formula $T_{k+1} \wedge LF_{k+1}$ is unsatisfiable. This implies that $Step_k \equiv T_{k+1} \wedge LF_{k+1} \wedge \varphi_k \Rightarrow safe_k$ is a tautology. Hence $\neg Step_k$ is unsatisfiable. $\square$

We now continue to show how this temporal induction approach can be applied to the verification of ladder logic programs via a small example.

6.5 Application of Temporal Induction to Ladder Logic

In this section, we show how the temporal induction algorithm given in the previous section can be used to verify the example ladder logic program we have use throughout this document. Again we do not repeat any specifications, but the example will use the same transition specification as given in Figure 4.4. Also, we will use the same initial specification as given in Figure 4.5. Finally the full specification is given in the Appendix.

To specify the Temporal induction approach in CASL, would involve creating a series of specifications with an incremental number of transitions in each. For this reason, here we shall just show the specification which leads to the system being verified as safe. Figure 6.7 and Figure 6.8 give the specifications for the base case and inductive step respectively.

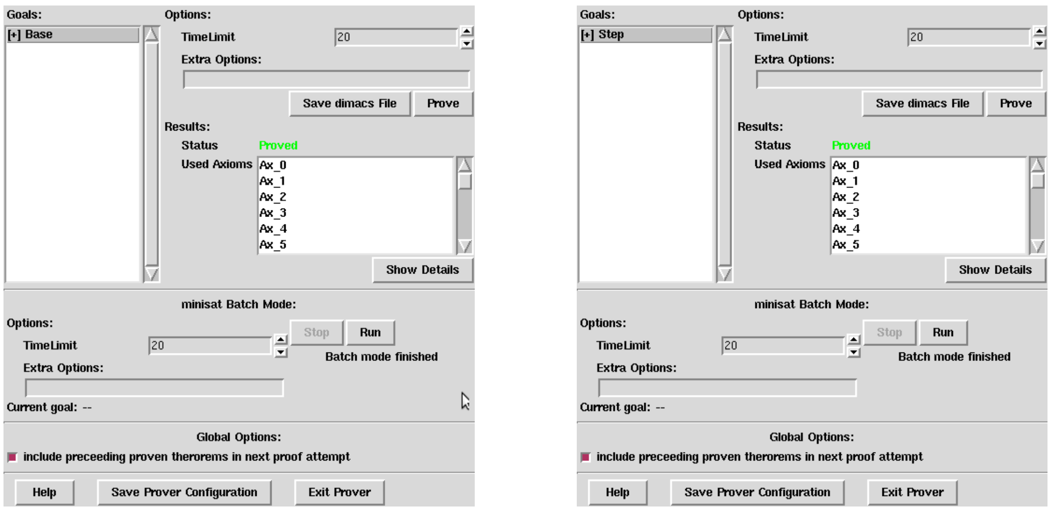
```
spec BASE =
     State0
and  State1
and  Initial[State0]
then Transition[State0][State1]
then %implies
     (tlag1 ∨ tlar1) ∧ ¬ (tlag1 ∧ tlar1) ∧ (tlbg1 ∨ tlbr1)
     ∧ ¬ (tlbg1 ∧ tlbr1)
end
```

Figure 6.7: Base case for temporal induction specified in CASL.

```
spec STEP =
     State0
and  State1
and  State2
then Transition[State0][State1]
and  Transition
     [State1 fit
      req0 ↦ req1 crossing0 ↦ crossing1 pressed0 ↦ pressed1
      tlag0 ↦ tlag1 tlbg0 ↦ tlbg1 tlar0 ↦ tlar1 tlbr0 ↦ tlbr1
      plag0 ↦ plag1 plar0 ↦ plar1 plbg0 ↦ plbg1 plbr0 ↦ plbr1
      audio0 ↦ audio1]
     [State2 fit
      req1 ↦ req2 crossing1 ↦ crossing2 pressed1 ↦ pressed2
      tlag1 ↦ tlag2 tlbg1 ↦ tlbg2 tlar1 ↦ tlar2 tlbr1 ↦ tlbr2
      plag1 ↦ plag2 plar1 ↦ plar2 plbg1 ↦ plbg2 plbr1 ↦ plbr2
      audio1 ↦ audio2]
and  (¬ (crossing1 ⇔ crossing0) ∨ ¬ (req1 ⇔ req0)
      ∨ ¬ (pressed1 ⇔ pressed0) ∨ ¬ (tlag1 ⇔ tlag0))
     ∧ (¬ (crossing2 ⇔ crossing0) ∨ ¬ (req2 ⇔ req0)
        ∨ ¬ (pressed2 ⇔ pressed0) ∨ ¬ (tlag2 ⇔ tlag0))
     ∧ (¬ (crossing2 ⇔ crossing1) ∨ ¬ (req2 ⇔ req1)
        ∨ ¬ (pressed2 ⇔ pressed1) ∨ ¬ (tlag2 ⇔ tlag1))
and  (tlag1 ∨ tlar1) ∧ ¬ (tlag1 ∧ tlar1)
     ∧ (tlbg1 ∨ tlbr1) ∧ ¬ (tlbg1 ∧ tlbr1)
then %implies
     (tlag2 ∨ tlar2) ∧ ¬ (tlag2 ∧ tlar2) ∧ (tlbg2 ∨ tlbr2)
     ∧ ¬ (tlbg2 ∧ tlbr2)
end
```

Figure 6.8: Induction step for temporal induction specified in CASL.

The base case specification in Figure 6.7 simply specifies that a single transition from an initial state upholds the safety condition. Here we have only specified a single transition, because as we will see, this is enough for the step case of our temporal induction approach to be proven. Also, notice that in this particular situation, the base case specification happens to be identical to the base case specification used within the inductive verification approach by Kanso. However, this is pure co-incidence, as with the temporal induction approach the base case needed for a solution could also have contained a much larger series of transitions.



Goals: [+] Base
Options:
TimeLimit 20
Extra Options:
Results:
Status Proved
Used Axioms Ax_0 Ax_1 Ax_2 Ax_3 Ax_4 Ax_5
Options:
TimeLimit 20
Extra Options:
Batch mode finished
Current goal: --
include preceeding proven therorems in next proof attempt

Goals: [+] Step
Options:
TimeLimit 20
Extra Options:
Results:
Status Proved
Used Axioms Ax_0 Ax_1 Ax_2 Ax_3 Ax_4 Ax_5
Options:
TimeLimit 20
Extra Options:
Batch mode finished
Current goal: --
include preceeding proven therorems in next proof attempt

Figure 6.9: Result of verification using temporal induction.

Figure 6.8 gives the specification of the inductive step of our temporal induction approach. Again we notice that there is no instantiation of the initial specification, and hence the condition speaks about any path in the whole automaton, not only those starting at an initial state. Also notice that the specification contains one transition more than in the base case. Along with the transitions, the specification also contains formulae describing that the path through the automaton should be loop free. Finally, we notice that before the property to be proven, the assumption that the safety condition holds after one transition is included.

This of course differs from the inductive step given in Chapter 4, as here we specify an extra transition. This trick is possible due to the fact that the base case is strengthened on each iteration, and hence if we do find a violation whilst verifying the inductive step, we just ignore it and wait until it is found via the base case. Of course if it is not found via the base case then it is unreachable. Also, having the assumption that the safety condition holds in n previous transitions allows us to disregard unreachable states more quickly as this n increases.

Using Hets and the tool structure it provides, we have once again checked these properties hold using the MiniSat SAT solver. Both the successful base case and induction step verification can be seen in Figure 6.9.

Given that we have now shown both the inclusion check and temporal inductive approaches to be successful and complete verification techniques, we will move on to discuss methods to reduce formula size. Applying such formula reduction methods, will allow us to verify larger ladder programs to a greater number of transitions. Results of these verification techniques will finally be discussed in Chapter 9.

Chapter 7

Program Slicing

Contents

This chapter shall concentrate on a technique known as program slicing [Wei81, Tip95]. We show how the technique of program slicing, popular in other areas of Computer Science including testing [GL91], can be applied to improve the verification processes we have proposed. We begin by introducing the field of program slicing, before giving an algorithm to apply the technique to ladder logic programs. We then give an example of its application to a verification problem, before finally showing the correctness of the slicing algorithm.

7.1 The Concept of Program Slicing

Originally introduced by Mark Weiser [Wei81] in the early 1980's, the concept of program slicing can be defined intuitively as constructing a slice of a program where *"Given an imperative program, a slice is an executable program whose behaviour must be identical to the specified subset of the original program's behaviour"* [Wei81]. Weiser proposed that this is the process taken by a programmer when debugging a program. Given a point of interest in a program, for example where a run-time error occurs, a programmer will search the program code for program statements affecting the point of interest. The programmer will then only consider these effecting statements in the debugging process. The set of statements effecting the point of interest is known as the program slice. Weiser formulated the problem of program slicing, and proposed an algorithm to solve the problem [Wei81].

Since being proposed, program slicing has not only been applied to debugging situations, but also to the areas of program maintenance e.g. [GL91], testing e.g. [HD95] and more importantly program verification e.g. [FH98]. With this wide range of applications, program slicing comes in many forms, here we shall use a technique known as backwards static slicing which consists of given a slicing criterion i.e., a point of interest, computing in a backwards fashion the program slice. Some related approaches and extensions to this approach exist and include:

- Dynamic Slicing [Tip95] - Extend slicing criterion to include a set of inputs and then consider the effect these inputs have on the flow of the program.
- Hybrid Approaches [Tip95] - Combine static and dynamic approaches, e.g, specify only some input values.
- Forward vs Backward Slicing [Tip95] - Direction to slice from. E.g., is the given point of interest a start point or end point. Both forward and backward approaches can be applied to static, dynamic or hybrid approaches.

For further details on program slicing and the terminology used within the field see [Tip95, Wei81].

We now give a small example of how regular program slicing works before continuing to show how we can apply such techniques to aid us in the verification of ladder logic programs.

7.1.1 A Motivating Example

As an example of how program slicing works, consider the code sample in Figure 7.1, which, given a number n, computes the sum and product of the numbers $1 \dots n$, printing the results to the screen.

```
 1  public void Calc(n){
 2     i = 1;
 3     prod = 1;
 4     sum = 0;
 5     while (i<=n){
 6        sum = sum + i;
 7        prod = prod * i;
 8        i++;
 9     }
10     Print(sum);
11     Print(prod);
12  }
```

(a) Original Program.

```
 1  public void Calc(n){
 2     i = 1;
 3
 4     sum = 0;
 5     while (i<=n){
 6        sum = sum + i;
 7
 8        i++;
 9     }
10     Print(sum);
11
12  }
```

(b) Sliced Program.

Figure 7.1: An example of program slicing.

Figure 7.1 also shows the resultant slice that would typically be given by a slicing algorithm when slicing with respect to the criterion $C =$ Point 10. In this simple example it is easy to see that the original program and the computed program slice compute the same values at statement 10 of the program. All statements in the program that affect statement 10 remain in the sliced program, and all statements that have no effect on statement 10 have been removed. Another point that is illustrated by this example is how useful program slicing may actually be when debugging software. Even though the example program is a small and simple program, the sliced program contains 25% less code than the original program.

7.2 Slicing Ladder Logic

The approaches we have proposed for the verification of ladder logic programs quickly give rise to large formulae to be verified. For every iteration of the ladder logic program, both the formulae size and the number of variables increase. As the formula size increases, both the space and time requirements increase. This increase leads to a rather small bound[1] on the number of iterations of a ladder logic program we can verify in a feasible amount of time. We will now discuss, how, like with imperative programs, a slicing technique can dramatically reduce the size of the problem (i.e., the formulae). This in turn will reduce the space and time requirements required to verify ladder logic programs and hence improve the bound for feasibility.

The overall intuition behind slicing in our setting, is that given a particular safety condition for verification, the variables that occur within the safety condition are only dependant on some part of the ladder logic program. Therefore we can remove the parts that do not effect the safety condition. To do this we construct a new formulae to be verified, and hence just verify a subset of the ladder logic program. Here it must be ensured that removing such irrelevant parts of the program does not change the verification result. In the following we will present an algorithm to identify and extract the important parts of a ladder logic program, continuing to state what it means for such a slicing algorithm to be correct, and finally proving its correctness in our automaton theoretic modelling approach.

7.2.1 Algorithm for Slicing Ladder Logic

To begin, we firstly define exactly what it means to construct a slice for a ladder logic program with respect to some safety condition φ. Here we note that we speak on the modelling level, and talk about constructing a slice of our ladder logic program in propositional form.

The first definition we give is the notion of dependence between rungs in a ladder logic program.

Definition 7.1 (Dependant Relation)**:** Given a ladder logic propositional formula $\psi_P = [R_1, R_2, \ldots R_n]$ for some ladder logic program P. We define the relation of $\mathit{dependant}_P \subseteq \{1, \ldots, n\} \times \{1, \ldots, n\}$ between rungs of the ladder logic program, as the transitive closure of

$$\{(i, j) \mid j < i \text{ and } c_j \in \mathit{vars}(\psi_i)\}$$

where a rung i has the form $R_i \equiv c_i \Leftrightarrow \psi_i$.

[1]approximately 100 iterations.

Using the notion of dependence, we can define what it means to be a slice of a ladder logic program.

Definition 7.2 (Slice)**:** Given a ladder logic propositional formula $\psi_P = [R_1, R_2, \dots R_n]$ for some ladder logic program P, and a safety condition φ. A slice $\psi_{p\varphi}$ of ψ_P is an order preserving selection of rungs such that,

- $R_i \in \psi_{P\varphi}$ if $c'_i \in vars(\varphi)$ or $c_i \in vars(\varphi)$, i.e., the coil c'_i or c_i occurs directly in the safety condition, and
- $R_i \in \psi_{P\varphi}$ and $c'_i, c'_j \in dependant_P$ implies that $R_j \in \psi_{P\varphi}$, i.e., if there is a rung already in the slice, and the coil of this rung is dependant on another rung, then this rung is in the slice.

Remark 7.3: This definition does not contain a notion of minimality, that is, a ladder logic propositional formula ψ_P is always a slice of itself.

Given the fact the rung order matters within ladder logic programs, the definition of a slice for a ladder logic propositional formula, shows us that if we wish to compute such a slice we have to do so in a backward fashion. That is, we have to start working from the last rung in a ladder logic propositional formula back to the first. Here we present an algorithm to compute such a slice of a ladder logic propositional formula. The algorithm we present is as presented in [FH98]. We again note that the algorithm is implemented over our propositional formula modelling of ladder logic programs and not over the actual ladder logic diagrams themselves. To present the algorithm we will give three main tasks to be completed. For each task we then explain the process which needs to be implemented, before finally arguing why such a process computes a slice as defined above.

- **Step 1 - Extract variables from safety condition.**
  The first step of the algorithm is to compute a set of all the variables that occur in the safety condition. Given a safety condition of the form described in Section 3.4, we have to extract each variable from the safety condition via an implementation of the `vars` function described in Section 3.1. The result of this set is the production of a set of variable U which occur in the given safety condition.

- **Step 2 - Calculate dependant coils.**
  The second step in the algorithm involves calculating all the coils of the ladder logic propositional formula that effect the variables computed in the previous step. That is, effect the safety condition. To do this, we can began at the last rung of the ladder logic propositional formula, call this rung $R_i \equiv c'_i \Leftrightarrow \psi_i$. We then compare the coil c_i of this rung with the set of variables U computed in step one. If the coil $c'_i \in U$, then we add all the variables occurring in ψ_i to the set U. We now continue to repeat this step for each rung, until a fixed point is reached within the variable set. When this process has been completed, the set U will contain all variables that somehow effect the safety condition. Figure 7.2 illustrates the code that could be used to perform this step.

```
$U_{n+1} \leftarrow U$
do
$\overline{U} \leftarrow U$
$U_{n+1} \leftarrow U$
for $i = n$ down to 1 do
        if $c'_i \in U_{i+1}$ then $U_i \leftarrow U_{i+1} \cup vars(\psi_i)$
        else $U_i \leftarrow U_{i+1}$
$U \leftarrow U_i$
until $U_i \subset \overline{U}$
return $\overline{U}$
```

Figure 7.2: Algorithm to compute step two.

- **Step 3 - Extract dependant rungs.**
  Finally, using the set of variables $\overline{U}$ computed in step two, we can remove all rungs that do not effect the safety condition. To do this, construct the set:

  $$index = \{i \in \{1, \dots, n\} | c_i \in \overline{U} \text{ or } c'_i \in \overline{U}.$$

  Now, remove from the original program, all rungs R_i whose indicies do not appear in *index*. The result ψ_{P_φ} is the sliced version of program ψ_P.

Remark 7.4: The set of indicies representing rungs that are to be removed we shall call $\overline{index}$.

To ensure that the steps of this algorithm are correct, we now give an argument that the steps one to three above compute the slice as defined earlier.

7.2.2 The Algorithm computes a slice

To argue that the algorithm we have outlined computes a slice, we have to argue several things. Firstly it is easy to see that the computed formulae ψ_{P_φ} is a ladder logic propositional formula as defined in Section 3.2. This is due to the fact that we have only removed rungs from the original formula ψ_P and have also upheld the order condition in step two of the algorithm. Now, it remains to argue the condition given by our definition of a slice. To argue this, we need that the rungs occurring in ψ_{P_φ} are all closed under the transitive closure of the *dependant* relation. If we consider the fashion that we construct the variable set $\overline{U}$ in step two of the algorithm, we can see that this condition is fulfilled. This is because every time we find a coil which effects the safety condition, we add all variables on which that coil is dependant to the set U_i of dependant variables. Hence when we consider the next rung in the process, we are not only checking to see if the safety condition is dependant on this rung, but also to see if any of the previously found coils are dependant on this rung. This shows that the dependence between coils is upheld in this step. Given that our algorithm meets these criteria, we know that it computes a slice of the original ladder logic propositional formula.

To give an intuitive understanding of how this algorithm works, we now give a brief example of the resultant slice obtained from applying the algorithm to a concrete problem.

7.3 Application of Slicing to a Simple Example

In this section, we show the results of applying the proposed slicing algorithm to our pelican crossing example ladder logic program. Figure 7.3 shows the original ladder logic propositional formula to be sliced.

$$\begin{aligned}
&[crossing' \iff (req \land \neg crossing),\\
&req' \iff (pressed \land \neg req),\\
&tlag' \iff ((\neg crossing') \land (\neg pressed \lor req')),\\
&tlbg' \iff ((\neg crossing') \land (\neg pressed \lor req')),\\
&tlar' \iff crossing',\\
&tlbr' \iff crossing',\\
&plag' \iff crossing',\\
&plbg' \iff crossing',\\
&plar' \iff (\neg crossing'),\\
&plbr' \iff (\neg crossing'),\\
&audio' \iff crossing']
\end{aligned}$$

Figure 7.3: An example ladder logic program.

Now if we consider the safety condition:

$$(tlag' \lor tlar') \land \neg(tlag' \land tlar') \land (tlbg' \lor tlbr') \land \neg(tlbg' \land tlbr'),$$

we can see that it only speaks about certain variables of the ladder logic program. Therefore, we can apply our slicing algorithm to reduce the program in Figure 7.3 with respect to the given safety condition. Step one of the algorithm would compute the list of variables occurring in the safety condition, that is the set:

$$U = \{tlag', tlar', tlbg', tlbr'\}.$$

After this, step two of the algorithm would find all dependant rungs in a backwards fashion. So in our example the first rung to be checked would be

$$audio' \iff crossing'.$$

Since the coil of this rung does not occur in U, the algorithm moves on to the next rung. This process continues until the rung

$$tlbr' \iff crossing'$$

is reached. Now we see that the coil $tlbr'$ occurs in U, so we add all variables occurring in this rung to the set U_i. This then gives the new set

$$U_i = \{tlag'.tlar', tlbg', tlbr', crossing'\}.$$

Continuing this process until all rungs of the program have been checked will then lead to the resultant set of step two of our algorithm being

$$\overline{U} = \{tlag'.tlar', tlbg', tlbr', crossing', req', pressed\}.$$

Finally, step three of the algorithm will construct a new ladder logic propositional formula containing those rungs whose coils correspond to the variables in the set $\overline{U}$ produced by step two of the algorithm. This removal is done in a backwards fashion, and the resultant ladder logic propositional formula returned by step three of the algorithm for our pelican crossing example is shown in Figure 7.4.

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

Figure 7.4: A Sliced version of Figure 7.3.

In Figure 7.4, we can see that there is a considerably smaller number of rungs than in the original ladder logic propositional formula. Even though our pelican crossing is a small example of a ladder logic program, the slicing algorithm has reduced the number of rungs from eleven down to just six, a decrease of nearly 50%. Such an decrease in size will mean that the formulae we require for our approaches to verification will be reduced considerably. Hence, the problem of hitting space and time issues has been delayed until a much higher number of iterations.

Given that we have shown the success of our slicing approach, we now consider the effect slicing has on our automaton theoretic modelling of ladder logic programs.

7.4 Correctness of Slicing Ladder Logic

In the section, we argue that the operation of constructing a slice through extraction of dependant rungs is correct. That is, we argue that the validity of a ladder logic program is upheld in the sliced version, with respect to the given safety condition. We state what this means in our automaton modelling, before giving a correctness proof of the statement based on such a modelling. To allow us to do this we firstly introduce some notions we will require in our proof.

We firstly introduce what it means to reduce a ladder logic program a series of variables, namely coils. This is the operation that is applied in step three of our algorithm. Here we denote the set of indicies of rungs remaining in the sliced program as *index*, and the indicies of rungs removed from the slice as $\overline{index}$.

Definition 7.5 (Reducing a valuation – $\mu|_{index}$)**:** For a variable set V, given a valuation $\mu : V \to \{0,1\}$, and a set of variables $index$, we define $\mu|_{index} : \{c_i | c_i \in index\} \to \{0,1\}$ as,

$$\mu|_{index}(x) = \mu(x)$$

for all $x \in index$.

For use within our proof, we shall also introduce the following definition of extending a valuation by a set of variables.

Definition 7.6 (Extending a valuation – $\nu_{index} :: f$)**:** For a set of variables $index$, and a valuation $\nu : \{c_i | i \in index\} \to \{0,1\}$, we define $\nu_{index} :: f : \{c_i | i \in \{1,\dots,n\}\} \to \{0,1\}$ where $f : \{c_i | i \in \overline{index}\} \to \{0,1\}$ as,

$$(\nu_{index} :: f)(x) = \begin{cases} \nu(x) & \text{if } x \in \{c_i | i \in index\} \\ f(x) & \text{otherwise} \end{cases}$$

for all $x \in \{1,\dots,n\}$. Often, we use $\nu :: f$ to represent $\nu_{index} :: f$.

Remark 7.7: We shall also overload the $|_{index}$ and $:: f$ notation to paired valuations. That is, $\mu \fatsemi \mu'|_{index}$ will represent the paired valuation $\mu \fatsemi \mu'$ both reduced by the rungs in $index$. Also $\nu :: f \fatsemi \nu' :: f'$ will represent the paired valuation $\nu \fatsemi \nu'$ extended by the functions f, f'.

Now we have given a definition of what is means to remove a set of rungs from our ladder logic program, we will now show that the removal of such a set of rungs using the defined technique, does not effect validity in our automaton representation.

Firstly we introduce a lemma stating that the reduction of paired valuations preserves validity with regards to a given formula.

Lemma 7.8: $\mu \fatsemi \mu' \models \psi_P \Rightarrow \mu \fatsemi \mu'|_{index} \models \psi_{P\varphi}$

Proof. Let R_i be a rung of $\psi_{P\varphi}$. As $\mu \fatsemi \mu' \models \psi_P$, we know $\mu \fatsemi \mu' \models R_i$. As R_i does not depend on $c_k, c'_k, k \in \overline{index}$, we have that $\mu \fatsemi \mu'|_{index} \models R_i$. $\square$

Using this lemma, we shall now show that the reduction of paired valuations by some rung set $index$, does not effect the reachability of states in our automaton model.

Lemma 7.9: For all μ, μ' in an automaton $A(\psi_P)$ for some ladder logic program P and some safety condition φ, if $\mu \fatsemi \mu'$ is reachable with respect to $A(\psi_P)$ then $\mu \fatsemi \mu'|_{index}$ is reachable with respect to $A(\psi_{P\varphi})$.

Proof. By induction on the length l of a derivation for $\mu \fatsemi \mu'$, i.e.,

$$\mu_0 \to \mu_1 \to \dots \to \mu_{l-1} \to \mu \to \mu'$$

where $\mu_0 \in I_s$.

Base Case: $l = 0$. Let $\mu \to \mu'$ be a derivation of length 0, then we know that:

1. $\mu \in I_s$ of $A(\psi_P)$, that is $\exists \nu$ s.t. $\nu \models \neg I \wedge \nu \,\text{\textfrc{;}}\, \mu \models \psi_P$ and
2. $\mu \,;\, \mu' \models \psi_P$.

Due to the fact we have not removed variables from I, we clearly have that $\nu|_{index}(i) = \nu(i)$ for all $i \in I$. Thus $\nu|_{index} \models \neg I$. By Lemma 7.8 and given that $\nu \,;\, \mu \models \psi_P$ we have that $\nu \,;\, \mu|_{index} \models \psi_{P\varphi}$ and hence that $\mu|_{index} \in I_s$ of $A(\psi_{P\varphi})$. Given that $\mu|_{index}$ is an initial state in $A(\psi_{P\varphi})$ it remains to show that $\mu \,;\, \mu'|_{index} \models \psi_{P\varphi}$. This follows directly from Lemma 7.8 and thus $\mu|_{index} \to \mu'|_{index}$ is a derivation in $A(\psi_{P\varphi})$.

Induction Step: $l+1$. Let

$$\mu_0 \to \mu_1 \to \cdots \to \mu_{l-1} \to \mu_l \to \mu \to \mu'$$

be a derivation of length $l+1$ for $\mu \,;\, \mu'$ in $A(\psi_P)$. Here we know that $\mu_l \,;\, \mu$ are reachable via a derivation of length l in $A(\psi_P)$. Hence by induction we know that $\mu_l \,;\, \mu|_{index}$ are reachable in a derivation of length l in $A(\psi_{P\varphi})$. We know that $\mu \,;\, \mu' \models \psi_P$ and hence by Lemma 7.8 we know that $\mu \,;\, \mu'|_{index} \models \psi_{P\varphi}$. Therefore

$$\mu_0|_{index} \to \mu_1|_{index} \to \cdots \to \mu_{l-1}|_{index} \to \mu_l|_{index} \to \mu|_{index} \to \mu'|_{index}$$

is a derivation in $A(\psi_{P\varphi})$.

□

Next, we introduce a lemma which will aid us to show the reverse direction. That is that including the removed set of rungs again does not change reachability in our automaton.

Lemma 7.10: $\nu \,;\, \nu' \models \psi_{P\varphi}$ implies that for all f there exists a f' such that $\nu :: f \,;\, \mu' :: f' \models \psi_P$

Proof. Taking $\nu \,;\, \nu'$, we can choose an arbitrary f to give $\nu :: f$. Now we can define for each rung $R_i \in \psi_p$ and $i \in \overline{index}$:

$$f'(x) = \begin{cases} 0 & \text{if } x = c_i \text{ and } \nu :: f \not\models \psi_i \\ 1 & \text{if } x = c_i \text{ and } \nu :: f \models \psi_i \end{cases}$$

Hence, we gain $\nu :: f$ and $\nu' :: f'$ such that $\nu :: f \,;\, \nu' :: f' \models \psi_P$. □

Using this lemma, we now proof that expanding an automaton by adding extra variables does not effect reachability in the automaton.

Lemma 7.11: For all ν, ν' in an automaton $A(\psi_{P\varphi})$ for some ladder logic program P, if $\nu \,;\, \nu'$ is reachable with respect to $A(\psi_{P\varphi})$ then there exists f, f' such that $\nu :: f \,;\, \mu' :: f'$ is reachable with respect to $A(\psi_P)$.

Proof. By induction on the length l of a derivation for $\nu \,;\, \nu'$, i.e.,

$$\nu_0 \to \nu_1 \to \cdots \to \nu_{l-1} \to \nu \to \nu'$$

where $\nu_0 \in I_s$.

Base Case: $l = 0$. Let $\nu \to \nu'$ be a derivation of length 0, then we know that:

1. $\nu \in I_s$ of $A(\psi_{P\varphi})$, that is $\exists \mu$ s.t. $\nu \models \neg I \wedge \mu \mathbin{⨾} \nu \models \psi_{P\varphi}$ and
2. $\nu \mathbin{⨾} \nu' \models \psi_{P\varphi}$.

Therefore, we need to show there exists a $\mu :: f$ such that $\mu :: f \models \neg I$. Since we have not added any input variables, any arbitrary f suffices. As $\nu \mathbin{⨾} \nu' \models \psi_{P\varphi}$, we know by Lemma 7.10 that there exists f' such that $\nu :: f \mathbin{⨾} \nu' :: f' \models \psi_P$.

Induction Step: $l + 1$. Let

$$\nu_0 \to \nu_1 \to \cdots \to \nu_{l-1} \to \nu_l \to \nu \to \nu'$$

be a derivation of length $l + 1$ for $\nu \mathbin{⨾} \nu'$ in $A(\psi_{p\varphi})$. Here we know that $\nu_l \mathbin{⨾} \nu$ are reachable via a derivation of length l in $A(\psi_{P\varphi})$. Hence by induction, we know that $\nu_l :: f_1 \mathbin{⨾} \nu :: f$ are reachable in a derivation of length l in $A(\psi_P)$, for some f_1 and f. We know that $\nu \mathbin{⨾} \nu' \models \psi_P|_k$ and hence by Lemma 7.10 we know that $\nu :: f \mathbin{⨾} \mu :: f' \models \psi_p$ for some f'. Therefore,

$$\nu_0 :: f_0 \to \nu_1 :: f_1 \to \cdots \to \nu_{l-1} ::_{l-1} \to \nu_l :: f_l \to \mu :: f \to \mu' :: f'$$

is a derivation in $A(\psi_P)$. □

Given that we have now proven the removal of rungs still ensures validity is upheld within our automaton theoretic modelling, we shall now show that the overall operation of slicing is also correct.

Theorem 7.12: Given a ladder logic propositional formula ψ_P for some ladder logic program P, its corresponding automaton $A(\psi_P)$, a safety condition φ,

$$A(\psi_P) \models \varphi \Rightarrow A(\psi_{P\varphi}) \models \varphi.$$

Proof. Let $\nu \mathbin{⨾} \nu'$ be reachable in $A(\psi_{P\varphi})$. Then by Lemma 7.11 there exists f, f' such that $\nu :: f \mathbin{⨾} \nu' :: f'$ is reachable in $A(\psi_P)$. Thus, via our assumption, we have that $\nu :: f \mathbin{⨾} \nu' :: f' \models \varphi$. Hence $\nu \mathbin{⨾} \nu \models \varphi$ as φ does not depend on variables in $\{c_i | i \in \overline{index}\}$. □

Theorem 7.13: Given a ladder logic propositional formula ψ_P for some ladder logic program P, its corresponding automaton $A(\psi_P)$, a safety condition φ,

$$A(\psi_P) \models \varphi \Leftarrow A(\psi_{P\varphi}) \models \varphi.$$

Proof. Let $\mu \mathbin{⨾} \mu'$ be reachable in $A(\psi_P)$. Then by Lemma 7.9, $\mu \mathbin{⨾} \mu|_{index}$, where *index* is the set of variables produced by slicing, is reachable in $A(\psi_{P\varphi})$. Thus via our assumption we know that $\mu \mathbin{⨾} \mu'|_{index} \models \varphi$. Hence $\mu \mathbin{⨾} \mu \models \varphi$, as φ does not depend on the variables in $\{c_i | i \in \overline{index}\}$. □

As we have shown the slicing of ladder logic propositional formulae to be correct with respect to some safety condition, we can now apply this algorithm to reduce the size of ladder logic propositional formulae. We continue by showing how we have implemented this algorithm along with the other verification algorithms given earlier, into a verification tool for Westrace interlockings.

Chapter 8

The Verification Tool

Contents

In this chapter, we describe how we have modified and extended the tool created by Kanso in [Kan08]. The result of which is a new verification tool for Westrace interlocking ladder logic programs. We shall give an overview of how the tool given by Kanso firstly works, and then how it has been improved. We comment on aspects such as how it has been made more generic and enhanced via efficiency improvements. We shall show the overall structure of the new tool, including where the application of algorithms such as the slicing algorithm given in Chapter 7 take place. We comment on some software engineering issues that were encountered along the way, before giving some possible future improvements for the tool.

8.1 The Original Tool

The verification tool created by Kanso [Kan08] consisted of two underlying programs. One program concerned with safety conditions and one program concerned with verification. The general outline of these tools is shown in Figure 8.1 and Figure 8.2.

Figure 8.1 shows the overall software architecture for the program created by Kanso to deal with the generation of safety conditions. We can see that there are two inputs to the main program itself. One is an informal first order form safety condition, given in a language defined in [Kan08]. Whilst the other is a railway plan. The railway plan is constructed out of two further parts, namely, a railway topology describing the layout of the railway via an encoding in Prolog. Then secondly a Java mnemonic wrapper. The namespace of the railway topology used for signals, points and other entities, differs from the namespace of the concrete ladder logic program[1]. For this reason, a specific namespace mnemonic wrapper was created by Kanso using Java. This wrapper is then responsible for converting names used in the railway topology into concrete names used within the ladder logic program. This process takes place when the overall safety condition generation process queries this namespace converter for the names of track entities used within the safety condition. Obviously, this part of the tool is not very generic, as both the Prolog topology and the Java wrapper is dependant on the railway plan being verified.

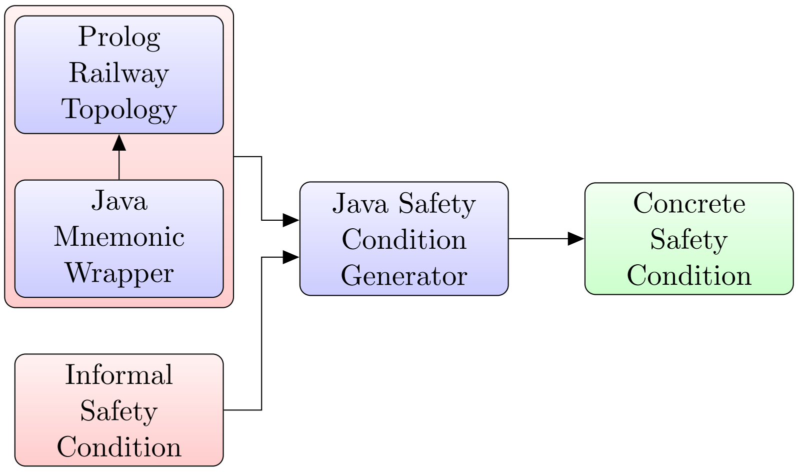

Prolog Railway Topology
Java Mnemonic Wrapper
Informal Safety Condition
Java Safety Condition Generator
Concrete Safety Condition

Figure 8.1: Architecture of the Safety Condition Generator created by Kanso.

The safety condition generation program, given these two inputs, transforms the informal safety condition into a series of propositional formulae to be verified. The propositional formulae would contain concrete names instead of the abstract names that were given in the informal safety condition. For example the word "point" could be replaced by the actual point "TP101". These names are gained, as explained above, from the Prolog encoding and Java wrapper.

The second part of the tool, the structure of which is shown in Figure 8.2, was concerned with the actual verification process. This part of the tool is predominantly programmed using Haskell. As inputs it takes, a ladder logic program and a formal safety condition obtained from the safety condition generator. This program firstly uses a ladder logic parser to parse the ladder logic program into a propositional formula representation. This was completed via the use of an abstract syntax for propositional logic in Haskell. Then, using this propositional formula, and the safety condition, the program constructs a pair of inductive formulae to be verified (see Chapter 4). These formulae are passed, by the program to a SAT solver to be verified. Depending on the result from the SAT solver, the program would then either return that the system was safe, or return a counter example showing the system to be unsafe. To return such a counter example, the counter example from the SAT solver would be run through a series of scripts. These scripts would then create a pictorial form of the counter example for the engineers at Invensys to study.

[1]This is a design decision made by Invensys.

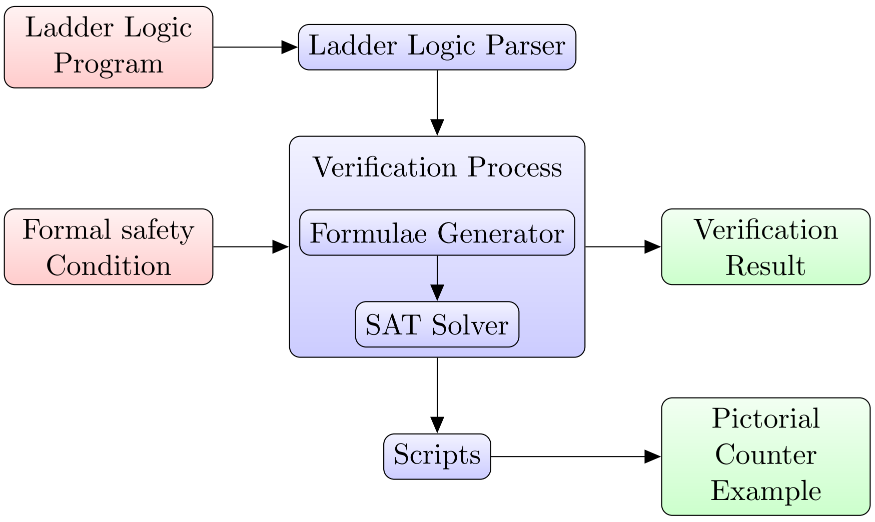


Figure 8.2: Architecture of the verification tool by Kanso.

Finally combining these two programs, results in the overall tool developed by Kanso. We shall now consider some problems with these two programs before continuing to show the structure of our new tool.

8.1.1 Problems with the Implementation

With regards to the the two programs commented on above, there are a few main problems. The largest problem is that of genericity. If we consider the two programs, then both contain parts that are tightly coupled with the particular railway that was being verified. In the safety condition tool, the Prolog topological encoding of the railway and the Java mnemonic wrapper are both specific to the interlocking verified. In the verification program, the ladder logic parser was also based around the single ladder logic program under consideration. Finally, with regards to genericity, the counter example generation uses scripts and files that are based around a template picture of the concrete railway under verification. Therefore this template would have to be reconstructed for every train station that is verified. Hence, a more generic tool would require these parts of the programs to be improved.

Other problems with the two programs that are not so fundamental from a theoretical point of view, but more from a software engineering point of view, were those regarding the actual implementation details. As discuss above, several different implementation languages were used throughout the tool. This makes understanding and maintaining all aspects of the tool somewhat difficult, as obviously it requires expertise in all the languages used. Also, the technical documentation for the tool was lacking, with much of the Haskell code missing proper documentation. This again adds to the difficulty of maintaining and improving the tool. These software issues are probably due to the fact that the implementation by Kanso was a prototypical one. That is, the tool was created to answer the theoretical question of whether or not verification of Westrace interlockings was feasible. As we plan to create a more concrete solution from this prototype, we shall try to improve on such software engineering aspects of the tool. These improvements to not only the software engineering, but also to the theoretical framework of the tool will now be given discuss.

8.2 Development of the New Tool

The tool we have implemented has been based on aspects of the tool by Kanso. The overall aim of improving the tool by Kanso can be split into some smaller aims, namely:

- Test the genericity of the tool by Kanso, by trying to use it with another train station.
- Improve the verification processes by implementing the algorithms for model checking.
- Improve the efficiency of verification by adding functionality to perform slicing.
- Finally, improve the software engineering aspects of the tool, for example, code quality and maintainability.

We no give a general overview of the structure of our new tool, before commenting on the steps that were taken, and problems that arose when trying to reach the above goals.

Firstly, with regards to structure of the new tool, we note that we also split the tool into two smaller programs, one program concerning generation of safety conditions, and one program concerning the verification process.

The overall structure of the safety condition generator has remained the same, i.e., as given in Figure 8.1, but it has been extended to cope with another railway station. That is, we have given another Prolog encoding of the new railway, along with a Java mnemonic wrapper for this new railway. We have then used the existing program, and plugged these two pieces into it. More information on this implementation process is given in Section 8.3. The reasons for not creating a more generic tool at this point is down to the fundamental design decisions taken by Invensys. Currently, mnemonics used by Invensys differ for every railway station and corresponding interlocking. Often these mnemonic mappings are highly complicated and hence unless further work with Invensys is completed to try to generalise such mnemonics, it will be difficult to this given program generic.

With regards to the overall architecture of the verification part of the tool, many changes have been made. Figure 8.3 outlines the structure of this new program.

In Figure 8.3 we can see that the overall verification process still takes as inputs a ladder logic program and a formal safety condition, but also now has an input representing the bound to compute to with some verification processes. We can also see that the ladder logic program is still parsed using a parser, although this parser has been extended. These extensions we will see in more detail in Section 8.3. After the ladder logic has been parsed into an abstract syntax for propositional formulae, we can see that is is entered along with the safety condition into the main verification process.

This verification process is where most of the changes to the original tool have occurred. The first change that is noticeable, is that the ladder logic propositional formula is now run through a slicing algorithm, as described in Chapter 7. This algorithm slices, with respect to a given safety condition, the ladder logic propositional formula and produces a new, usually smaller formula to be used within the verification process. Next, instead of only performing inductive verification, the user now has an option of choosing between three verification processes. That is, the user can select to perform inductive verification, bounded model checking or temporal induction. These three approaches are implemented using the algorithms we have described in earlier chapters. We note that the inductive verification approach has also been implemented more efficiently, the result of which we shall see in Chapter 9. Also, we notice that here we make use of the newly added third input, this defines how many iterations the user would like to run the bounded model checker over. We should also note that partly implemented, is the infrastructure to allow the user to use our backwards iteration algorithm. Currently, a counter example can be hard coded into the tool, but to fully implement such an approach, a parser for counter example files would need to be created. Thus, in turn allowing a trace to be constructed backwards from the counter example.

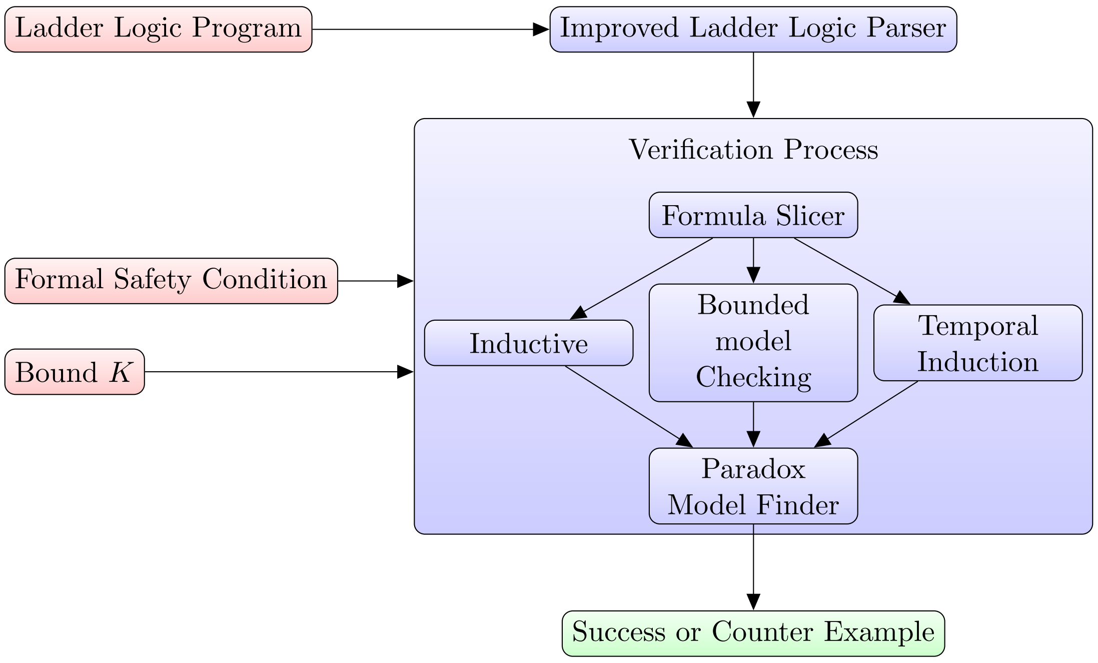


Figure 8.3: Architecture of our new verification tool.

Finally, we notice that the scripts and pictorial counter example have been replace with a text based counter example trace[2]. This counter example trace allows engineers at Invensys to see how the interlocking system reacts through each execution of the ladder logic. Also, such a counter example trace, when compared to a pictorial format trace, is completely generic and will work for any given railway interlocking. We also comment here that the removal of these scripts has the effect that the tool should work correctly on any operating system, unlike the tool given by Kanso which was tied to a Linux based operating system.

[2]Of course the trace output with inductive verification is simple a counter example.

We now continue by commenting on the processes that were undertaken to make the changes described above.

8.3 Encoding A New Train Station

The first main topic of implementation we shall discuss is the changes that were made to successfully allow the tool to work for a new train station. To do this two main tasks were completed. One was to encode the new train station topology into a Prolog file and to give a Java mnemonic wrapper for this topology. The second was to extend the ladder logic parser with some extra features to allow it to parse the new ladder logic file.

To encode a new train station, involved creating a Prolog file that represented the topological structure of the railway station. To do this, we followed the same Prolog term naming conventions as Kanso [Kan08]. That is, we used terms such as "tracksegment" and "mainsignal" to encode the main entities occurring in the new train station. An example of part of this encoding is given in Figure 8.4. Here we note that real track segment names have been converted into artificial ones for legality reasons.

```
        .
        .
% define track segments
tracksegment('T192').
tracksegment('T513').
tracksegment('T199').
tracksegment('T213').
tracksegment('T411').
        .
        .
```

Figure 8.4: Example of encoding track segments into Prolog.

This sample encoding illustrates how track segments of the new railway have been encoded. During this encoding process, we note that there was no need to introduce any new term constructs over the constructs defined by Kanso. Hence the encoding of the new train station into a Prolog file was trouble free.

The next part of the tool that required creating for the new train station, was the java mnemonic wrapper. Unlike the process of encoding the train station into a Prolog file, this process was more complicated. This was down to the fact the mnemonic mappings are not standardised by Invensys. Thus there are different mappings for different train stations. This meant that the mnemonic mappings created by Kanso were of no use for this train station, and hence a whole new mnemonic mapping structure was given. Here, again for legality reasons, we have emitted an example of how such a mnemonic mapping is constructed. Although a simple example is that every track segment name, e.g., like "T411" from Figure 8.4, would be extended with a suffix, such as ".TSR". Thus the overall variable used within the ladder logic program to represent the example track segment would be “T411.TSR”. This process of appending such suffixes to variable names is the job of the mnemonic wrapper we have created.

The last part of the tool which required extension was the ladder logic parser. As mentioned previously, this parser was centered around the ladder logic program for a single train stations, hence when we tried to parse a new ladder logic program for the new train station, the process failed. When this was looked into further, it was found that the new train station simply made us of some new “key words”, and hence it was a relatively simple task to extend the ladder logic parser to recognise these key words.

By making these two changes, we have managed to successfully run the verification tool for a new train station. This shows that the approaches taken by Kanso do in fact scale up to allow the verification of further Westrace interlockings.

8.4 Implementing New Verification Strategies and Slicing

The most substantial improvement to the tool itself was the inclusion of new verification strategies and the functionality to perform slicing. To complete this task involved the implementation of the algorithms discussed in Chapters 5, 6 and 7. That is, we implemented the following verification processes:

- an improved inductive approach to verification,
- a forward iteration approach to verification,
- a backward iteration approach to verification, and
- a temporal induction approach to verification,

along with an implementation of the slicing algorithm given previously. As we have already discuss the algorithmic details of these approaches, here we shall not concentrate on the algorithms, but instead how these verification strategies have lead to the inclusion of a new underlying proof tool begin used in the tool, and how the tool has been modified to interface with such a proof tool.

8.4.1 Changing the Proof Tool

The proof tool originally used by Kanso in his verification tool was the OKsolver [Kul08]. This is a pure SAT solver which takes as input so called Dimacs format files, and also produces outputs in Dimacs format. The tool we have used, namely Paradox [par09], differs from this as it is not a SAT solver but a model finder which makes use of the MiniSat SAT solver [min09]. For this reason, the Paradox tool does not take as input Dimacs format files, but instead, files in the format of the TPTP [tpt09] specification language. The results returned by Paradox are also are then given in the corresponding results language of TSTP [tpt09]. This change meant that the tool had to be modified to create TPTP specifications instead of Dimacs format files. Figure 8.5 shows one of the Haskell functions used to output such TPTP specifications.

The Haskel function shown in Figure 8.5, is the function that is used to output a TPTP specification of the initial conditions for a ladder logic program. We can see that the first thing the function does is to open a file called "Initial.tptp". Next the function defines a in line function *putVarv* to write a single axiom to the specification file. This function basically writes a axiom representing that the given variable *v* should be false. Using this function, we hen see a *sequence* function which runs the defined *putVar* function for every input variable in the given ladder. The effect of this is that we get a file which contains axioms stating that every input variable is set to false.

```
printInitialTp :: [Formula] -> IO()
printInitialTp ladder = do
  h <- getAndOpenFile "Initial.tptp" WriteMode
  let putVar v = do
        hPutStrLn h ("fof(ax,axiom, ~" ++ v ++ ") .\n")
        return()
  sequence_ (map putVar (getVarsPara 0 ladder))
  hClose h
  return()
```

Figure 8.5: Haskell code for printing TPTP specification of initial conditions.

In a similar manner to the function in Figure 8.5, functions have been implemented to create all the specifications for the verification approaches we require. Also, the result returned from Paradox is in the format of a TSTP results. This results does not contain complicated features and basically shows exactly the value assigned to each variable. For this reason now processing of the results returned from Paradox was needed.

Finally, by combining our verification algorithms with such functions to print TPTP specifications, we have enabled our tool to interface with the Paradox tool completely automatically. The results of the verification using Paradox as the underlying tool proved to be highly successful, as we shall see in the next chapter.

8.5 Software Engineering Practices

The last item of the new tool we shall comment on, is regarding the software engineering aspects to the tool. We have revisited all the code base of Kanso's, improving it by adding full technical documentation. Of course, with this, full documentation of all new code has also been given. That is for all Haskell code used within the tool, there is now complete Haddock documentation, and for the Java sections of the tool, there is the corresponding complete Java doc. A simple example of such Haddock documentation is given in Figure 8.6. This comment snippet illustrates the type of Haddock documentation that has been added to all files. In this particular example we see that the comment speaks about the whole Haskell code file, which happened to be used to represent ladder logic programs. Using these comments web pages have been created containing documentation for every function in the code base.

Along with such Haddock comments, some technical comments have also been added to the code in places. Such comments will allow future maintainers of the code to understand quickly what complicated operations are being performed.

```
--------------------------------------------------------------
--
-- |
-- Module:       Ladder
-- Maintainer:   cspj@swan.ac.uk
-- Stability:    provisional
-- Portability:  portable
--
-- Defines the data type used to represent the ladder, it is
-- simple but has lots of operations to help.
-- All ladders start with a root and a list of rungs. each rung
-- has an identifier and a list of cells. the cells are used
-- to represent the components of the ladder logic diagram.
-- A module for displaying tree like structures.
--
--------------------------------------------------------------
```

Figure 8.6: Example illustrating the inclusion of Haddock documentation in Haskell files.

The maintainability of the tool as also been made easier by the reduction in the number of languages used to create the tool. As mentioned, we have managed to simplify the overall implementation of the tool by removing the use of any scripting languages. This in turn also has the advantage that the tool should work correctly on any operating system, not being tied to a Linux based operating system like the implementation by Kanso.

8.5.1 Future Improvements

We now give some issues, mainly regarding the user interface of the tool we have created. That is, all the algorithms used for verification in the tool are sound, but the tool could maybe be adapted to suit the needs of Invensys further.

The first example of this is that currently the tool runs with a simple command line interface. What would be more suited to Invensys, would be to have some form of graphical user interface displaying various options to the user. This idea could be extended further, and the verification tool could be built into the tools that Invensys already use in the design process of Westrace Interlockings. Then as the Interlocking ladder logic programs are designed, they could continually be verified. Also, as mentioned above, the user cannot currently run the backwards iteration algorithm without changing the source code. Obviously this is not feasible for an average user, and hence the extension of the tool with a counter example parser is highly recommended.

Finally the last improvement that would be recommended for the tool, it to solve the current namespace problem with the safety condition generator. Again, as mentioned above the mnemonic mappings used by Invensys are non-trivial, and differ for every railway interlocking under verification. Therefore we propose that a graphical user interface is created, which gives the option for the user to enter mnemonic mappings for a given station. These mappings could then be stored for the given station, and used whenever a verification request is issued for that station. To do this successfully, features would have to be introduced to give the user a standard way to enter mnemonic names, and hence further discussions with Invensys would be required to determine the requirements for such a feature. If this feature was implemented successfully, then the tool would be as generic as possible, with the only aspect requiring editing to allow a net train station to be verified, would be the topological track plan encoded in Prolog.

In this chapter, we have covered the overall architecture of the tool we have created. In the next chapter, we shall show the results of applying our tool to the verification of two concrete real world interlockings.

Chapter 9

Verification Results

Contents

In this chapter, we give some results that were obtained through the application of our tool to some verification problems proposed by Invensys. We note that some of these safety conditions have been verified previously in [Kan08] and we simply apply different verification techniques to these problems, mainly to explore the feasibility of each technique. For legal reasons, we shall not give details on the railway stations that were verified, or give formulae for the safety conditions under verification. Instead, we shall give abstract names to the railway stations and give safety conditions in an informal but descriptive way. Finally, when a counter example has been obtained, again for legal reasons, the counter example will be omitted. We will conclude this chapter by giving a discussion of the results obtained from each verification technique, before comparing the result's from a practical and then theoretical viewpoint.

9.1 The Testing Platform

Before giving the concrete verification results that have been achieved, we shall firstly comment on the hardware and software used for the experiment environment. All the verification experiments have been completed using the same computer with the following specifications:

- Operating System – Ubuntu 9.04, 64-bit edition.
- CPU – Intel Q9650, Quad core - 3GHz.
- System Memory – 8GB DDR2 ram.

Also, in all tests, we have used the Paradox model finder[1] as the underlying proof tool.

9.2 Experimental Results – Interlocking A

The first railway station interlocking we have verified is the same interlocking that was verified by Kanso in [Kan08]. This interlocking contains a total of 331 rungs and 599 variables and in terms of physical size, the interlocking controls a small train stations. For this interlocking, we verify four safety conditions. These four safety conditions have been previously verified by Kanso, and here we shall use the conditions to both test the results of each verification method we have proposed, whilst also showing the generation of counter example traces to be possible. Below we give each safety condition, along with the verification results for each approach, namely:

1. Inductive verification result by Kanso in [Kan08].
2. Inductive verification result via our new tool.
3. Bounded model checking (BMC) result using our forwards iteration algorithm both with and without inclusion check.
4. Temporal induction verification result.

Obviously, if a condition is proven safe using inductive verification, then it will also be safe for all possible bounds k when using a bounded model checking approach, and similarly for a temporal induction approach. For these reasons, when a safety condition is proven to hold via induction, we shall simply show the performance of other verification algorithms up to the largest feasible bound. Also, when a counterexample is found, by say the bounded model checking approach, we will not repeat details as to whether it was found by the other approaches too, as this is always the case.

9.2.1 Point can not Move Unless Free

The first safety condition that we verified was that concerning the movement of points. The safety condition basically says that a if a point has moved position when compared to its position in the previous execution of the ladder logic, then it must have been allowed to do so in the previous execution of the ladder logic. I.e., "$P.normal \wedge P'.reverse \implies P.free$".

Verification Type	**Result**	**Time Taken(s)**
Inductive Base	Succeed	0.38
Inductive Step	Fail	0.37
Forward Iteration (BMC)	Trace length = 3	0.81
Forward Iteration (BMC) with slicing	Trace length = 3	0.58

Figure 9.1: Our verification results for safety condition 1.

[1]Based on MiniSat.

This safety condition was verified by Kanso with the result that the verification of the base case succeeded whilst the verification of the inductive step failed. The overall verification time to find this failure was 182 seconds. With our verification approaches we gained the following results:

In Figure 9.1 we can see that, like with kanso's inductive verification, the base case for this condition holds where as the inductive step is violated. We also see that the efficiency of the inductive verification approach has been increased dramatically. The violation of this safety condition illustrates the first successful application of our forward iteration algorithm. That is, in the sense that we have successfully constructed a counter example trace to where the safety condition is violated. We can also see that the application of the slicing algorithm, with regards to the forward iterations algorithm, has also reduced the time taken to produce such a counter example trace.

9.2.2 Points not in Normal and Reverse

The second safety condition we have verified for train station A again concerns movement of points. It says that a point can not be in both normal and reverse at the same time. I.e., "$\neg(P.normal \wedge P.reverse)$" Obviously if such a condition did occur within the ladder logic program, them the interlocking would not be providing accurate information about the physical track layout.

Verification Type	**Result**	**Time Taken(s)**
Inductive Base	Succeed	0.42
Inductive Step	Succeed	0.29
Forward Iteration (BMC)	Safe – Bound = 10	3.38
	Safe – Bound = 20	7.23
	Safe – Bound = 50	20.16
	Safe – Bound = 100	47.49
	Safe – Bound = 200	120.59
	Safe – Bound = 500	522.86
	Safe – Bound = 1000	1705.65
	Unknown – Bound = 2000	Out of Memory
Froward Iteration (BMC) with slicing	Safe – Bound = 10	0.76
	Safe – Bound = 20	1.38
	Safe – Bound = 50	3.55
	Safe – Bound = 100	7.59
	Safe – Bound = 200	17.05
	Safe – Bound = 500	58.65
	Safe – Bound = 1000	366.91
	Safe – Bound = 2000	4553.82
	Unknown – Bound = 3000	Out of Memory

Figure 9.2: Our bounded verification results for safety condition 2.

This safety condition was verified by Kanso as being safe, with the overall process taking a total of 1.7 seconds. Below, we give various results of our tool, beginning with the results for bounded verification techniques, then sowing the results for unbounded techniques.

Figure 9.2 shows the verification results for the improved inductive verification, along with a series of forward iteration results with and without slicing. Again we can clearly see an increase in the speed of inductive verification when compared to the result from Kanso. Also, this table illustrates how the complexity of the problem grows as we increase the number of iterations with the forwards reachability approach. We can see that without applying a slicing technique, verification is only possible up to 1000 iterations. An interesting point here is that the verification time for such a large amount of iterations is still feasible with regards to time, just not with regards to space. This is due to the large, but not necessarily complicated to solve, formulae we obtain during the verification process. It is also why, as we can see from the table of results, by applying our slicing algorithm to the verification process, we are able to verify up to 2000 iterations in total.

Verification Type	**Result**	**Time Taken(s)\ Inclusion Reached**
Forward Iteration (BMC) with inclusion	Safe – Bound = 10	13.48\No
	Safe – Bound = 20	56.09\No
	Safe – Bound = 50	652.45\No
	Unknown – Bound = 100	Out of Memory\No
Forward Iteration (BMC) with inclusion and slicing	Safe – Bound = 10	1.7\No
	Safe – Bound = 20	5.14\No
	Safe – Bound = 50	31.68\No
	Safe – Bound = 100	177.59\No
	Safe – Bound = 200	1554.85\No
	Unknown – Bound = 500	Out of Memory\No
Temporal Induction with slicing	Safe – Bound = 10	2.60\Yes

Figure 9.3: Our bounded verification results for safety condition 2.

Finally, Figure 9.3 shows the results of applying our unbounded verification techniques to the problem. As we can see, the first unbounded verification technique, namely using an inclusion formula, fails to prove that inclusion is reached. We can also see the effect of adding the inclusion check to the verification, as with the bounded techniques, 2000 iterations were possible where as here only 200 iterations were possible. This again is due to the rapid growth in formula size when the inclusion check is added. Although these results show that out temporal induction approach to verification succeeds to reach inclusion. This fact follows from the fact that we can prove the safety property inductively, and hence it is going t be provable via our temporal induction approach. Overall, in this result, we can see that it may be unfeasible to reach inclusion using the current forward iteration techniques.

9.2.3 Points Locked when Route Set

The next safety condition we have verified, is concerned with route setting. It says that if a given route is set, then the corresponding points must be set to their correct positions. I.e., "$R \implies P1.normal \wedge P2.reverse$". Again, if a route was set and the corresponding points were set to the incorrect positions, then a train derailment would occur.

With regards to verification by Kanso, the verification of this safety condition failed. The overall time taken to find this failure was a total of 1143 seconds. Below we once again summarise our verification attempts.

Verification Type	**Result**	**Time Taken(s)**
Inductive Base	Succeed	0.42
Inductive	Fail	0.34
Forward Iteration (BMC)	Trace length = 4	1.76
Forward Iteration (BMC) with slicing	Trace length = 3	0.40

Figure 9.4: Our verification results for safety condition 3.

As with the verification of safety condition one, Figure 9.4 shows that our inductive verification approach gave the same result as Kanso, only much quicker. We can also see that the forwards iteration approach was also able to give a counter example trace with respect to the failure of this safety condition too. Here though it is interesting to see that the slicing algorithm not only has the effect of increasing the efficiency of the forward iteration approach when finding a counter example trace, but also had the effect of shortening the number of iterations required to reach the counter example. This in turn means that the counter example trace will be shorter, and hence easier for the engineers at Invensys to understand.

9.2.4 Point can not Move when Track Occupied

The final safety condition we shall verify for train station A, is one that describes that a point should not be allowed to move if there is a train occupying the track segment where the point occurs. This safety condition is obviously something that should be upheld by the interlocking program.

Interestingly here though, the verification by Kanso failed within the base case verification, taking 297 seconds. Hence when we applied our inductive verification approach to the problem, we also obtained a failure within the base case verification, but this time only taking 0.51 seconds. As the failure occurs in the base case verification, a counter example return by inductive verification, also happens to be a trace to the problem. For this reason there is no point in commenting our further techniques, as they all fail in the first iteration, and hence the counter example obtained from inductive verification suffices.

9.3 Experimental Results – Interlocking B

The second railway station interlocking which we have applied our tool to is a completely new interlocking that has not previously been verified. This interlocking contains a series of 238 rungs and 361 variables and in terms of physical size, the interlocking controls another slightly more complex train station than train station A. Also, according to the engineers at Invensys the overall structure of the ladder logic program for interlocking B is more complicated than that of interlocking A, even though the number of rungs and variables are less. Hence it is interesting to see how the approach fairs on a more complicated program. Below, we give the verification results of applying the tool to verify three safety conditions. For each result we shall give an overview of the approaches we have applied, again namely:

1. Inductive verification result via our new tool.
2. Bounded model checking (BMC) result using our forwards iteration algorithm both with and without inclusion check.
3. Temporal induction verification result.

Obviously, once again if a condition is proven safe using inductive verification, then it will also be safe for all possible bounds k when using a bounded model checking approach, and similarly for a temporal induction approach. For these reasons, when a safety condition is proven to hold via induction, we will once again, simply show the results of our iterative approach to the largest bound possible. Similarly to the results for train station A, we will avoid the repetition of details when a counter example trace is found by a given approach.

9.3.1 Point can not Move unless Free

The first safety condition which we verify for the new train station, is the same as the first condition verified for train station A. That is, it describes that a point can only have moved if it was free to move in the previous execution of the ladder logic. The table below give the verification result for this property.

Verification Type	**Result**	**Time Taken(s)**
Inductive Base	Succeed	0.24
Inductive	Fail	0.18
Forward Iteration (BMC)	Trace length = 2	0.64
Forward Iteration (BMC) with slicing	Trace length = 2	0.39

Figure 9.5: Our verification results for safety condition 1.

In Figure 9.5, we can see that overall the inductive verification fails in the inductive step. Then, as would be expected, the forwards iteration approach gives a counter example trace. Interestingly this counter example trace occurs very quickly within only 2 iterations of the algorithm. Also, once again we can see that slicing makes an effect on the overall speed of finding such a counter example.

9.3.2 Point not in Normal and Reverse

Again, as with the verification of interlocking A, we have verified for interlocking B that a point can not be in both normal and reverse positions at the same time. With regards to the verification of this condition for train station A, the result was that the interlocking upheld the safety condition. Interestingly, Figure 9.6 shows that this safety condition is also respected by the interlocking for train station B.

Overall, Figure 9.6 shows that firstly, inductive verification succeeds and secondly, how many iterations of our forwards reachability algorithm was possible, with and without the application of slicing. Interestingly, even though this train station is more complex from an Invensys point of view, the verification times, and number of possible iterations both improve when compared to the results from train station A. We see that a much larger number of iterations is possible before the program exceeds the machine memory, I.e., 20000. Although we still have the space restriction issue at this point.

Verification Type	**Result**	**Time Taken(s)**
Inductive Base	Succeed	0.24
Inductive Step	Succeed	0.21
Forward Iteration (BMC)	Safe – Bound = 10	2.01
	Safe – Bound = 20	4.05
	Safe – Bound = 50	11.03
	Safe – Bound = 100	24.61
	Safe – Bound = 200	58.71
	Safe – Bound = 500	220.28
	Safe – Bound = 1000	668.85
	Safe – Bound = 2000	2267.73
	Unknown – Bound = 3000	Out Of Memory
Forward Iteration (BMC) with slicing	Safe – Bound = 10	0.26
	Safe – Bound = 20	0.46
	Safe – Bound = 50	1.12
	Safe – Bound = 100	2.21
	Safe – Bound = 200	4.61
	Safe – Bound = 500	12.79
	Safe – Bound = 1000	29.46
	Safe – Bound = 2000	73.84
	Safe – Bound = 3000	133.13
	Safe – Bound = 4000	205.49
	Safe – Bound = 5000	293.63
	Safe – Bound = 10000	958.26
	Safe – Bound = 20000	1706.19
	Unknown – Bound = 50000	Out Of Memory

Figure 9.6: Our bounded verification results for safety condition 2.

Next, in Figure 9.7, we can see the results of our unbounded approaches when applied the verification problem for this safety condition. Here again we see an improvement over the results from the previous train station, with respect to the same safety condition. Also we notice that once again temporal induction proves that inclusion is reached, and the forward iteration approach with inclusion check does not.

Verification Type	**Result**	**Time Taken(s)\ Inclusion Reached**
Forward Iteration (BMC) with inclusion	Safe – Bound = 10	8.17\No
	Safe – Bound = 20	32.47\No
	Safe – Bound = 50	333.39\No
	Unknown – Bound = 100	Out Of Memory
Forward Iteration (BMC) with inclusion and slicing	Safe – Bound = 10	1.13\No
	Safe – Bound = 20	3.90\No
	Safe – Bound = 50	26.43\No
	Safe – Bound = 100	153.81\No
	Safe – Bound = 200	1389.47\No
	Unknown – Bound = 500	Out Of Memory
Temporal Induction with slicing	Safe – Bound = 10	2.57\Yes

Figure 9.7: Our bounded verification results for safety condition 2.

Finally, we note that is is interesting to see that this safety condition is upheld with regards to both interlockings. This could be due to the fact that Invensys re-use some parts of ladder logic programs to form new ladder logic programs. If this safety condition speaks about a ladder logic segment that is common between many interlocking programs, then it could be interesting to explore whether or not other features of these ladder logic programs effect the validity of this condition. In this sense we may be able to form a composition based tool for verification. Later in Chapter 10 we shall consider this idea further.

9.3.3 Signal X Yellow then Signal Y Red

The final safety condition which we applied our verification tool to was one concerning the signals at the given train station. The safety condition basically describes that if one particular signal is showing yellow, than it must be the case that the next signal along the track is showing red. I.e., "$S1.yellow \implies S2.red$". As described in Chapter 2 this situation is quite a common rule in signalling systems.

Verification Type	**Result**	**Time Taken(s)**
Inductive Base	Succeed	0.24
Inductive	Fail	0.18
Forward Iteration (BMC)	Trace length 2	0.64
Forward Iteration (BMC) with slicing	Trace length 2	0.39

Figure 9.8: Our verification results for safety condition 3.

This safety condition was found not to hold in the inductive step verification, and hence a counter example trace was generated by our forward iteration algorithm. These results are summarised in Figure 9.8.

We can once again see, in Figure 9.8 that our slicing algorithm improves the efficiency of counter example generation. Also, at this point it is interesting to note that like all the other safety conditions that have failed verification, we have been able to successfully produce a counter example trace using our forward iteration approach.

In the next section we shall give a discussion of these results, commenting on what effect they have from a theoretical and practical point of view.

9.4 Analysis of Results

Overall, the results we have gained have been positive. For every safety condition we have verified, the tool has either given a successful verification, or a counter example trace has been constructed. This meets the initial goals of the project, namely to explore the feasibility of SAT based verification of interlockings. It also means, that the tool is of practical use to Invensys. The interesting result from a theoretical point of view is that we did not manage to show inclusion with respect to our forward iteration approach. Below we shall comment on some general outcomes of each verification approach, along with the implications of such results in both a practical and theoretical sense.

9.4.1 Results of Inductive Verification

The results gained from our inductive verification approach were as to be expected. That is, in the case of interlocking A, the results were in line with the results obtained by Kanso. With regards to interlocking B, inductive verification proved successful in only one out of three verification processes. Again this is to be expected given the problem of unreachable states. The two interesting points with regards to inductive verification are the following:

- In all cases where inductive verification failed, a counter example trace was constructed. Hence the verification via induction happens to be correct. This raises the question of how often does inductive verification actually fail to give an incorrect result due to the problem of unreachable states.
- The inductive verification always completes very quickly (less than a second). This is a improvement on the verification times obtained by Kanso, which were usually in the region of a few minutes.

These positive results show that inductive verification should always be the first approach to be used when verifying a safety condition. The very quick production of a result means that it is not costly to run such a process. If the result happens to be positive, then there is no need to apply other verification approaches. Whereas if the result is negative, the forward or backward iteration approach could be used to give a full counter example trace at the expense that constructing such a trace is more costly in terms of time. The inclusion of the inductive verification approach in the tool is justified exactly for these reasons.

9.4.2 Results of Bounded Model Checking

The series of results gained from our forward iteration model checking approaches can be split into several sub approaches, namely,

- forward iteration,
- forward iteration with slicing,
- forward iteration with inclusion, and
- forward iteration with inclusion and slicing.

We will now comment on each of the four approaches in turn.

The basic forward iteration approach proved to be a useful technique in the generation of counter example traces. In all the verification results where inductive verification gave a counter example, the forward iteration approach was successfully able to construct a counter example trace. We note that all counter example traces constructed were fairly short, i.e., less than five iterations. Hence, with such a small number of iterations required, this basic approach proved to be efficient in creating counter example traces. Results obtained from safety conditions shown to hold by inductive verification, showed that forward iteration was possible up to two thousand iterations before memory issues were incurred. This is a large number of iterations, but given that there is no inclusion check encoded with the approach, it gives no idea as to how many iterations would be required to verify all reachable states. Therefore, to conclude, the most successful use of the forward iteration approach would be for the construction of counter example traces.

When slicing was applied to the forward iteration approach, a large improvement in both time and space efficiency was recorded. With the application of slicing, counter example generation times were reduced by over half in all cases, and in some cases the counter example traces were even reduced in length. The reduction in length of a counter example trace is beneficial to the engineers at Invensys, as it of course means less manual processing is required when compared to larger counter example traces. For this reason, the application of slicing to the forward iteration approach is not only useful for efficiency reasons, but also for practical reasons such as counter example trace length. Finally, the application of slicing allowed the verification of up to twenty thousand iterations, in comparison to only two thousand iterations being possible without slicing.

The next aspect shown by the results is the effect of adding an inclusion check to our forward iteration approach. The addition of an inclusion check property has a definite impact on the overall efficiency in terms of both space and time. In all cases where an inclusion check was included in our forward iteration approach, the time taken for the verification increased over ten fold. Also, the number of iterations possible with inclusion is reduced by over ten fold. Given that none of the results shows that inclusion has been reached, a more efficient method of encoded inclusion is needed.

Again, when slicing was applied to the forward iteration approach with inclusion check, the result is quite clear. Namely, slicing once again successfully reduced the verification time in all cases. Also, the application of slicing allowed up to two hundred iterations of the forward iteration with inclusion approach. This being double the amount of iterations that was possible without the application of slicing. The think that is most noticeable, is that even though slicing gives good results with regards to reducing the complexity of the problem, once again inclusion was not reached. This also means that a more efficient inclusion check is required for effective use of our forward iteration approach to be complete.

9.4.3 Results of Temporal Induction

The final verification results we shall comment on are those obtained from the temporal inductions approach to verification. The results obtained here are maybe the least interesting, as whenever inductive verification succeeded, by the way that we construct our temporal induction formula, so did temporal induction. Also, whenever a counter example was generated using the forward iterations technique, the same counter example would also have been generated by temporal induction. These two results show that temporal induction works correctly, however they do not demonstrate the full power of temporal induction. To illustrate the full power, we would need to find a safety condition which is incorrectly verified by the inductive approach, and which also fails to reach inclusion via the forward iteration approach. Then we could apply the temporal induction approach to see if it could be used to reach inclusion in such a case. If, in such a case temporal induction was successful in reaching inclusion, then it would make it the most suited verification approach to be used by Invensys. This is due to the fact that it combines both the power of inductive verification with the possible production of counter example traces. For this reason, we think a good extension to the results presented here would be to experiment further with the temporal induction approach to verification.

9.4.4 Results of Slicing

Finally, to conclude our summary of the results obtained, we shall comment further on the successfulness of applying slicing o the verification approaches.

Overall, all the results obtained show that applying our slicing method to the formulae involved in the verification process gives a large efficiency increase. Some analysis of the application of the slicing algorithm have shown that the following reductions were possible in some of the above verification tests:

- For interlocking A, the number of rungs contained in the ladder logic propositional formula, was reduced, on average from 599 rungs to the region of 60 to 80 rungs.
- For interlocking B, the number of rungs contained in the ladder logic propositional formula, was reduced on average from 238 rungs to between 25 to 50 rungs.

These figures illustrate exactly why the results improved so dramatically when applying slicing. Given that the number of rungs in each ladder logic propositional formula was reduced by up to ten times, results in the overall formulae to be verified being up to ten times smaller. Obviously, with slicing we have that the resultant formula size is dependant on the safety condition being verified. Hence here it would be interesting to see the effect slicing has on much more complicated, larger interlockings. With the simple examples train stations we have used, there are only few physical track components, and hence each track component can only be dependant on a few others. This means that values in the ladder logic program are only dependant on small numbers of other values. If we consider a larger train stations, then there will be many more physical track components, which all depend on each other in much more complex ways. Hence here it would be interesting to see if the application of slicing still has a reduction effect in the region of ten fold.

We have now given a review of all the verification result we have obtained with our tool. Now we shall move on to discuss the outcome of the overall project, giving suggestions for some interesting aspects that could be explored in future work.

Chapter 10

Conclusions and Future Improvements

Contents

We now bring this thesis to a close by commenting on the work that has been completed. Finally, we give an overview for possible directions of future work.

10.1 Summary

In this thesis, we have completed a feasibility study into various techniques for SAT based model checking of Westrace interlockings. We have provided a modelling process for Westrace interlockings via propositional logic and given an automaton theoretic semantics for this propositional model. We have studied in some depth, the verification processes of inductive verification, bounded model checking, unbounded model checking via inclusion and temporal induction. As a natural continuation from this, we have reviewed how a slicing algorithm can be applied to reduce the complexity of the verification problem, showing the correctness of its application. The overall outcome being the development of a verification tool, with varied verification techniques on offer.

To show the success of our approach, we have applied our tool to the verification problem for two real world interlocking systems. This illustrates several key points:

- The approaches we propose work. That is, an interlocking can successfully be verified with respect to some safety condition. The result being either that the interlocking is safe, or that a counter example trace is generated.
- The approaches we propose scale up to real world systems.
- SAT based verification is a successful method of verifying large systems.

Finally, the main contribution of this work has been concerned with the practical needs of Invensys. Through a sound theoretical basis, we have shown how slicing can successfully be used to allow such verification processes to be highly successful within the domain of verifying interlocking systems. We have shown the theoretical limits of SAT based verification techniques, illustrating that in an industrial setting these theoretical limits are very rarely reached.

10.2 Possible Future Work

We shall now comment on several aspects of work that should be undertaken to follow on from this project. Firstly, it would be interesting to see if there are further techniques that can be applied to reduce the complexity of the verification process further. This in turn will allow our iteration algorithms to be executed up to a larger bound, maybe even to the stage where inclusion is reached. Secondly, with our current approaches towards encoding a inclusion check, we have yet to reach inclusion. Hence, further research into encoding an inclusion check efficiently would be required. Thirdly, a parser should be created to allow more effective use of the ability to apply our backwards iteration algorithm to counter examples generated by the tool. Finally, the potential of our tool should be demonstrated further through the verification of a greater number of interlockings.

10.2.1 Functional Dependencies

One possibility to reduce the size of the formulae involved in our verification process is via the extraction of functional dependencies. Like the slicing algorithm we have proposed, the removal of functional dependencies would help to remove variables that are not necessarily needed within the verification process. To explain functional dependencies further, consider once more our pelican crossing ladder logic program modelled as a propositional formula.

$$\begin{array}{l}
[crossing' \iff (req \land \neg crossing), \\
req' \iff (pressed \land \neg req), \\
tlag' \iff ((\neg crossing') \land (\neg pressed \lor req')), \\
tlbg' \iff ((\neg crossing') \land (\neg pressed \lor req')), \\
tlar' \iff crossing', \\
tlbr' \iff crossing', \\
plag' \iff crossing', \\
plbg' \iff crossing', \\
plar' \iff (\neg crossing'), \\
plbr' \iff (\neg crossing'), \\
audio' \iff crossing']
\end{array}$$

Figure 10.1: An example ladder logic program.

Figure 10.1 shows that some of the coils occurring in the pelican crossing ladder logic program are dependant on similar values. For example, both $tlar'$ and $tlbr'$ are dependant on exactly the same value, namely *crossing'*. Therefore, to reduce the formula size, it would make sense to perform some pre-processing to remove one of these variables. That is, we could remove *tlbr'* from the formulae to be verified, and then simple obtain its value using the value of *tlar'* after the verification process. Of course, this notion can be extended not only to variables that depend on the same value, but to variables that depend on each other via some logical function f. Again considering our example in Figure 10.1, we can see that the value of *plar'* can be derived by negating the value of *plag'*. Hence here, the function f relating the value of *plar'* to the value of *plag'* can simply be described as a logical negation.

This simple example illustrates the idea of functional dependency removal. To give a more formal definition, we have the following,

Definition 10.1 (Functional Dependancy)**:** A variable x is functionally dependant on variables $x_1, \ldots, x_n$ if there exists a function f such that

$$x = f(x_1, \ldots, \_n).$$

Now if we were to remove such functional dependencies from a ladder logic program, the hope would be that the verification process would be simplified further. The interesting theoretical problem with functional dependencies is how to compute what variables are functionally dependant on other variables, and through what function. Some work has been completed into such topics in [LJHM07, LW09, HD93], but it would be interesting to see how the notion of functional dependencies could be applied to the verification problem for ladder logic.

10.2.2 Explore Inclusion Check Further

The current methods we have used to encode an inclusion check into our forward iteration approaches is by using an over approximation. This over approximation, namely the length of the longest loop free path, has proved to expensive to add into the verification process, as not once have we managed to reach inclusion. This is due to that fact that the longest loop free path through a system may be much larger than the bound that is needed to reach inclusion. Also, encoding that a path is loop free also causes a increase in overall formula size. For these reasons, it would be interesting to research further into encoding inclusion checks. One possibility to do this could be to analyse the type of state space gained from ladder logic programs, to see in general if a large or small bound is required for inclusion. Another possibility could be to consider further relationships within the ladder logic program, to see how various parts of the ladder logic program effect the automaton. Again, considerations would have to be taken to try to make such an approach as generic as possible, but to begin maybe relationships between a few small ladder logic programs could be explored.

10.2.3 Counter Example Parser

As previously mentioned, the functionality to perform backwards iteration based verification has been included in the tool. Also some hard coded examples have been used to experiment further with this technique. What would now be of interest to Invensys, is to build another part to our verification tool to parse counter example files. Currently, counter example traces are generated in the TSTP format [tpt09]. This format is very similar to the input format used for the Paradox model finder we have used within our tool. The syntax of the counter examples is reasonable simple, and hence the construction of a simple parser for such counter examples would be time well spent. Once such a counter example has been parsed, a simply transformation would be required to change the result into a format of an initial condition for the verification process. This in turn would allow further tests to be completed on the reachability of counter examples generated from the inductive verification approach included in the tool. Again, this whole process could be completely automated when a counter example is found via inductive verification, making the overall verification process very simple for the engineers at Invensys.

10.2.4 Tool Integration and Compositional Reasoning

The final point we give as possible future work, is not so much involved with theoretical aspects or limitations of the tool, but more with how it can aid further the engineers in Invensys. Currently Invensys have a specific tool set they use for the creation of ladder logic programs and hence the control programs for their interlockings. What we propose here is it that the tool we have given is built into this tool set. This has two advantages from a practical point of view. Firstly, the engineers are already using such tools, so through combination, using this tool would be no more of a deviation than just clicking a button and maybe entering a safety condition, or selecting one from a given list.

Secondly, the integration with the tool could be completed in such a way that the verification process can actually be improved further. Currently Invensys use so called ladder logic “templates”. These templates describe common scenario’s that arise in many train stations, and can then be combined to give a full interlocking program. If the tool enabled the verification of such templates. Then under certain conditions, it might be possible to guarantee that combining two templates has the property that the safety conditions verified earlier would still be upheld. The advantages of such a compositional reasoning to the verification process are obvious. All templates would only need to be verified once, an then the composition of such templates just has to ensure that the required side conditions are met. Also, with respect to efficiency, it may be much less complex to verify templates than whole ladder logic programs. Hence the verification process would be quicker overall. This would also make understanding of counter example traces easier, as they would contain less components to be considered. Overall, we believe the exploration of such compositional reasoning techniques would be the perfect way to extend our tool.

Bibliography

[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 J. Paul, editors, *CHARME*, volume 3725 of *Lecture Notes in Computer Science*. Springer, 2005.

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, 27(6), 1978.

[BAB$^+$95] Anselmi Bernardeschi, A. Anselmi, C. Bernardeschi, A. Fantechi, S. Gnesi, S. Larosa, G. Mongardi, and F. Torielli. An experience in formal verification of safety properties of a railway signalling control system. In G. Rabe, editor, *in Proceedings of the SAFECOMP'95 Conference, Belgirate*. Springer - Verlag, 1995.

[BC00] Per Bjesse and Koen Claessen. SAT-based verification without state space traversal. In Jr. Warren A. Hunt and Steven D. Johnson, editors, *In Formal Methods in Computer-Aided Design*, pages 372–389. Springer, 2000.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In Rance Cleaveland, editor, *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*. Springer-Verlag, 1999.

[BFG$^+$98] Cinzia Bernardeschi, Alessandro Fantechi, Stefania Gnesi, Salvatore Larosa, Giorgio Mongardi, and Dario Romano. A formal verification environment for railway signaling system design. *Form. Methods Syst. Des.*, 12(2), 1998.

[BG00] J. Boulanger and M. Gallardo. Validation and verification of METEOR safety software. In *Advances in Transport Vol 7*, pages 189–200. WIT Press, 2000.

[BHvMW09] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[Bjo09] D. Bjoerner. Towards a domain model of transportation. In *Domain Engineering – Technology Management, Research and Engineering*, pages 333–358. JAIST Press, 2009.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* The MIT Press, May 2008.

[Bol06] William Bolton. *Programmable Logic Controllers.* Newnes, 2006.

[Bur02] Anthony Burton. *Richard Trevithick: Giant of Steam.* Aurum Press, 2002.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logic of Programs, Workshop.* Springer-Verlag, 1981.

[CESS08] Koen Claessen, Niklas Een, Mary Sheeran, and Niklas Sörensson. SAT-solving in practice. In Bengt Lennartson, Martin Fabian, Knut Åkesson, Alessandro Giua, and Ratnesh Kumar, editors, *Proceedings of Workshop on Discrete Event Systems (WODES).* IEEE, May 2008.

[CGJ$^+$01] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. *Lecture Notes in Computer Science*, 2000, 2001.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* The MIT Press, January 1999.

[cha09] zChaff, Webpage, last accessed in October 2009. `http://www.princeton.edu/ chaff/zchaff.html`.

[Cla08] Edmund M. Clarke. The birth of model checking. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science.* Springer, 2008.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael Harrison, Ranan Banerji, and Jeffrey Ullman, editors, *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing.* ACM, 1971.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7), 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *ACM*, 7(3), 1960.

[ES03] Niklas Een and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4), 2003. BMC'2003, First International Workshop on Bounded Model Checking.

[ES04] Niklas Een and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT 2003*, LNCS 2919. Springer, 2004.

[FH98] Wan Fokkink and Paul Hollingshead. Verification of interlockings: from control tables to ladder logic diagrams. In J.F. Groote, S.P. Luttik, and J.J. van Wamel, editors, *FMICS'98.* CWI, 1998.

[Fok96] Wan Fokkink. Safety criteria for the vital processor interlocking at Hoorn-Kersenboogerd. In J. Allan, C.A. Brebbia, R.J. Hill, G. Sciutto, and S. Sone, editors, *Proceedings of the 5th Conference on Computers in Railways (COMPRAIL'96), Volume I: Railway Systems and Management.* Computational Mechanics Publications, 1996.

[GL91] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transaction. Software Engineering*, 17(8), 1991.

[GPFW96] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the satisfiability (SAT) problem: A survey. In Dingzhu Du, Jun Gu, and Panos M. Pardalos, editors, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science.* American Mathematical Society, 1996.

[GRV08] Silvio Ghilardi, Silvio Ranise, and Thomas Valsecchi. Light-weight SMT-based Model Checking. In Muffy Calder and Alice Miller, editors, *Proceedings of AVoCS 2008.* University of Glasgow, 2008.

[HD93] Alan J. Hu and David L. Dill. Reducing BDD size by exploiting functional dependencies. In Alfred E. Dunlop, editor, *DAC '93: Proceedings of the 30th international Design Automation Conference.* ACM, 1993.

[HD95] Mark Harman and Sebastian Danicic. Using program slicing to simplify testing. *Journal of Software Testing, Verification and Reliability*, 5, 1995.

[HET09] HETS, Webpage, last accessed in October 2009. `http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/hets/index_e.htm`.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[HP00] Anne E. Haxthausen and Jan Peleska. Formal development and verification of a distributed railway control system. *IEEE Trans. Softw. Eng.*, 26(8), 2000.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science.* Cambridge University Press, 2004.

[Hut07] Graham Hutton. *Programming in Haskell.* Cambridge University Press, January 2007.

[HvM09] Marijn J. H. Heule and Hans van Maaren. *Look-Ahead Based SAT Solvers*, chapter 5, pages 155–184. Volume 185 of Biere et al. [BHvMW09], February 2009.

[IEC03] Programmable Controllers - Part 3: Programming languages, 2003. IEC Standard 61131-3.

[INE09] INESS, Webpage, last accessed in October 2009. `http://www.iness.eu/`.

[inv09] Invensys Rail, Webpage, last accessed October 2009. `http://www.invensysrail.com/`.

[Jac04] René Jacquart, editor. *IFIP 18th World Computer Congress, Topical Sessions*, chapter TRain: The Railway Domain - A Grand Challenge. Kluwer, 2004.

[Jam09a] Phillip James. Verifying Train Control Software, 2009. Presentation at BCTCS 2009, `http://www2.warwick.ac.uk/fac/sci/dcs/events/bctcs/` webpage, last accesses in October 2009.

[Jam09b] Phillip James. Verifying Train Control Software Using SAT-based Model Checking., 2009. Presentation at VINO09, P.h.D. workshop , Italy.

[JB04] Jie-Hong Roland Jiang and Robert K. Brayton. Functional dependency for verification reduction. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification*, LNCS 2919. Springer, 2004.

[JIR09] Phillip James, Yoshinao Isobe, and Markus Roggenbach. Verifying Train Control Software - An exercise in SAT-based Model Checking. In *11th JSSST Workshop on Programming and Programming Languages (PPL2009)*. Japan Society for Software Science and Technology, 2009.

[JK09] Phillip James and Karim Kanso. Automated Verification of Train Control Software, 2009. Presentation at Swansea Science Day.

[JR09] Phillip James and Markus Roggenbach. SAT-based Model Checking of Train Control Systems. Technical report, University of Udine, September 2009.

[Kan08] Karim Kanso. Formal verification of ladder logic. Master's thesis, Swansea University, 2008.

[KMS08] Karim Kanso, Faron Moller, and Anton Setzer. Verfication of safety properties in railway interlocking systems defined with ladder logic. In Muffy Calder and Alice Miller, editors, *AVOCS08*, Glasgow 2008.

[KP07] Stephanie Kemper and André Platzer. SAT-based abstraction refinement for real-time systems. In Frank S. de Boer and Vladimir Mencl, editors, *Formal Aspects of Component Software, Third International Workshop, FACS 2006, Prague, Czech Republic, Proceedings*, volume 182 of *ENTCS*, 2007.

[KR01] David Kerr and Tony Rowbotham. *Introduction To Railway Signalling*. Institution of Railway Signal Engineers, 2001.

[KR09] Temesghen Kahsai and Markus Roggenbach. Property preserving refinement for Csp-Casl. *Recent Trends in Algebraic Development Techniques: 19th International Workshop, WADT 2008, Pisa, Italy, June 13-16, 2008, Revised Selected Papers*, 2009.

[Kro99] T. Kropf. *Introduction to formal hardware verification*. Springer Verlag, 1999.

[Kro09] Daniel Kroening. *Software Verification*, chapter 16, pages 505–532. Volume 185 of Biere et al. [BHvMW09], February 2009.

[KSS09] Henry A. Kautz, Ashish Sabharwal, and Bart Selman. *Incomplete Algorithms*, chapter 6, pages 185–203. Volume 185 of Biere et al. [BHvMW09], February 2009.

[Kul08] Oliver Kullmann. The OKlibrary: A generative research platform for (generalised) SAT solving. Technical Report CSR 1-2008, Swansea University, Computer Science Report Series, February 2008.

[Lea91] Maurice Leach. *Railway Control Systems.* A and C Black Publishers Ltd., 1991.

[Lev96] William Levine, editor. *The Control Handbook (Electrical Engineering Handbook).* Crc Press, 1996.

[LJHM07] Chih-Chun Lee, Jie-Hong R. Jiang, Chung-Yang (Ric) Huang, and Alan Mishchenko. Scalable exploration of functional dependency by interpolation and incremental SAT solving. In Georges Gielen, editor, *ICCAD '07: Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design.* IEEE Press, 2007.

[LW09] Chen-Hsuan Lin and Chun-Yao Wang. Dependent latch identification in the reachable state space. In Kazutoshi Wakabayashi, editor, *ASP-DAC '09: Proceedings of the 2009 Asia and South Pacific Design Automation Conference.* IEEE Press, 2009.

[min09] Minisat, Webpage, last accessed in October 2009. `http://minisat.se/Main.html`.

[MO07] Maura Mazzarello and Ennio Ottaviani. A traffic management system for real-time traffic optimisation in railways. *Transportation Research Part B: Methodological*, 41(2), 2007. Advanced Modelling of Train Operations in Stations and Networks.

[Mos04] P. D. Mosses, editor. *CASL Reference Manual.* LNCS 2960. Springer, 2004.

[MP91] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer, 1991.

[MSLM09] Joao P. Marques-Silva, Ines Lynce, and Sharad Malik. *Conflict-Driven Clause Learning SAT Solvers*, chapter 4, pages 131–153. Volume 185 of Biere et al. [BHvMW09], February 2009.

[Noc85] 0.S. Nock. *Railway Signalling.* A and C Black Publishers Ltd., 1985.

[par09] Paradox model finder, Webpage, last accessed in October 2009. `http://www.cs.chalmers.se/ koen/folkung/`.

[PGHD04] J. Peleska, D. Große, A. E. Haxthausen, and R. Drechsler. Automated verification for train control systems. In *Proceedings of Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2004)), Braunschweig*, 2004.

[PP06] Wojciech Penczek and Agata Pólrola. *Advances in Verification of Time Petri Nets and Timed Automata: A Temporal Logic Approach (Studies in Computational Intelligence)*. Springer-Verlag New York, Inc., 2006.

[Sim94] Andrew Simpson. A formal specification of an automatic train protection system. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *FME '94: Proceedings of the Second International Symposium of Formal Methods Europe on Industrial Benefit of Formal Methods*. Springer-Verlag, 1994.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-Solver. In Warren Hunt Jr and Steven Johnson, editors, *FMCAD '00: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*. Springer-Verlag, 2000.

[Thi09] Harold Thimbleby. Contributing to safety and due diligence in safety-critical interactive systems development by generating and analyzing finite state models. In Gaelle Calvary, T. C. Nicholas Graham, and Philip Gray, editors, *EICS '09: Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*. ACM, 2009.

[Tip95] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995.

[tpt09] The TPTP problem library for automated theorem proving, Webpage, last accessed October 2009. `http://www.cs.miami.edu/ tptp/`.

[Tra09] TRain, Webpage, last accessed in October 2009. `http://www.railwaydomain.org/`.

[upp09] Uppaal tool, Webpage, last accessed in October 2009. `http://www.uppaal.com/`.

[vpi09] VPI, Webpage, last accessed in October 2009. `http://www.alstomsignalingsolutions.com/OurProducts/WaysideProducts/InterlockingSolutions/VPI/`.

[Wei81] Mark Weiser. Program slicing. In S. Jeffrey and L. Stucki, editors, *ICSE '81: Proceedings of the 5th international conference on Software engineering*. IEEE Press, 1981.

[wes09] Invensys Rail, Westrace, Webpage, last accessed October 2009. `http://www.wrsl.com/assets/files/Interlocking/westrace/WESTRACE%20Intorduction.pdf`.

[Win02] Kirsten Winter. Model checking railway interlocking systems. *Australian Computer Science Communications*, 24(1), 2002.

[WR03] Kirsten Winter and Neil J. Robinson. Modelling large railway interlockings and model checking small ones. In Michael J. Oudshoorn, editor, *ACSC '03: Proceedings of the 26th Australasian computer science conference*. Australian Computer Society, Inc., 2003.

[ZKvH01] Peter J. Zwaneveld, Leo G. Kroon, and Stan P. M. van Hoesel. Routing trains through a railway station based on a node packing model. *European Journal of Operational Research*, 128(1):14 – 33, 2001.

[ZRK03] Bohumir Zoubek, Jean-Marc Roussel, and Marta Kwiatowska. Towards automatic verification of ladder logic programs. In *Proceedings of IMACS-IEEE and CESA'03*, 2003.

Appendix A

Inductive Verification

library KANSO

logic PROPOSITIONAL

spec STATE0 =
crossing0 req0 tlag0 tlbg0 tlar0 tlbr0 plag0 plbg0 plar0 plbr0 audio0 pressed0
end

spec STATE1 =
crossing1 req1 tlag1 tlbg1 tlar1 tlbr1 plag1 plbg1 plar1 plbr1 audio1 pressed1
end

spec STATE2 =
crossing2 req2 tlag2 tlbg2 tlar2 tlbr2 plag2 plbg2 plar2 plbr2 audio2 pressed2
end

spec TRANSITION[STATE0][STATE1] =

$crossing1 \Leftrightarrow req0 \wedge \neg\ crossing0$

$req1 \Leftrightarrow pressed0 \wedge \neg\ req0$

$tlag1 \Leftrightarrow \neg\ crossing1 \wedge (\neg\ pressed0 \vee req1)$

$tlbg1 \Leftrightarrow \neg\ crossing1 \wedge (\neg\ pressed0 \vee req1)$

$tlar1 \Leftrightarrow crossing1$

$tlbr1 \Leftrightarrow crossing1$

$plag1 \Leftrightarrow crossing1$

$plbg1 \Leftrightarrow crossing1$

$plar1 \Leftrightarrow \neg\ crossing1$

$plbr1 \Leftrightarrow \neg\ crossing1$

$audio1 \Leftrightarrow crossing1$

```
end

spec INITIAL[STATE0] =
     ¬ pressed0
end

spec KANSOCONDITIONONE =
     STATE0
and  STATE1
and  STATE2
and  INITIAL[STATE0]
then TRANSITION[STATE0][STATE1]
and  TRANSITION
     [STATE1 fit
      req0 ↦ req1 crossing0 ↦ crossing1 pressed0 ↦ pressed1
      tlag0 ↦ tlag1 tlbg0 ↦ tlbg1 tlar0 ↦ tlar1 tlbr0 ↦ tlbr1
      plag0 ↦ plag1 plar0 ↦ plar1 plbg0 ↦ plbg1 plbr0 ↦ plbr1
      audio0 ↦ audio1]
     [STATE2 fit
      req1 ↦ req2 crossing1 ↦ crossing2 pressed1 ↦ pressed2
      tlag1 ↦ tlag2 tlbg1 ↦ tlbg2 tlar1 ↦ tlar2 tlbr1 ↦ tlbr2
      plag1 ↦ plag2 plar1 ↦ plar2 plbg1 ↦ plbg2 plbr1 ↦ plbr2
      audio1 ↦ audio2]
then %implies
     (tlag1 ∨ tlar1) ∧ ¬ (tlag1 ∧ tlar1) ∧ (tlbg1 ∨ tlbr1)
     ∧ ¬ (tlbg1 ∧ tlbr1)
                                                        %(Initial => phi)%
end

spec KANSOCONDITIONTWO =
     STATE0
and  STATE1
and  STATE2
then (tlag0 ∨ tlar0) ∧ ¬ (tlag0 ∧ tlar0)
     ∧ (tlbg0 ∨ tlbr0) ∧ ¬ (tlbg0 ∧ tlbr0)
then TRANSITION[STATE0][STATE1]
and  TRANSITION
     [STATE1 fit
      req0 ↦ req1 crossing0 ↦ crossing1 pressed0 ↦ pressed1
      tlag0 ↦ tlag1 tlbg0 ↦ tlbg1 tlar0 ↦ tlar1 tlbr0 ↦ tlbr1
      plag0 ↦ plag1 plar0 ↦ plar1 plbg0 ↦ plbg1 plbr0 ↦ plbr1
      audio0 ↦ audio1]
     [STATE2 fit
      req1 ↦ req2 crossing1 ↦ crossing2 pressed1 ↦ pressed2
      tlag1 ↦ tlag2 tlbg1 ↦ tlbg2 tlar1 ↦ tlar2 tlbr1 ↦ tlbr2
      plag1 ↦ plag2 plar1 ↦ plar2 plbg1 ↦ plbg2 plbr1 ↦ plbr2
```

```
    $audio1 \mapsto audio2]$
then %implies
    $(tlag1 \vee tlar1) \wedge \neg (tlag1 \wedge tlar1) \wedge (tlbg1 \vee tlbr1)$
    $\wedge \neg (tlbg1 \wedge tlbr1)$
end
```

Appendix B

Forwards Reachability – Incorrect Ladder Logic

library ITERATIVE

logic PROPOSITIONAL

spec STATE0 =
crossing0 req0 tlag0 tlbg0 tlar0 tlbr0 plag0 plbg0 plar0
plbr0 audio0 pressed0
end

spec STATE1 =
crossing1 req1 tlag1 tlbg1 tlar1 tlbr1 plag1 plbg1 plar1
plbr1 audio1 pressed1
end

spec STATE2 =
crossing2 req2 tlag2 tlbg2 tlar2 tlbr2 plag2 plbg2 plar2
plbr2 audio2 pressed2
end

spec STATE3 =
crossing3 req3 tlag3 tlbg3 tlar3 tlbr3 plag3 plbg3 plar3
plbr3 audio3 pressed3
end

spec STATE4 =
crossing4 req4 tlag4 tlbg4 tlar4 tlbr4 plag4 plbg4 plar4
plbr4 audio4 pressed4
end

spec STATE5 =
crossing5 req5 tlag5 tlbg5 tlar5 tlbr5 plag5 plbg5 plar5
plbr5 audio5 pressed5
end

spec STATE6 =
crossing6 req6 tlag6 tlbg6 tlar6 tlbr6 plag6 plbg6 plar6
plbr6 audio6 pressed6
end

spec STATE7 =
crossing7 req7 tlag7 tlbg7 tlar7 tlbr7 plag7 plbg7 plar7
plbr7 audio7 pressed7
end

spec TRANSITION[STATE0][STATE1] =
$crossing1 \Leftrightarrow req0 \wedge \neg\ crossing0$
$req1 \Leftrightarrow pressed0 \wedge \neg\ req0$
$tlag1 \Leftrightarrow \neg\ crossing1$
$tlbg1 \Leftrightarrow \neg\ crossing1 \wedge \neg\ pressed0$
$tlar1 \Leftrightarrow crossing1$
$tlbr1 \Leftrightarrow crossing1$
$plag1 \Leftrightarrow crossing1$
$plbg1 \Leftrightarrow crossing1$
$plar1 \Leftrightarrow \neg\ crossing1$
$plbr1 \Leftrightarrow \neg\ crossing1$
$audio1 \Leftrightarrow crossing1$
end

spec INITIAL[STATE0] =
$\neg\ crossing0$
$\neg\ req0$
$\neg\ pressed0$
$\neg\ tlag0$
$\neg\ tlbg0$
$\neg\ tlar0$
$\neg\ tlbr0$
$\neg\ plag0$
$\neg\ plbg0$
$\neg\ plar0$
$\neg\ plbr0$
$\neg\ audio0$
end

spec FORWARDSITERATION =
tt ff

```
and  State0
and  State1
and  State2
and  State3
and  State4
and  State5
and  State6
and  State7
and  Initial[State0]
then Transition[State0][State1]
and  Transition
     [State1 fit
      $req0 \mapsto req1$ $crossing0 \mapsto crossing1$ $pressed0 \mapsto pressed1$
      $tlag0 \mapsto tlag1$ $tlbg0 \mapsto tlbg1$ $tlar0 \mapsto tlar1$ $tlbr0 \mapsto tlbr1$
      $plag0 \mapsto plag1$ $plar0 \mapsto plar1$ $plbg0 \mapsto plbg1$ $plbr0 \mapsto plbr1$
      $audio0 \mapsto audio1$]
     [State2 fit
      $req1 \mapsto req2$ $crossing1 \mapsto crossing2$ $pressed1 \mapsto pressed2$
      $tlag1 \mapsto tlag2$ $tlbg1 \mapsto tlbg2$ $tlar1 \mapsto tlar2$ $tlbr1 \mapsto tlbr2$
      $plag1 \mapsto plag2$ $plar1 \mapsto plar2$ $plbg1 \mapsto plbg2$ $plbr1 \mapsto plbr2$
      $audio1 \mapsto audio2$]
and  Transition
     [State2 fit
      $req0 \mapsto req2$ $crossing0 \mapsto crossing2$ $pressed0 \mapsto pressed2$
      $tlag0 \mapsto tlag2$ $tlbg0 \mapsto tlbg2$ $tlar0 \mapsto tlar2$ $tlbr0 \mapsto tlbr2$
      $plag0 \mapsto plag2$ $plar0 \mapsto plar2$ $plbg0 \mapsto plbg2$ $plbr0 \mapsto plbr2$
      $audio0 \mapsto audio2$]
     [State3 fit
      $req1 \mapsto req3$ $crossing1 \mapsto crossing3$ $pressed1 \mapsto pressed3$
      $tlag1 \mapsto tlag3$ $tlbg1 \mapsto tlbg3$ $tlar1 \mapsto tlar3$ $tlbr1 \mapsto tlbr3$
      $plag1 \mapsto plag3$ $plar1 \mapsto plar3$ $plbg1 \mapsto plbg3$ $plbr1 \mapsto plbr3$
      $audio1 \mapsto audio3$]
and  Transition
     [State3 fit
      $req0 \mapsto req3$ $crossing0 \mapsto crossing3$ $pressed0 \mapsto pressed3$
      $tlag0 \mapsto tlag3$ $tlbg0 \mapsto tlbg3$ $tlar0 \mapsto tlar3$ $tlbr0 \mapsto tlbr3$
      $plag0 \mapsto plag3$ $plar0 \mapsto plar3$ $plbg0 \mapsto plbg3$ $plbr0 \mapsto plbr3$
      $audio0 \mapsto audio3$]
     [State4 fit
      $req1 \mapsto req4$ $crossing1 \mapsto crossing4$ $pressed1 \mapsto pressed4$
      $tlag1 \mapsto tlag4$ $tlbg1 \mapsto tlbg4$ $tlar1 \mapsto tlar4$ $tlbr1 \mapsto tlbr4$
      $plag1 \mapsto plag4$ $plar1 \mapsto plar4$ $plbg1 \mapsto plbg4$ $plbr1 \mapsto plbr4$
      $audio1 \mapsto audio4$]
and  Transition
     [State4 fit
      $req0 \mapsto req4$ $crossing0 \mapsto crossing4$ $pressed0 \mapsto pressed4$
```

$tlag0 \mapsto tlag4\ tlbg0 \mapsto tlbg4\ tlar0 \mapsto tlar4\ tlbr0 \mapsto tlbr4$
$plag0 \mapsto plag4\ plar0 \mapsto plar4\ plbg0 \mapsto plbg4\ plbr0 \mapsto plbr4$
$audio0 \mapsto audio4]$
[STATE5 **fit**
$req1 \mapsto req5\ crossing1 \mapsto crossing5\ pressed1 \mapsto pressed5$
$tlag1 \mapsto tlag5\ tlbg1 \mapsto tlbg5\ tlar1 \mapsto tlar5\ tlbr1 \mapsto tlbr5$
$plag1 \mapsto plag5\ plar1 \mapsto plar5\ plbg1 \mapsto plbg5\ plbr1 \mapsto plbr5$
$audio1 \mapsto audio5]$

and TRANSITION
[STATE5 **fit**
$req0 \mapsto req5\ crossing0 \mapsto crossing5\ pressed0 \mapsto pressed5$
$tlag0 \mapsto tlag5\ tlbg0 \mapsto tlbg5\ tlar0 \mapsto tlar5\ tlbr0 \mapsto tlbr5$
$plag0 \mapsto plag5\ plar0 \mapsto plar5\ plbg0 \mapsto plbg5\ plbr0 \mapsto plbr5$
$audio0 \mapsto audio5]$
[STATE6 **fit**
$req1 \mapsto req6\ crossing1 \mapsto crossing6\ pressed1 \mapsto pressed6$
$tlag1 \mapsto tlag6\ tlbg1 \mapsto tlbg6\ tlar1 \mapsto tlar6\ tlbr1 \mapsto tlbr6$
$plag1 \mapsto plag6\ plar1 \mapsto plar6\ plbg1 \mapsto plbg6\ plbr1 \mapsto plbr6$
$audio1 \mapsto audio6]$

and TRANSITION
[STATE6 **fit**
$req0 \mapsto req6\ crossing0 \mapsto crossing6\ pressed0 \mapsto pressed6$
$tlag0 \mapsto tlag6\ tlbg0 \mapsto tlbg6\ tlar0 \mapsto tlar6\ tlbr0 \mapsto tlbr6$
$plag0 \mapsto plag6\ plar0 \mapsto plar6\ plbg0 \mapsto plbg6\ plbr0 \mapsto plbr6$
$audio0 \mapsto audio6]$
[STATE7 **fit**
$req1 \mapsto req7\ crossing1 \mapsto crossing7\ pressed1 \mapsto pressed7$
$tlag1 \mapsto tlag7\ tlbg1 \mapsto tlbg7\ tlar1 \mapsto tlar7\ tlbr1 \mapsto tlbr7$
$plag1 \mapsto plag7\ plar1 \mapsto plar7\ plbg1 \mapsto plbg7\ plbr1 \mapsto plbr7$
$audio1 \mapsto audio7]$

and $tt \Leftrightarrow true$
$ff \Leftrightarrow false$

then %implies
$(tlag1 \lor tlar1) \land \neg\ (tlag1 \land tlar1) \land (tlbg1 \lor tlbr1)$
$\land \neg\ (tlbg1 \land tlbr1)$
%(T1)%
$(tlag2 \lor tlar2) \land \neg\ (tlag2 \land tlar2) \land (tlbg2 \lor tlbr2)$
$\land \neg\ (tlbg2 \land tlbr2)$
%(T2)%
$(tlag3 \lor tlar3) \land \neg\ (tlag3 \land tlar3) \land (tlbg3 \lor tlbr3)$
$\land \neg\ (tlbg3 \land tlbr3)$
%(T3)%
$(tlag4 \lor tlar4) \land \neg\ (tlag4 \land tlar4) \land (tlbg4 \lor tlbr4)$
$\land \neg\ (tlbg4 \land tlbr4)$
%(T4)%
$(tlag5 \lor tlar5) \land \neg\ (tlag5 \land tlar5) \land (tlbg5 \lor tlbr5)$

$\wedge \neg (tlbg5 \wedge tlbr5)$

%(T5)%

$(tlag6 \vee tlar6) \wedge \neg (tlag6 \wedge tlar6) \wedge (tlbg6 \vee tlbr6)$
$\wedge \neg (tlbg6 \wedge tlbr6)$

%(T6)%

$(\neg (crossing1 \Leftrightarrow crossing0) \vee \neg (req1 \Leftrightarrow req0)$
$\vee \neg (pressed1 \Leftrightarrow pressed0) \vee \neg (tlag1 \Leftrightarrow tlag0))$
$\wedge (\neg (crossing2 \Leftrightarrow crossing0) \vee \neg (req2 \Leftrightarrow req0)$
$\vee \neg (pressed2 \Leftrightarrow pressed0) \vee \neg (tlag2 \Leftrightarrow tlag0))$
$\wedge (\neg (crossing2 \Leftrightarrow crossing1) \vee \neg (req2 \Leftrightarrow req1)$
$\vee \neg (pressed2 \Leftrightarrow pressed1) \vee \neg (tlag2 \Leftrightarrow tlag1))$
$\wedge (\neg (crossing3 \Leftrightarrow crossing0) \vee \neg (req3 \Leftrightarrow req0)$
$\vee \neg (pressed3 \Leftrightarrow pressed0) \vee \neg (tlag3 \Leftrightarrow tlag0))$
$\wedge (\neg (crossing3 \Leftrightarrow crossing1) \vee \neg (req3 \Leftrightarrow req1)$
$\vee \neg (pressed3 \Leftrightarrow pressed1) \vee \neg (tlag3 \Leftrightarrow tlag1))$
$\wedge (\neg (crossing3 \Leftrightarrow crossing2) \vee \neg (req3 \Leftrightarrow req2)$
$\vee \neg (pressed3 \Leftrightarrow pressed2) \vee \neg (tlag3 \Leftrightarrow tlag2))$
$\wedge (\neg (crossing4 \Leftrightarrow crossing0) \vee \neg (req4 \Leftrightarrow req0)$
$\vee \neg (pressed4 \Leftrightarrow pressed0) \vee \neg (tlag4 \Leftrightarrow tlag0))$
$\wedge (\neg (crossing4 \Leftrightarrow crossing1) \vee \neg (req4 \Leftrightarrow req1)$
$\vee \neg (pressed4 \Leftrightarrow pressed1) \vee \neg (tlag4 \Leftrightarrow tlag1))$
$\wedge (\neg (crossing4 \Leftrightarrow crossing2) \vee \neg (req4 \Leftrightarrow req2)$
$\vee \neg (pressed4 \Leftrightarrow pressed2) \vee \neg (tlag4 \Leftrightarrow tlag2))$
$\wedge (\neg (crossing4 \Leftrightarrow crossing3) \vee \neg (req4 \Leftrightarrow req3)$
$\vee \neg (pressed4 \Leftrightarrow pressed3) \vee \neg (tlag4 \Leftrightarrow tlag3))$
$\wedge (\neg (crossing5 \Leftrightarrow crossing0) \vee \neg (req5 \Leftrightarrow req0)$
$\vee \neg (pressed5 \Leftrightarrow pressed0) \vee \neg (tlag5 \Leftrightarrow tlag0))$
$\wedge (\neg (crossing5 \Leftrightarrow crossing1) \vee \neg (req5 \Leftrightarrow req1)$
$\vee \neg (pressed5 \Leftrightarrow pressed1) \vee \neg (tlag5 \Leftrightarrow tlag1))$
$\wedge (\neg (crossing5 \Leftrightarrow crossing2) \vee \neg (req5 \Leftrightarrow req2)$
$\vee \neg (pressed5 \Leftrightarrow pressed2) \vee \neg (tlag5 \Leftrightarrow tlag2))$
$\wedge (\neg (crossing5 \Leftrightarrow crossing3) \vee \neg (req5 \Leftrightarrow req3)$
$\vee \neg (pressed5 \Leftrightarrow pressed3) \vee \neg (tlag5 \Leftrightarrow tlag3))$
$\wedge (\neg (crossing5 \Leftrightarrow crossing4) \vee \neg (req5 \Leftrightarrow req4)$
$\vee \neg (pressed5 \Leftrightarrow pressed4) \vee \neg (tlag5 \Leftrightarrow tlag4))$
$\wedge (\neg (crossing6 \Leftrightarrow crossing0) \vee \neg (req6 \Leftrightarrow req0)$
$\vee \neg (pressed6 \Leftrightarrow pressed0) \vee \neg (tlag6 \Leftrightarrow tlag0))$
$\wedge (\neg (crossing6 \Leftrightarrow crossing1) \vee \neg (req6 \Leftrightarrow req1)$
$\vee \neg (pressed6 \Leftrightarrow pressed1) \vee \neg (tlag6 \Leftrightarrow tlag1))$
$\wedge (\neg (crossing6 \Leftrightarrow crossing2) \vee \neg (req6 \Leftrightarrow req2)$
$\vee \neg (pressed6 \Leftrightarrow pressed2) \vee \neg (tlag6 \Leftrightarrow tlag2))$
$\wedge (\neg (crossing6 \Leftrightarrow crossing3) \vee \neg (req6 \Leftrightarrow req3)$
$\vee \neg (pressed6 \Leftrightarrow pressed3) \vee \neg (tlag6 \Leftrightarrow tlag3))$
$\wedge (\neg (crossing6 \Leftrightarrow crossing4) \vee \neg (req6 \Leftrightarrow req4)$
$\vee \neg (pressed6 \Leftrightarrow pressed4) \vee \neg (tlag6 \Leftrightarrow tlag4))$
$\wedge (\neg (crossing6 \Leftrightarrow crossing5) \vee \neg (req6 \Leftrightarrow req5)$

$$
\begin{aligned}
&\vee \neg\,(\mathit{pressed6} \Leftrightarrow \mathit{pressed5})\\
&\vee \neg\,(\mathit{tlag6} \Leftrightarrow \mathit{tlag5})) \Rightarrow ((\mathit{crossing7} \Leftrightarrow \mathit{crossing6})\\
&\qquad\qquad\qquad\qquad\qquad\quad \wedge (\mathit{req7} \Leftrightarrow \mathit{req6})\\
&\qquad\qquad\qquad\qquad\qquad\quad \wedge (\mathit{pressed7} \Leftrightarrow \mathit{pressed6})\\
&\qquad\qquad\qquad\qquad\qquad\quad \wedge (\mathit{tlag7} \Leftrightarrow \mathit{tlag6}))\\
&\vee ((\mathit{crossing7} \Leftrightarrow \mathit{crossing5})\\
&\quad \wedge (\mathit{req7} \Leftrightarrow \mathit{req5})\\
&\quad \wedge (\mathit{pressed7} \Leftrightarrow \mathit{pressed5})\\
&\quad \wedge (\mathit{tlag7} \Leftrightarrow \mathit{tlag5}))\\
&\vee ((\mathit{crossing7} \Leftrightarrow \mathit{crossing4})\\
&\quad \wedge (\mathit{req7} \Leftrightarrow \mathit{req4})\\
&\quad \wedge (\mathit{pressed7} \Leftrightarrow \mathit{pressed4})\\
&\quad \wedge (\mathit{tlag7} \Leftrightarrow \mathit{tlag4}))\\
&\vee ((\mathit{crossing7} \Leftrightarrow \mathit{crossing3})\\
&\quad \wedge (\mathit{req7} \Leftrightarrow \mathit{req3})\\
&\quad \wedge (\mathit{pressed7} \Leftrightarrow \mathit{pressed3})\\
&\quad \wedge (\mathit{tlag7} \Leftrightarrow \mathit{tlag3}))\\
&\vee ((\mathit{crossing7} \Leftrightarrow \mathit{crossing2})\\
&\quad \wedge (\mathit{req7} \Leftrightarrow \mathit{req2})\\
&\quad \wedge (\mathit{pressed7} \Leftrightarrow \mathit{pressed2})\\
&\quad \wedge (\mathit{tlag7} \Leftrightarrow \mathit{tlag2}))\\
&\vee ((\mathit{crossing7} \Leftrightarrow \mathit{crossing1})\\
&\quad \wedge (\mathit{req7} \Leftrightarrow \mathit{req1})\\
&\quad \wedge (\mathit{pressed7} \Leftrightarrow \mathit{pressed1})\\
&\quad \wedge (\mathit{tlag7} \Leftrightarrow \mathit{tlag1}))
\end{aligned}
$$

%(inclusion)%

end

Appendix C

Forward Reachability with Inclusion Check

library ITERATIVE

logic PROPOSITIONAL

spec STATE0 =
crossing0 req0 tlag0 tlbg0 tlar0 tlbr0 plag0 plbg0 plar0 plbr0 audio0 pressed0
end

spec STATE1 =
crossing1 req1 tlag1 tlbg1 tlar1 tlbr1 plag1 plbg1 plar1 plbr1 audio1 pressed1
end

spec STATE2 =
crossing2 req2 tlag2 tlbg2 tlar2 tlbr2 plag2 plbg2 plar2 plbr2 audio2 pressed2
end

spec STATE3 =
crossing3 req3 tlag3 tlbg3 tlar3 tlbr3 plag3 plbg3 plar3 plbr3 audio3 pressed3
end

spec STATE4 =
crossing4 req4 tlag4 tlbg4 tlar4 tlbr4 plag4 plbg4 plar4 plbr4 audio4 pressed4
end

spec STATE5 =
crossing5 req5 tlag5 tlbg5 tlar5 tlbr5 plag5 plbg5 plar5
plbr5 audio5 pressed5
end

spec STATE6 =
crossing6 req6 tlag6 tlbg6 tlar6 tlbr6 plag6 plbg6 plar6
plbr6 audio6 pressed6
end

spec STATE7 =
crossing7 req7 tlag7 tlbg7 tlar7 tlbr7 plag7 plbg7 plar7
plbr7 audio7 pressed7
end

spec TRANSITION[STATE0][STATE1] =
$crossing1 \Leftrightarrow req0 \wedge \neg\ crossing0$
$req1 \Leftrightarrow pressed0 \wedge \neg\ req0$
$tlag1 \Leftrightarrow \neg\ crossing1 \wedge (\neg\ pressed0 \vee req1)$
$tlbg1 \Leftrightarrow \neg\ crossing1 \wedge (\neg\ pressed0 \vee req1)$
$tlar1 \Leftrightarrow crossing1$
$tlbr1 \Leftrightarrow crossing1$
$plag1 \Leftrightarrow crossing1$
$plbg1 \Leftrightarrow crossing1$
$plar1 \Leftrightarrow \neg\ crossing1$
$plbr1 \Leftrightarrow \neg\ crossing1$
$audio1 \Leftrightarrow crossing1$
end

spec INITIAL[STATE0] =
$\neg\ crossing0$
$\neg\ req0$
$\neg\ pressed0$
$\neg\ tlag0$
$\neg\ tlbg0$
$\neg\ tlar0$
$\neg\ tlbr0$
$\neg\ plag0$
$\neg\ plbg0$
$\neg\ plar0$
$\neg\ plbr0$
$\neg\ audio0$
end

spec SAFE =
tt ff

and STATE0
and STATE1
and STATE2
and STATE3
and STATE4
and STATE5
and STATE6
and STATE7
and INITIAL[STATE0]
then TRANSITION[STATE0][STATE1]
and TRANSITION
[STATE1 **fit**
$req0 \mapsto req1$ $crossing0 \mapsto crossing1$ $pressed0 \mapsto pressed1$
$tlag0 \mapsto tlag1$ $tlbg0 \mapsto tlbg1$ $tlar0 \mapsto tlar1$ $tlbr0 \mapsto tlbr1$
$plag0 \mapsto plag1$ $plar0 \mapsto plar1$ $plbg0 \mapsto plbg1$ $plbr0 \mapsto plbr1$
$audio0 \mapsto audio1$]
[STATE2 **fit**
$req1 \mapsto req2$ $crossing1 \mapsto crossing2$ $pressed1 \mapsto pressed2$
$tlag1 \mapsto tlag2$ $tlbg1 \mapsto tlbg2$ $tlar1 \mapsto tlar2$ $tlbr1 \mapsto tlbr2$
$plag1 \mapsto plag2$ $plar1 \mapsto plar2$ $plbg1 \mapsto plbg2$ $plbr1 \mapsto plbr2$
$audio1 \mapsto audio2$]
and TRANSITION
[STATE2 **fit**
$req0 \mapsto req2$ $crossing0 \mapsto crossing2$ $pressed0 \mapsto pressed2$
$tlag0 \mapsto tlag2$ $tlbg0 \mapsto tlbg2$ $tlar0 \mapsto tlar2$ $tlbr0 \mapsto tlbr2$
$plag0 \mapsto plag2$ $plar0 \mapsto plar2$ $plbg0 \mapsto plbg2$ $plbr0 \mapsto plbr2$
$audio0 \mapsto audio2$]
[STATE3 **fit**
$req1 \mapsto req3$ $crossing1 \mapsto crossing3$ $pressed1 \mapsto pressed3$
$tlag1 \mapsto tlag3$ $tlbg1 \mapsto tlbg3$ $tlar1 \mapsto tlar3$ $tlbr1 \mapsto tlbr3$
$plag1 \mapsto plag3$ $plar1 \mapsto plar3$ $plbg1 \mapsto plbg3$ $plbr1 \mapsto plbr3$
$audio1 \mapsto audio3$]
and TRANSITION
[STATE3 **fit**
$req0 \mapsto req3$ $crossing0 \mapsto crossing3$ $pressed0 \mapsto pressed3$
$tlag0 \mapsto tlag3$ $tlbg0 \mapsto tlbg3$ $tlar0 \mapsto tlar3$ $tlbr0 \mapsto tlbr3$
$plag0 \mapsto plag3$ $plar0 \mapsto plar3$ $plbg0 \mapsto plbg3$ $plbr0 \mapsto plbr3$
$audio0 \mapsto audio3$]
[STATE4 **fit**
$req1 \mapsto req4$ $crossing1 \mapsto crossing4$ $pressed1 \mapsto pressed4$
$tlag1 \mapsto tlag4$ $tlbg1 \mapsto tlbg4$ $tlar1 \mapsto tlar4$ $tlbr1 \mapsto tlbr4$
$plag1 \mapsto plag4$ $plar1 \mapsto plar4$ $plbg1 \mapsto plbg4$ $plbr1 \mapsto plbr4$
$audio1 \mapsto audio4$]
and TRANSITION
[STATE4 **fit**
$req0 \mapsto req4$ $crossing0 \mapsto crossing4$ $pressed0 \mapsto pressed4$

tlag0 ↦ *tlag4* *tlbg0* ↦ *tlbg4* *tlar0* ↦ *tlar4* *tlbr0* ↦ *tlbr4*
plag0 ↦ *plag4* *plar0* ↦ *plar4* *plbg0* ↦ *plbg4* *plbr0* ↦ *plbr4*
audio0 ↦ *audio4*]
[State5 **fit**
req1 ↦ *req5* *crossing1* ↦ *crossing5* *pressed1* ↦ *pressed5*
tlag1 ↦ *tlag5* *tlbg1* ↦ *tlbg5* *tlar1* ↦ *tlar5* *tlbr1* ↦ *tlbr5*
plag1 ↦ *plag5* *plar1* ↦ *plar5* *plbg1* ↦ *plbg5* *plbr1* ↦ *plbr5*
audio1 ↦ *audio5*]
and Transition
[State5 **fit**
req0 ↦ *req5* *crossing0* ↦ *crossing5* *pressed0* ↦ *pressed5*
tlag0 ↦ *tlag5* *tlbg0* ↦ *tlbg5* *tlar0* ↦ *tlar5* *tlbr0* ↦ *tlbr5*
plag0 ↦ *plag5* *plar0* ↦ *plar5* *plbg0* ↦ *plbg5* *plbr0* ↦ *plbr5*
audio0 ↦ *audio5*]
[State6 **fit**
req1 ↦ *req6* *crossing1* ↦ *crossing6* *pressed1* ↦ *pressed6*
tlag1 ↦ *tlag6* *tlbg1* ↦ *tlbg6* *tlar1* ↦ *tlar6* *tlbr1* ↦ *tlbr6*
plag1 ↦ *plag6* *plar1* ↦ *plar6* *plbg1* ↦ *plbg6* *plbr1* ↦ *plbr6*
audio1 ↦ *audio6*]
and Transition
[State6 **fit**
req0 ↦ *req6* *crossing0* ↦ *crossing6* *pressed0* ↦ *pressed6*
tlag0 ↦ *tlag6* *tlbg0* ↦ *tlbg6* *tlar0* ↦ *tlar6* *tlbr0* ↦ *tlbr6*
plag0 ↦ *plag6* *plar0* ↦ *plar6* *plbg0* ↦ *plbg6* *plbr0* ↦ *plbr6*
audio0 ↦ *audio6*]
[State7 **fit**
req1 ↦ *req7* *crossing1* ↦ *crossing7* *pressed1* ↦ *pressed7*
tlag1 ↦ *tlag7* *tlbg1* ↦ *tlbg7* *tlar1* ↦ *tlar7* *tlbr1* ↦ *tlbr7*
plag1 ↦ *plag7* *plar1* ↦ *plar7* *plbg1* ↦ *plbg7* *plbr1* ↦ *plbr7*
audio1 ↦ *audio7*]
and *tt* ⇔ *true*
ff ⇔ *false*
then %implies
$(tlag1 \lor tlar1) \land \neg (tlag1 \land tlar1) \land (tlbg1 \lor tlbr1)$
$\land \neg (tlbg1 \land tlbr1)$
%(T1)%
$(tlag2 \lor tlar2) \land \neg (tlag2 \land tlar2) \land (tlbg2 \lor tlbr2)$
$\land \neg (tlbg2 \land tlbr2)$
%(T2)%
$(tlag3 \lor tlar3) \land \neg (tlag3 \land tlar3) \land (tlbg3 \lor tlbr3)$
$\land \neg (tlbg3 \land tlbr3)$
%(T3)%
$(tlag4 \lor tlar4) \land \neg (tlag4 \land tlar4) \land (tlbg4 \lor tlbr4)$
$\land \neg (tlbg4 \land tlbr4)$
%(T4)%
$(tlag5 \lor tlar5) \land \neg (tlag5 \land tlar5) \land (tlbg5 \lor tlbr5)$

$\wedge \neg (\mathit{tlbg5} \wedge \mathit{tlbr5})$

%(T5)%

$(\mathit{tlag6} \vee \mathit{tlar6}) \wedge \neg (\mathit{tlag6} \wedge \mathit{tlar6}) \wedge (\mathit{tlbg6} \vee \mathit{tlbr6})$
$\wedge \neg (\mathit{tlbg6} \wedge \mathit{tlbr6})$

%(T6)%

$(\neg (\mathit{crossing1} \Leftrightarrow \mathit{crossing0}) \vee \neg (\mathit{req1} \Leftrightarrow \mathit{req0})$
$\vee \neg (\mathit{pressed1} \Leftrightarrow \mathit{pressed0}) \vee \neg (\mathit{tlag1} \Leftrightarrow \mathit{tlag0}))$
$\wedge (\neg (\mathit{crossing2} \Leftrightarrow \mathit{crossing0}) \vee \neg (\mathit{req2} \Leftrightarrow \mathit{req0})$
$\vee \neg (\mathit{pressed2} \Leftrightarrow \mathit{pressed0}) \vee \neg (\mathit{tlag2} \Leftrightarrow \mathit{tlag0}))$
$\wedge (\neg (\mathit{crossing2} \Leftrightarrow \mathit{crossing1}) \vee \neg (\mathit{req2} \Leftrightarrow \mathit{req1})$
$\vee \neg (\mathit{pressed2} \Leftrightarrow \mathit{pressed1}) \vee \neg (\mathit{tlag2} \Leftrightarrow \mathit{tlag1}))$
$\wedge (\neg (\mathit{crossing3} \Leftrightarrow \mathit{crossing0}) \vee \neg (\mathit{req3} \Leftrightarrow \mathit{req0})$
$\vee \neg (\mathit{pressed3} \Leftrightarrow \mathit{pressed0}) \vee \neg (\mathit{tlag3} \Leftrightarrow \mathit{tlag0}))$
$\wedge (\neg (\mathit{crossing3} \Leftrightarrow \mathit{crossing1}) \vee \neg (\mathit{req3} \Leftrightarrow \mathit{req1})$
$\vee \neg (\mathit{pressed3} \Leftrightarrow \mathit{pressed1}) \vee \neg (\mathit{tlag3} \Leftrightarrow \mathit{tlag1}))$
$\wedge (\neg (\mathit{crossing3} \Leftrightarrow \mathit{crossing2}) \vee \neg (\mathit{req3} \Leftrightarrow \mathit{req2})$
$\vee \neg (\mathit{pressed3} \Leftrightarrow \mathit{pressed2}) \vee \neg (\mathit{tlag3} \Leftrightarrow \mathit{tlag2}))$
$\wedge (\neg (\mathit{crossing4} \Leftrightarrow \mathit{crossing0}) \vee \neg (\mathit{req4} \Leftrightarrow \mathit{req0})$
$\vee \neg (\mathit{pressed4} \Leftrightarrow \mathit{pressed0}) \vee \neg (\mathit{tlag4} \Leftrightarrow \mathit{tlag0}))$
$\wedge (\neg (\mathit{crossing4} \Leftrightarrow \mathit{crossing1}) \vee \neg (\mathit{req4} \Leftrightarrow \mathit{req1})$
$\vee \neg (\mathit{pressed4} \Leftrightarrow \mathit{pressed1}) \vee \neg (\mathit{tlag4} \Leftrightarrow \mathit{tlag1}))$
$\wedge (\neg (\mathit{crossing4} \Leftrightarrow \mathit{crossing2}) \vee \neg (\mathit{req4} \Leftrightarrow \mathit{req2})$
$\vee \neg (\mathit{pressed4} \Leftrightarrow \mathit{pressed2}) \vee \neg (\mathit{tlag4} \Leftrightarrow \mathit{tlag2}))$
$\wedge (\neg (\mathit{crossing4} \Leftrightarrow \mathit{crossing3}) \vee \neg (\mathit{req4} \Leftrightarrow \mathit{req3})$
$\vee \neg (\mathit{pressed4} \Leftrightarrow \mathit{pressed3}) \vee \neg (\mathit{tlag4} \Leftrightarrow \mathit{tlag3}))$
$\wedge (\neg (\mathit{crossing5} \Leftrightarrow \mathit{crossing0}) \vee \neg (\mathit{req5} \Leftrightarrow \mathit{req0})$
$\vee \neg (\mathit{pressed5} \Leftrightarrow \mathit{pressed0}) \vee \neg (\mathit{tlag5} \Leftrightarrow \mathit{tlag0}))$
$\wedge (\neg (\mathit{crossing5} \Leftrightarrow \mathit{crossing1}) \vee \neg (\mathit{req5} \Leftrightarrow \mathit{req1})$
$\vee \neg (\mathit{pressed5} \Leftrightarrow \mathit{pressed1}) \vee \neg (\mathit{tlag5} \Leftrightarrow \mathit{tlag1}))$
$\wedge (\neg (\mathit{crossing5} \Leftrightarrow \mathit{crossing2}) \vee \neg (\mathit{req5} \Leftrightarrow \mathit{req2})$
$\vee \neg (\mathit{pressed5} \Leftrightarrow \mathit{pressed2}) \vee \neg (\mathit{tlag5} \Leftrightarrow \mathit{tlag2}))$
$\wedge (\neg (\mathit{crossing5} \Leftrightarrow \mathit{crossing3}) \vee \neg (\mathit{req5} \Leftrightarrow \mathit{req3})$
$\vee \neg (\mathit{pressed5} \Leftrightarrow \mathit{pressed3}) \vee \neg (\mathit{tlag5} \Leftrightarrow \mathit{tlag3}))$
$\wedge (\neg (\mathit{crossing5} \Leftrightarrow \mathit{crossing4}) \vee \neg (\mathit{req5} \Leftrightarrow \mathit{req4})$
$\vee \neg (\mathit{pressed5} \Leftrightarrow \mathit{pressed4}) \vee \neg (\mathit{tlag5} \Leftrightarrow \mathit{tlag4}))$
$\wedge (\neg (\mathit{crossing6} \Leftrightarrow \mathit{crossing0}) \vee \neg (\mathit{req6} \Leftrightarrow \mathit{req0})$
$\vee \neg (\mathit{pressed6} \Leftrightarrow \mathit{pressed0}) \vee \neg (\mathit{tlag6} \Leftrightarrow \mathit{tlag0}))$
$\wedge (\neg (\mathit{crossing6} \Leftrightarrow \mathit{crossing1}) \vee \neg (\mathit{req6} \Leftrightarrow \mathit{req1})$
$\vee \neg (\mathit{pressed6} \Leftrightarrow \mathit{pressed1}) \vee \neg (\mathit{tlag6} \Leftrightarrow \mathit{tlag1}))$
$\wedge (\neg (\mathit{crossing6} \Leftrightarrow \mathit{crossing2}) \vee \neg (\mathit{req6} \Leftrightarrow \mathit{req2})$
$\vee \neg (\mathit{pressed6} \Leftrightarrow \mathit{pressed2}) \vee \neg (\mathit{tlag6} \Leftrightarrow \mathit{tlag2}))$
$\wedge (\neg (\mathit{crossing6} \Leftrightarrow \mathit{crossing3}) \vee \neg (\mathit{req6} \Leftrightarrow \mathit{req3})$
$\vee \neg (\mathit{pressed6} \Leftrightarrow \mathit{pressed3}) \vee \neg (\mathit{tlag6} \Leftrightarrow \mathit{tlag3}))$
$\wedge (\neg (\mathit{crossing6} \Leftrightarrow \mathit{crossing4}) \vee \neg (\mathit{req6} \Leftrightarrow \mathit{req4})$
$\vee \neg (\mathit{pressed6} \Leftrightarrow \mathit{pressed4}) \vee \neg (\mathit{tlag6} \Leftrightarrow \mathit{tlag4}))$
$\wedge (\neg (\mathit{crossing6} \Leftrightarrow \mathit{crossing5}) \vee \neg (\mathit{req6} \Leftrightarrow \mathit{req5})$

$$
\begin{array}{l}
\vee \neg\, (\mathit{pressed6} \Leftrightarrow \mathit{pressed5}) \\
\vee \neg\, (\mathit{tlag6} \Leftrightarrow \mathit{tlag5})) \Rightarrow ((\mathit{crossing7} \Leftrightarrow \mathit{crossing6}) \\
\qquad\qquad\qquad\qquad\qquad \wedge\, (\mathit{req7} \Leftrightarrow \mathit{req6}) \\
\qquad\qquad\qquad\qquad\qquad \wedge\, (\mathit{pressed7} \Leftrightarrow \mathit{pressed6}) \\
\qquad\qquad\qquad\qquad\qquad \wedge\, (\mathit{tlag7} \Leftrightarrow \mathit{tlag6})) \\
\vee\, ((\mathit{crossing7} \Leftrightarrow \mathit{crossing5}) \\
\quad \wedge\, (\mathit{req7} \Leftrightarrow \mathit{req5}) \\
\quad \wedge\, (\mathit{pressed7} \Leftrightarrow \mathit{pressed5}) \\
\quad \wedge\, (\mathit{tlag7} \Leftrightarrow \mathit{tlag5})) \\
\vee\, ((\mathit{crossing7} \Leftrightarrow \mathit{crossing4}) \\
\quad \wedge\, (\mathit{req7} \Leftrightarrow \mathit{req4}) \\
\quad \wedge\, (\mathit{pressed7} \Leftrightarrow \mathit{pressed4}) \\
\quad \wedge\, (\mathit{tlag7} \Leftrightarrow \mathit{tlag4})) \\
\vee\, ((\mathit{crossing7} \Leftrightarrow \mathit{crossing3}) \\
\quad \wedge\, (\mathit{req7} \Leftrightarrow \mathit{req3}) \\
\quad \wedge\, (\mathit{pressed7} \Leftrightarrow \mathit{pressed3}) \\
\quad \wedge\, (\mathit{tlag7} \Leftrightarrow \mathit{tlag3})) \\
\vee\, ((\mathit{crossing7} \Leftrightarrow \mathit{crossing2}) \\
\quad \wedge\, (\mathit{req7} \Leftrightarrow \mathit{req2}) \\
\quad \wedge\, (\mathit{pressed7} \Leftrightarrow \mathit{pressed2}) \\
\quad \wedge\, (\mathit{tlag7} \Leftrightarrow \mathit{tlag2})) \\
\vee\, ((\mathit{crossing7} \Leftrightarrow \mathit{crossing1}) \\
\quad \wedge\, (\mathit{req7} \Leftrightarrow \mathit{req1}) \\
\quad \wedge\, (\mathit{pressed7} \Leftrightarrow \mathit{pressed1}) \\
\quad \wedge\, (\mathit{tlag7} \Leftrightarrow \mathit{tlag1}))
\end{array}
$$

%(inclusion)%

end

Appendix D

Temporal Induction

library ITERATIVE

logic PROPOSITIONAL

spec STATE0 =
  crossing0 req0 tlag0 tlbg0 tlar0 tlbr0 plag0 plbg0 plar0 plbr0 audio0 pressed0
end

spec STATE1 =
  crossing1 req1 tlag1 tlbg1 tlar1 tlbr1 plag1 plbg1 plar1 plbr1 audio1 pressed1
end

spec STATE2 =
  crossing2 req2 tlag2 tlbg2 tlar2 tlbr2 plag2 plbg2 plar2 plbr2 audio2 pressed2
end

spec STATE3 =
  crossing3 req3 tlag3 tlbg3 tlar3 tlbr3 plag3 plbg3 plar3 plbr3 audio3 pressed3
end

spec STATE4 =
  crossing4 req4 tlag4 tlbg4 tlar4 tlbr4 plag4 plbg4 plar4 plbr4 audio4 pressed4
end

spec STATE5 =
  crossing5 req5 tlag5 tlbg5 tlar5 tlbr5 plag5 plbg5 plar5

```
        plbr5 audio5 pressed5
end

spec State6 =
        crossing6 req6 tlag6 tlbg6 tlar6 tlbr6 plag6 plbg6 plar6
        plbr6 audio6 pressed6
end

spec State7 =
        crossing7 req7 tlag7 tlbg7 tlar7 tlbr7 plag7 plbg7 plar7
        plbr7 audio7 pressed7
end

spec Transition[State0][State1] =
        crossing1 ⇔ req0 ∧ ¬ crossing0
        req1 ⇔ pressed0 ∧ ¬ req0
        tlag1 ⇔ ¬ crossing1 ∧ (¬ pressed0 ∨ req1)
        tlbg1 ⇔ ¬ crossing1 ∧ (¬ pressed0 ∨ req1)
        tlar1 ⇔ crossing1
        tlbr1 ⇔ crossing1
        plag1 ⇔ crossing1
        plbg1 ⇔ crossing1
        plar1 ⇔ ¬ crossing1
        plbr1 ⇔ ¬ crossing1
        audio1 ⇔ crossing1
end

spec Initial[State0] =
        ¬ crossing0
        ¬ req0
        ¬ pressed0
        ¬ tlag0
        ¬ tlbg0
        ¬ tlar0
        ¬ tlbr0
        ¬ plag0
        ¬ plbg0
        ¬ plar0
        ¬ plbr0
        ¬ audio0
end

spec base =
        State0
and     State1
and     Initial[State0]
```

then TRANSITION[STATE0][STATE1]
then **%implies**
$(tlag1 \lor tlar1) \land \neg\, (tlag1 \land tlar1) \land (tlbg1 \lor tlbr1)$
$\land \neg\, (tlbg1 \land tlbr1)$
%(Base)%
end

spec STEP =
STATE0
and STATE1
and STATE2
then TRANSITION[STATE0][STATE1]
and TRANSITION
[STATE1 **fit**
$req0 \mapsto req1$ $crossing0 \mapsto crossing1$ $pressed0 \mapsto pressed1$
$tlag0 \mapsto tlag1$ $tlbg0 \mapsto tlbg1$ $tlar0 \mapsto tlar1$ $tlbr0 \mapsto tlbr1$
$plag0 \mapsto plag1$ $plar0 \mapsto plar1$ $plbg0 \mapsto plbg1$ $plbr0 \mapsto plbr1$
$audio0 \mapsto audio1$]
[STATE2 **fit**
$req1 \mapsto req2$ $crossing1 \mapsto crossing2$ $pressed1 \mapsto pressed2$
$tlag1 \mapsto tlag2$ $tlbg1 \mapsto tlbg2$ $tlar1 \mapsto tlar2$ $tlbr1 \mapsto tlbr2$
$plag1 \mapsto plag2$ $plar1 \mapsto plar2$ $plbg1 \mapsto plbg2$ $plbr1 \mapsto plbr2$
$audio1 \mapsto audio2$]
and $(\neg\, (crossing1 \Leftrightarrow crossing0) \lor \neg\, (req1 \Leftrightarrow req0)$
$\lor \neg\, (pressed1 \Leftrightarrow pressed0) \lor \neg\, (tlag1 \Leftrightarrow tlag0))$
$\land (\neg\, (crossing2 \Leftrightarrow crossing0) \lor \neg\, (req2 \Leftrightarrow req0)$
$\lor \neg\, (pressed2 \Leftrightarrow pressed0) \lor \neg\, (tlag2 \Leftrightarrow tlag0))$
$\land (\neg\, (crossing2 \Leftrightarrow crossing1) \lor \neg\, (req2 \Leftrightarrow req1)$
$\lor \neg\, (pressed2 \Leftrightarrow pressed1) \lor \neg\, (tlag2 \Leftrightarrow tlag1))$
and $(tlag1 \lor tlar1) \land \neg\, (tlag1 \land tlar1)$
$\land (tlbg1 \lor tlbr1) \land \neg\, (tlbg1 \land tlbr1)$
then **%implies**
$(tlag2 \lor tlar2) \land \neg\, (tlag2 \land tlar2) \land (tlbg2 \lor tlbr2)$
$\land \neg\, (tlbg2 \land tlbr2)$
%(Step)%
end