Electronic Communications of the EASST
Volume 35 (2010)

Proceedings of the 10th International Workshop on Automated Verification of Critical Systems (AVoCS 2010)

Automatically Verifying Railway Interlockings using SAT-based Model Checking

Phillip James and Markus Roggenbach

17 pages

Guest Editors: Jens Bendisposto, Michael Leuschel, Markus Roggenbach
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Automatically Verifying Railway Interlockings using SAT-based Model Checking

Phillip James[1]* **and Markus Roggenbach**[1]

Swansea University[1], Wales, UK

Abstract: In this paper, 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, we show how execution of the system can be modelled via a finite automaton. We give algorithms to perform SAT-based model checking over such an automaton. In order to tackle state-space explosion we propose slicing. Finally we comment on results obtained by applying these methods to verify two real-world railway interlocking systems.

Keywords: Model Checking, Interlocking, Ladder Logic, Railway, SAT, Slicing.

1 Introduction

Formal verification of railway control software has been identified to be one of the "grand challenges" [Jac04] of Computer Science. Various formal methods have been applied to this area, including algebraic specification, e.g., [Bjø09], process-algebraic modelling and verification, e.g., [Win02], and also model-oriented specification, where e.g., the B method has been used in order to verify part of the Paris Metro railway [BG00]. In partnership with Invensys, an internationally established company specialising in railway control systems, we explore various verification approaches based on SAT solving [BHMW09]. The aim is to explore and develop technologies that, at a later date, might be integrated into Invensys' design process.

Continuing work by Kanso et al. [KMS08] we verify interlockings of real-world train stations with respect to safety conditions. Our modelling language is propositional logic, see Figure 1: The physical layout of the train station together with an abstract safety condition, e.g., 'trains are separated by at least one empty track segment', yields a concrete safety condition φ. The initial configuration of a train station is characterised by some initialisation formula I. The control program (in ladder logic, an IEC standard [IEC03]) of the interlocking system is translated into a transition formula T. All the above translations have been automated in [KMS08]. Using an inductive approach, namely $I(Z) \Rightarrow \varphi(Z)$ and $T(Z,Z') \wedge \varphi(Z) \Rightarrow \varphi(Z')$, Kanso et al. [KMS08] successfully verify a medium sized real-world interlocking. Some of the required safety properties are automatically proven using a SAT solver [Kul08], however in some cases the SAT solver produces counter examples. These take the from of a pair of states, namely interpretations of Z and Z', which violate the safety property. In the context of the interlocking under discussion, these counter examples were excluded via manual analysis: it was claimed that they concern unreachable states. For inclusion into the standard development process of interlockings, Invensys requires further automation of the verification, namely the exclusion of the supposed to be

* Acknowledging the support of Invensys Rail.

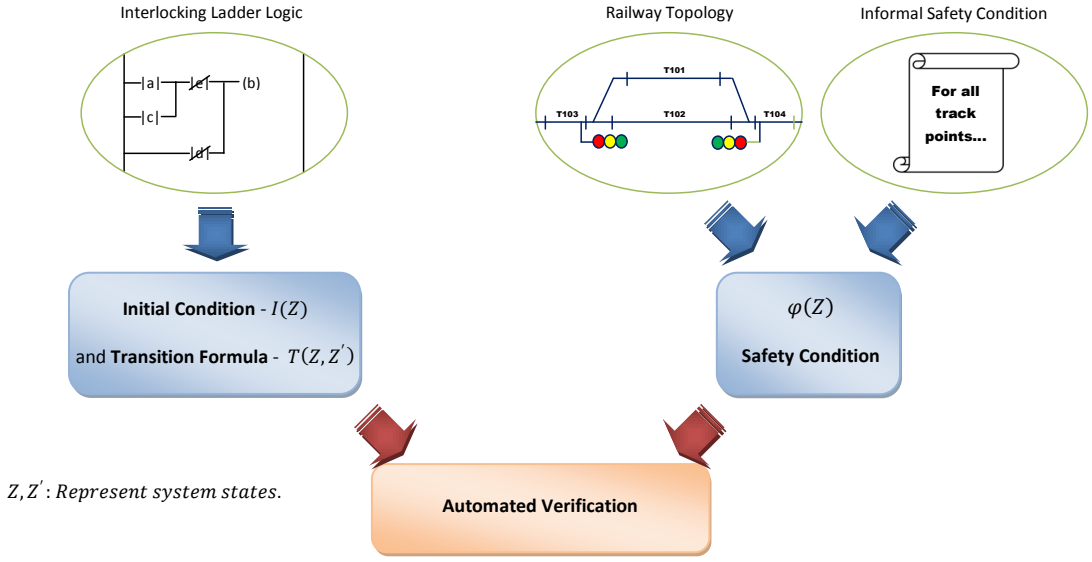

Figure 1: The basic verification setting.

unreachable states and the production of error traces if a safety property does not hold.

In order to accommodate these requirements, we develop and experiment with verification approaches based on ideas used in bounded model checking. Here, we deliberately stay within Boolean modelling: first, it is natural in the given context – the ladder logic program contains only Boolean variables; second, it allows the direct use of SAT solvers for verification.

In order to deal with real-world interlockings, we develop a slicing technique. To this end we re-use an algorithm first stated by [GKV95, FH98] and prove that it is correct w.r.t. our specific setting. In practice, slicing reduces the problem size by approximately a factor of five. This reduction has proven to be enough to automatically verify, using various techniques, two interlockings of medium complexity: either the safety condition could be proven, or an error trace was produced.

In [ZRK03, FH98] alternative approaches for the verification of ladder logic programs are provided. In [ZRK03] a translation from ladder logic into timed automata is defined, before using the Uppaal model checker [upp10] for verification. Due to state-space explosion their approach is limited to "small" programs. Secondly, in [FH98] an inductive verification approach is taken to verify ladder logic interlockings.

This paper is organised as follows: In Section 2, we introduce the basics of railway interlockings. Section 3 introduces a pelican crossing as a small example system. In Sections 4 and 5 we give a modelling of interlockings through propositional logic and automata. Section 6 introduces the model checking approaches we apply, with Section 7 giving a method to tackle state-space explosion. Finally, Section 8 shows the results gained from verifying two real-world interlockings. The results given in this paper are based on [Jam10] and have already been presented at CALCO-Jnr [JR10], a workshop for young researchers which encourages re-submission of papers to proper scientific events.

2 Interlockings

An interlocking provides a safety layer for a railway. It interfaces with both the physical track layout and the human (or computerised) controller. The controller issues requests, such as to move a point. On such a request the interlocking will determine whether it is safe for the operation to be permitted. If it is safe then the interlocking will issue requests to change the physical track layout, informing the controller of the change. Whereas if it is unsafe to perform the operation the interlocking will not allow the physical track layout to be changed, and will report back to the controller that the operation has not taken place as it would yield an unsafe situation.

Here, we consider Westrace [wes10] interlockings. A Westrace interlocking has the following typical control flow:

```
initialise
while True do
    read (Input)                          %% read
(*) State' <- Program(Input, State)       %% process
    write (Output) & State <- State'      %% update
```

After initialisation, there is a non terminating loop consisting of three steps: (1) Reading of `Input`, where `Input` includes requests from signallers and data from physical track sensors. (2) Internal processing: this depends on the `Input` as well as on the current `State` of the controller. Using these the next state `State'` is computed. (3) Committing of `Output`, which includes passing information back to the signaller, commands to change the physical track layout, as well as an update of the `State` of the controller.

In the context of Westrace interlockings, `Input`, `Output`, `State`, and `State'` are sets of Boolean variables, where `Output` is a subset of `State'`. The current *configuration* of the controller is given by the values of all variables in the sets `Input` and `State`. The `process` step then depends on the current *configuration*. The Westrace interlocking realises this controller in hardware (cycle time of approximately 1 sec), where the steps `initialise` and `process` depend on the installed control software written in ladder logic – see Section 4.

The `initialise` step performs the following:

```
set_to_false (Input)
State' <- Program(Input, State)
State <- State'
```

First, all `Input` variables are set to false, then the `process` step is executed once, finally `State` is updated.

3 Pelican crossing example

As a running example we study a pelican crossing. Such a system is 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 pedestrian 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 *pla_g*, *pla_r*, *plb_g*, *plb_r*, *tla_g*, *tla_r*, *tlb_g*, and *tlb_r*. When one of these variables is true, the corresponding light is on. There is also 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.

$$
\begin{aligned}
&[\ crossing' \iff (req \wedge \neg crossing),\\
&\ \ req' \iff (pressed \wedge \neg req),\\
&\ \ tla\_g' \iff ((\neg crossing') \wedge (\neg pressed \vee req')),\\
&\ \ tlb\_g' \iff ((\neg crossing') \wedge (\neg pressed \vee req')),\\
&\ \ tla\_r' \iff crossing',\\
&\ \ tlb\_r' \iff crossing',\\
&\ \ pla\_g' \iff crossing',\\
&\ \ plb\_g' \iff crossing',\\
&\ \ pla\_r' \iff (\neg crossing'),\\
&\ \ plb\_r' \iff (\neg crossing'),\\
&\ \ audio' \iff crossing'\ ]
\end{aligned}
$$

Figure 2: A ladder logic formula for the control program of a pelican crossing.

Figure 2 presents the control program of our pelican crossing. It uses unprimed variables to store the configuration of the controller *before* the `process` step. Primed variables store the values of state variables *after* the `process` step. We can also say: if the unprimed variables represent the configuration at `(⋆)`, then the primed variables represent the configuration at `(⋆)` in the next cycle of the control loop.

As an example of how to interpret the control program, consider the first line of Figure 2, namely "$crossing' \iff (req \wedge \neg crossing)$". This can be read as: if there was a request *req* in the last control cycle, and pedestrians were not able to cross the road, then at the end of the current cycle pedestrians will be able 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.

4 Ladder logic formulae

Ladder logic is a graphical programming language specified in the IEC standard 61131 [IEC03]. Westrace interlockings are programmed with ladder logic. A ladder logic program can be equivalently translated into a subset of propositional logic. We call this subset ladder logic formulae

(see below for its definition). This translation is straightforward: it replaces graphical symbols by logical operators, a process which has been automated in [Kan08][1]. For the rest of the paper we only deal with this representation in propositional logic. Figure 2 gives a concrete instance using a practical shorthand notation.

Ladder logic formulae have several underlying syntactical restrictions. These restrictions become important later for slicing. In order to describe their syntax we use the following notations: The function *vars* returns for a given propositional formula φ the set of propositional variables appearing in φ. We use "prime" to generate a fresh variable. $V' = \{v' \mid v \in V\}$ denotes the set of all fresh variables obtained from a set of variables V.

A ladder logic program is formulated relatively to a finite set of input variables I and a finite set of state variables C, such that $I \cap C = \emptyset$. It may also refer to primed state variables C', which represent the newly computed values within a control cycle.

Definition 1 (Ladder logic formulae) A ladder logic formula ψ (relative to a set of input variables I and a set of state variables C) is a propositional formula

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

where $n \geq 0$ and the ψ_i, $1 \leq i \leq n$, are propositional formulae, such that the following conditions hold:

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

If $n = 0$, as usual $\psi \equiv True$. Such an empty program proves useful in the context of slicing.

A ladder logic program prescribes the computation that takes place in the `process` step of the control loop. The equivalence ($\Leftrightarrow$) can be interpreted as assignment. The above conditions ensure that only primed state variables can be assigned to; a primed state variable is assigned to at most once; a primed state variable can only depend on input variables, primed state variables, or state variables. Here either the unprimed or the primed version of a state variable can be used, depending on the index i.

For a ladder logic formula we often write $\psi \equiv [R_1, R_2, \ldots, R_n]$ where $R_i \equiv c'_i \Leftrightarrow \psi_i$, for $1 \leq i \leq n$, for some $n \geq 0$. The subformulae R_i are called rungs.

5 Representation of an interlocking as an automaton

We capture the dynamics of a Westrace interlocking by defining an automaton relative to a given ladder logic formula. Consider the control loop in Section 2: a state in the automaton represents a configuration of the controller and a transition $p \to q$ represents one execution of the loop. That is, if p represents the configuration of the controller at `(*)`, then q represents the configuration of the controller at `(*)` one cycle later.

[1] A similar modelling approach has been taken in [FH98].

In order to define the transition relation via ladder logic formulae, we define paired valuations. In the definition we use I' to represent new inputs to the controller and the function *unprime* to remove the prime from a variable.

Definition 2 (Paired valuations) Given a finite set of input variables I, a finite set of state variables C, and valuations $\mu, \mu' : (I \cup C) \to \{0,1\}$ we define the paired valuation $\mu \wr \mu' : (I \cup C \cup I' \cup C') \to \{0,1\}$ where

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

We now define an automaton for a ladder logic formula:

Definition 3 (Automaton) Given a ladder logic formula ψ over $I \cup C$, we define the automaton

$$A(\psi) = (S, S_0, \to)$$

where

- $S = \{\mu \,|\, \mu : I \cup C \to \{0,1\}\}$ is the set of states,
- $\mu \to \mu'$ if $\mu \wr \mu' \models \psi$ defines the transitions, and
- $S_0 = \{\mu' \,|\, \exists \mu : \mu \models \neg I, \mu \wr \mu' \models \psi\}$ gives the set of initial states.
  Here, $\neg I$ expands to $\bigwedge_{i \in I} \neg i$ for all $i \in I$.

Remark 1 *The automaton $A(\psi)$ is non deterministic as ψ does not impose any conditions on the variables in I': The controller is not allowed to refuse any input. The automaton might have more than one start state as the computation of the set of initial states only sets the input variables I, the state variables C can take any value. Finally, the automaton $A(\psi)$ is* finite*; it has $2^{|I \cup C|}$ states.*

This automaton faithfully models the behaviour of the interlocking. The set of initial states S_0 of the automaton represents all possible configurations of the interlocking when reaching point `(*)` for the first time. As one transition corresponds to one execution of the loop, the traces of configurations observed at `(*)` directly correspond to the state sequences of the automaton.

Naturally, such a controller should never stop. In our formalisation of a Westrace interlocking we can prove this:

Theorem 1 *Let ψ be a ladder logic formula. Let μ be a state in $A(\psi)$. Then there exists a state μ' such that $\mu \wr \mu' \models \psi$, i.e. it holds that $\mu \to \mu'$.*

Proof. (Sketch) By induction on size n of a ladder logic formula. Assume the claim holds for length i. Given an evaluation μ_i for $V_i = I \cup \{c'_1, \ldots, c'_{i-1}\} \cup \{c_i, \ldots, c_n\}$ we set $\mu_{i+1}(x) = \mu_i(x)$ for $x \in V_i$, $\mu_{i+1}(c'_i) = 1$ if $\mu_i \models \psi_i$ and $\mu_{i+1}(c'_i) = 0$ if $\mu_i \not\models \psi_i$. Finally set $\mu'(c) = \mu_n(c)$ for all $c \in C$. □

A paired valuation $\mu \wr \mu'$ is reachable with respect to an automaton $A(\psi) = (S, S_0, \to)$, if there exists a series of transitions $\mu_0 \to \mu_1 \to \cdots \to \mu \to \mu'$ with $\mu_0 \in S_0$.

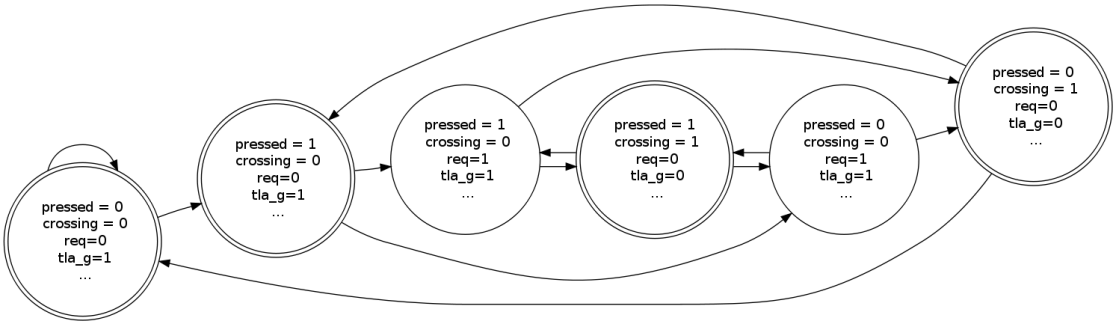

Figure 3: An automaton modelling of the ladder logic program for a pelican crossing.

Figure 3 illustrates the reachable states of the automaton constructed from the pelican crossing ladder logic formula in Figure 2. Here, initial states are represented via double circles, and some variable values have been excluded for ease of reading.

5.1 Safety conditions

A typical safety property in our pelican crossing example would be: "A traffic light always shows a single aspect". Using the vocabulary for the control program, we capture this property by the following propositional formula:

$$SingleAspect \equiv (tla\_g \vee tla\_r) \wedge \neg(tla\_g \wedge tla\_r) \wedge (tlb\_g \vee tlb\_r) \wedge \neg(tlb\_g \wedge tlb\_r).$$

I.e., "For both traffic lights, namely *tla* and *tlb*, it holds that they always show a signal, however, they never show green and red at the same time."

Experience with Westrace interlockings has shown that the safety properties arising in practice speak about at most two consecutive configurations at $(\star)$ of the control program depicted in Section 2 (here the above example speaks only about one configuration). This justifies the following definition:

Definition 4 (Safety condition) A safety condition φ for a ladder logic formula ψ over variables $I \cup C$ is a propositional formula over variables $I \cup C \cup C'$.

In this definition we exclude variables from the set I' as the controller has no influence over any input values.

5.2 The verification problem

With these notions at hand we can state our verification problem: Given a ladder logic formula ψ and a safety condition φ, we say that ψ is safe w.r.t. φ,

$$A(\psi) \models \varphi,$$

iff $\mu \wr \mu' \models \varphi$ for all reachable paired valuations $\mu \wr \mu'$ in $A(\psi)$.

The exclusion of non-reachable states from the verification problem is motivated by the verification results in [KMS08] – see Section 1 – and comes as a direct request from Invensys. Our Pelican crossing program is safe w.r.t. *SingleAspect* only thanks to the exclusion of non-reachable states. For example, let μ, μ' and μ'' be states with $\mu = \{crossing = 1, req = 1, pressed = 1, tla\_g = 1, tlb\_g = 1, tla\_r = 0, tlb\_r = 0, pla\_g = 0, plb\_g = 0, pla\_r = 1, plb\_r = 1, audio{=}0\}$, $\mu' = \{crossing = 0, req = 0, pressed = 0, tla\_g = 0, tlb\_g = 0, tla\_r = 0, tlb\_r = 0, pla\_g = 0, plb\_g = 0, pla\_r = 1, plb\_r = 1, audio{=}0\}$ and μ'' any arbitrary successor of μ' (its existence is guaranteed by Theorem 1). $\mu \wr \mu'$ is safe, i.e. $\mu \wr \mu' \models SingleAspect$, there is a transition from μ' to μ'', however, $\mu' \wr \mu''$ is not safe, i.e. $\mu' \wr \mu'' \not\models SingleAspect$. But $\mu;\mu'$ is not reachable to begin with, see Figure 3.

It is obvious how to extend our setting to safety properties that involve $k > 2$ configurations of the interlocking: instead of paired valuations one has to define k-tuples of valuations; a safety property φ can speak about k different copies of each variable in $I \cup C$; and ψ is safe if all reachable k-tuples of consecutive states satisfy the safety condition φ.

6 Applying model checking to ladder logic

In this section we discuss two verification techniques based on SAT solving: bounded model checking [BCCZ99] and temporal induction [SSS00]. To allow us to apply these techniques, we firstly have to give a representation of the state sequences of the automaton under consideration.

6.1 Representing state sequences

Given a set I of input variables and a set C of state variables, we define variable sets $W_i = C^{(i)} \cup I^{(i)}$ with $C^{(i)} = \{c^{(i)} \mid c \in C\}$ and $I^{(i)} = \{x^{(i)} \mid x \in I\}$ for $i \in \mathbf{Z}$. Here we use the superscript (i) to produce fresh variables. We write $[W_i/(I \cup C)]$ to denote the substitution where all superscripts are removed, and $[W_{i+1}/(I' \cup C')]$ for the substitution where all superscripts are replaced by primes. A sequence $W_0, W_1, W_2, \ldots$ of these variable sets is capable to "store" a state sequence of an automaton $A(\psi)$:

Definition 5 (Series of transitions) Let ψ be a ladder logic formula. We define the propositional formulae

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

where $n \ge 0$ and $T(W_i, W_{i+1}) \equiv \psi[W_i/(I \cup C)][W_{i+1}/(I' \cup C')]$.

Given a ladder logic formula ψ, then the formula $Init \wedge T_n$ is "satisfied" exactly by all state sequences $s_0, s_1, \ldots, s_n$ of $A(\psi)$. More formally: Given a state sequence $s_0, s_1, \ldots, s_n$ we construct an valuation $\mu : W_{-1} \cup W_0 \cup \cdots \cup W_n \to \{1,0\}$, where state s_j gives the interpretation of W_j for $0 \le j \le n$, i.e. $\mu(i^{(j)}) = s_j(i)$, $i \in I$, and $\mu(c^{(j)}) = s_j(c)$, $c \in C$; $\mu(i^{(-1)}) = 0$, $i \in I$, and $\mu(c^{(-1)})$ such that we reach s_0 via ψ. For this μ holds: $\mu \models Init \wedge T_n$. Conversely, given a μ with $\mu \models Init \wedge T_n$ one can decompose it to a state sequences $s_0, s_1, \ldots, s_n$ of $A(\psi)$. With these notations in place, we can define safety at a specific point in a sequence $W_0, W_1, W_2, \ldots$.

Definition 6 (Safety at step n) Let φ be a safety condition for a ladder logic formula ψ. We define the propositional formula

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

where $n > 0$.

6.2 Bounded model checking

Widely used within industrial applications [CESS08, ADK+05], bounded model checking restricts the search space by a bound which states how many transitions of the automaton should maximally be considered for the verification process. Using the formulae

$$\mathit{Initial} \equiv \mathit{Init} \wedge T_1 \Rightarrow \varphi_1 \qquad \mathit{Transition}_n \equiv T_n \Rightarrow \varphi_n, \qquad \text{for } n > 0$$

the algorithm shown in Figure 4 performs a forwards iteration of the state-space. Given an automaton $A(\psi)$ and safety condition φ, the algorithm will check: (1) that φ holds on all transitions leaving the initial states of the automaton, and that (2) φ holds for up to K transitions from an initial state of the automaton.

```
if ¬Initial is satisfiable return error trace
j ← 2
while j ≤ K do
        if ¬Transition_j is satisfiable return error trace
        j ← j + 1
return "K-Safe"
```

Figure 4: K-step forwards iteration algorithm.

The algorithm in Figure 4 calls a SAT solver once in every iteration. In practice, the algorithm performs better when multiple calls to the SAT solver are combined into one call, namely for $l > 1$, "$\neg \mathit{Transition}_j$ satisfiable", ..., "$\mathit{Transition}_{j+l}$ satisfiable", are combined to *one* call, namely "$\neg(\mathit{Transition}_j \wedge \cdots \wedge \mathit{Transition}_{j+l})$ satisfiable".

Practical results from the pelican crossing example, show that verification times are less than one second[2]. With inductive verification, see [Kan08], verification of the safety condition given in Section 5.1 fails for the induction step. With the proposed bounded model checking approach, we were able to show that this was in fact due to unreachable states. That is, a bound size of $k = 6$ is required when using the given algorithm. Then via inspecting the state space given in Figure 3 we see that a bound of 6 covers all states.

6.3 Unbounded model checking

Temporal induction [SSS00] is a method that is based on strengthening the inductive approach as e.g., given by Kanso [Kan08]. As the name suggests, the verification method still consists of

[2] All results presented in this paper are based on tests carried out using a 64-bit computer, with a 3GHz quad-core processor and 8 GBytes of memory.

two proof steps, namely a base case and an inductive step. These proof steps are however used differently: the (negation of the) base case is checked for satisfiability, and the (negation of the) inductive step is checked for unsatisfiability. Our presentation follows [ES03].

We define properties of a state sequence encoded by $W_0, W_1, \ldots, W_n$:

$$LF_n \equiv \Big(\bigwedge_{0 \le k < l \le n} \neg(W_k \Leftrightarrow W_l) \Big) \qquad safe_n \equiv \bigwedge_{1 \le j \le n} \varphi_j$$

where $(W_k \Leftrightarrow W_l) \equiv \bigwedge_{i \in I} i^{(k)} \Leftrightarrow i^{(l)} \wedge \bigwedge_{c \in C} c^{(k)} \Leftrightarrow c^{(l)}$; $k, l, n \ge 0$. 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. The formula $safe_n$ encodes that all transitions between two consecutive states are safe. Using these formulae, we define the base case and induction step of temporal induction:

$$Base_n \equiv \text{Init} \wedge T_n \Rightarrow \varphi_n \qquad Step_n \equiv T_{n+1} \wedge LF_{n+1} \wedge safe_n \Rightarrow \varphi_{n+1}, \qquad \text{for } n \ge 0.$$

Figure 5 gives the temporal induction algorithm, similar to [SSS00, CESS08].

$n \leftarrow 1$
while true do
    if $\neg Base_n$ is satisfiable return trace
    if $\neg Step_n$ is unsatisfiable return "Safe"
    $n \leftarrow n + 1$

Figure 5: Temporal induction algorithm.

Theorem 2 *For all ladder logic formulae and safety conditions, temporal induction terminates, is sound, and is complete.*

Proof. (Only termination) 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 $Step_k \equiv T_{k+1} \wedge LF_{k+1} \wedge safe_k \Rightarrow \varphi_k$ is a tautology. Hence $\neg Step_k$ is unsatisfiable. □

This temporal induction algorithm verifies our pelican crossing example completely automatically. Once again, the verification time was less than one second.

7 Program slicing

The proposed approaches for the verification of ladder logic programs quickly give rise to large formulae to be verified. As the formula size increases, both the space and time requirements increase. This increase leads to a rather small bound[3] on the number of iterations of a ladder logic program we can verify in a feasible amount of time. Following approaches in [GKV95, FH98], we introduce slicing.

[3] I.e., with 361 variables, approximately 2000 iterations were possible.

Here, the novelty of our approach is that we prove slicing to be correct w.r.t. reachable states. Let Ψ be a program and φ be a property. Now consider two semantical approaches: $\models_{all}$ considers all states as in [GKV95, FH98], while $\models_{reach}$ – our approach – considers the reachable states only. Clearly, $\Psi \models_{all} \varphi$ implies $\Psi \models_{reach} \varphi$. However, as our Pelican crossing example demonstrates, the converse does not hold. Now consider a program Ψ_φ, which is a program Ψ sliced for φ. Slicing is considered correct if Ψ satisfies φ iff Ψ_φ satisfies φ. This results to two different correctness conditions, as illustrated by the following diagram:

$$\begin{array}{ccc} \Psi \models_{all} \varphi & \Leftrightarrow & \Psi_\varphi \models_{all} \varphi \\ \not\Uparrow \Downarrow & & \\ \Psi \models_{reach} \varphi & \Leftrightarrow & \Psi_\varphi \models_{reach} \varphi \end{array}$$

Note that we use the same slicing as [GKV95, FH98].

The intuition behind slicing is that the variables occurring in a safety condition often depend only on some part of the ladder logic program. Hence parts that have no effect on the safety condition can be removed.

7.1 Algorithm for slicing ladder logic

We begin by defining the dependence between rungs in a ladder logic formula.

Definition 7 (Dependency relation) Let $\psi = [R_1, R_2, \ldots R_n]$ be a ladder logic formula for some $n \geq 0$. We define the relation $dependant \subseteq \{1,\ldots,n\} \times \{1,\ldots,n\}$ between rungs of the ladder logic program, as the transitive closure of

$$\{(i,j) \mid j < i \text{ and } c'_j \in vars(\psi_i)\}$$

where rung k has the form $R_k \equiv c'_k \Leftrightarrow \psi_k$ for $1 \leq k \leq n$.

Using this notion of dependence, we define the slice of a ladder logic formula w.r.t. a safety condition as:

Definition 8 (Slice) Given a ladder logic formula $\psi = [R_1, R_2, \ldots R_n]$, and a safety condition φ, a slice ψ_φ of ψ is an order preserving selection of rungs such that the following two conditions hold:

- for all $1 \leq j \leq n : R_j \in \psi_\varphi$ if $c_j \in vars(\varphi) \vee c'_j \in vars(\varphi)$.
- for all $1 \leq i, j \leq n : R_j \in \psi_\varphi$ if $R_i \in \psi_\varphi$ and $(i,j) \in dependant$.

Given a slice ψ_φ we define the sets

$$\hat{I} = vars(\psi_\varphi) \cap I \qquad \hat{C} = \{c \in C \mid c' \in vars(\psi_\varphi) \cap C'\}$$

of those input variables (resp. state variables) that appear in the slice.

Note that this definition does not include a notion of minimality. Consequently, a ladder logic formula ψ is always a slice of itself. If the safety condition is $\varphi \equiv True$, then for every ladder

logic formula ψ we have that the empty program $\psi_\varphi \equiv true$ is a slice. To ensure that rung order is maintained, we compute a slice in a backward fashion. The algorithm we present is due to [GKV95, FH98].

Step 1 – Extract variables from safety condition. Given a safety condition φ of the form described in Section 5.1, we extract its variables: $U = vars(\varphi)$.

Step 2 – Calculate dependant variables. Calculate all the variables of the ladder logic formula that effect the variables in U. This step is repeated for each rung until a fixed point within the variable set is reached. Figure 6 illustrates the code that could be used to perform this step.

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

$$index = \{i \in \{1,\ldots,n\} \mid c_i \in \overline{U} \text{ or } c_i' \in \overline{U}\}.$$

Now, we remove from the original program all rungs R_i whose indicies do not appear in *index*. The result ψ_φ is the sliced version of program ψ.

```
do
    Ū ← U
    U_{n+1} ← U
    for i = n down to 1 do
        if c_i' ∈ U_{i+1} then U_i ← U_{i+1} ∪ vars(ψ_i) else U_i ← U_{i+1}
    U ← U_1
until U ⊆ Ū
return Ū
```

Figure 6: Algorithm to compute step two.

Figure 7 illustrates the effect of slicing the ladder logic formula of Figure 2 w.r.t. the safety condition presented in Section 5.1: The safety condition has four variables, six out of the original eleven rungs remain.

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

Figure 7: A sliced version of our pelican crossing ladder logic formulae.

7.2 Correctness of slicing

Given that slicing changes the ladder logic formulae under consideration, we need to ensure that the validity of safety conditions is still upheld.

Throughout this Section we assume that ψ_φ is the computed slice of a ladder logic formula $\psi = [R_1, R_2, \ldots R_n]$ w.r.t. a safety condition φ, where $\hat{I}$ is the set of inputs of ψ which appear in ψ_φ and $\hat{C}$ is the set of state variables of ψ required by ψ_φ – see Definition 8.

In order to compare the two automata $A(\psi)$ and $A(\psi_\varphi)$ we first need to relate their states. $A(\psi)$ has maps $\mu : (I \cup C) \to \{0,1\}$ as its states, while the states of $A(\psi_\varphi)$ take the form of maps $\nu : (\hat{I} \cup \hat{C}) \to \{0,1\}$. To this end, we define two functions: $\_\_|_{\hat{I}\cup\hat{C}}$ mapping states from $A(\psi)$ to states from $A(\psi_\varphi)$, and $\_\_ :: f$ mapping a state from $A(\psi_\varphi)$ to a state of $A(\psi)$, where f is a valuation that describes how we interpret the variables in $(I \cup C) - (\hat{I} \cup \hat{C})$.

Definition 9 (Reducing/Extending a valuation) Let μ be a state of $A(\psi)$. Its reduction $\mu|_{\hat{I}\cup\hat{C}} : \hat{I} \cup \hat{C} \to \{0,1\}$ w.r.t. $\hat{I} \cup \hat{C}$ is defined as $\mu|_{\hat{I}\cup\hat{C}}(x) = \mu(x)$ for all $x \in \hat{I} \cup \hat{C}$.

Let ν be a state of $A(\psi_\varphi)$. Let $f : (I \cup C) - (\hat{I} \cup \hat{C}) \to \{0,1\}$ be an evaluation. We define the extension of ν by f as $(\nu :: f) : C \cup I \to \{0,1\}$ where

$$(\nu :: f)(x) = \begin{cases} \nu(x) & \text{if } x \in \hat{I} \cup \hat{C} \\ f(x) & \text{otherwise} \end{cases}$$

for all $x \in C \cup I$.

Remark 2 *We also apply reduction and extension to paired valuations. That is,* $(\mu \fatsemi \mu')|_{\hat{I}\cup\hat{C}} = (\mu|_{\hat{I}\cup\hat{C}}) \fatsemi (\mu'|_{\hat{I}\cup\hat{C}})$ *is the paired evaluation obtained from individually reducing* μ *and* μ'*. Similarly* $\nu :: f \fatsemi \nu' :: f' = (\nu :: f) \fatsemi (\nu' :: f')$ *is the evaluation obtained by individually extending* ν *by* f *and* ν' *by* f' *and then pairing the results.*

We now study how to relate transitions of $A(\psi)$ to transitions of $A(\psi_\varphi)$: A step in $A(\psi)$ corresponds to a step in $A(\psi_\varphi)$; consequently, reachability in $A(\psi)$ implies reachability in $A(\psi_\varphi)$.

Lemma 1 ($A(\psi)$ transitions correspond to $A(\psi_\varphi)$ transitions) *Let* μ *and* μ' *be states of* $A(\psi)$.

1. $\mu \fatsemi \mu' \models \psi \Rightarrow \mu \fatsemi \mu'|_{\hat{I}\cup\hat{C}} \models \psi_\varphi$
2. *If* $\mu \fatsemi \mu'$ *is reachable with respect to* $A(\psi)$ *then* $\mu \fatsemi \mu'|_{\hat{I}\cup\hat{C}}$ *is reachable with respect to* $A(\psi_\varphi)$.

Proof. (Sketch) (1) follows as ψ_φ does not depend on removed variables. (2) is shown by induction on path length using point (1). □

Corresponding results hold for the reverse direction:

Lemma 2 ($A(\psi_\varphi)$ transitions can be extended to $A(\psi)$ transitions) *Let* ν *and* ν' *be states of* $A(\psi_\varphi)$.

1. *Let* $\nu \fatsemi \nu' \models \psi_\varphi$*. Then for all* f *there exists a* f' *such that* $\nu :: f \fatsemi \mu' :: f' \models \psi$.
2. *Let* $\nu \fatsemi \nu'$ *be reachable with respect to* $A(\psi_\varphi)$*. Then there exist* f, f' *such that* $\nu :: f \fatsemi \mu' :: f'$ *is reachable with respect to* $A(\psi)$.

Proof. (Sketch)

1. Choose f arbitrarily and define

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

   With these choices of f and f', ψ is satisfied.

2. By induction on path length and given point 1.

□

Using these lemmas we can prove that slicing is correct:

Theorem 3 *Let φ be a safety condition over a ladder logic formula ψ. Then*

$$A(\psi) \models \varphi \iff A(\psi_\varphi) \models \varphi.$$

Proof. By Lemma 1 and Lemma 2. □

Full proofs of Lemma 1, Lemma 2 and Theorem 3 are given in [Jam10].

8 Application and results

We summarise some results that have been obtained via a verification tool based on the discussed methods. A detailed discussion of the implementation of the tool, and the results are available in [Jam10]. In total, two railway interlocking ladder logic programs (one containing 331 rungs with 599 variables, and the other 238 rungs with 361 variables) were verified against a set of approximately ten safety conditions.

Overall, the results we have gained have been positive. For every safety condition the tool has either given a successful verification, or a counter example trace. All results have been obtained within the region of seconds.

With respect to the safety of the systems under consideration, all counter example traces could then manually be excluded as system runs by considering invariants. Such invariants have not been included in our automaton model, as in industry they are soft constraints used by the engineers, however, not part of the documentation for the interlocking control programs.

In this sense, our change of the semantic model, namely to consider reachable states only, turned out to be superfluous for the interlockings studied. It gave, however, an insight into the very nature of these interlockings and helped to understand the reasons why they are safe: not – as originally expected – due to unreachability, but thanks to states excluded by construction. Note that the inclusion of such invariants into our model will give rise to new proof obligations, namely counterparts for Theorem 1 will have to be established. As our Pelican crossing example demonstrates, the verification of technical systems can require our more sophisticated semantics, see Section 5.2.

8.1 Results of bounded model checking

The main success of the bounded model checking approach proved to be the generation of counter example traces. In all the verification results where inductive verification via Kanso's method [Kan08] gave a counter example, our forward iteration approach constructed a counter example trace.

Results obtained show that bounded model checking was possible up to two thousand iterations before memory issues occurred. With the application of our slicing algorithm, the number of iterations possible increased to twenty thousand. This is a large number of iterations, however, it remains unknown how many iterations would be required to verify all reachable states.

8.2 Results of temporal induction

The results obtained from the temporal induction approach are as expected:

Whenever inductive verification via Kanso's method [Kan08] succeeded, i.e., the safety property held, the safety property was also provable via temporal induction. In this special case, namely, that safety can be established via inductive verification, temporal induction is of equal complexity as Kanso's method: only its first iteration is executed, which requires the same resources as inductive verification.

Furthermore, whenever a counter example was generated using bounded model checking, a counter example would be generated by temporal induction. These two results show that temporal induction works correctly. The full power of temporal induction, however, is demonstrated by our Pelican crossing example: only temporal induction is capable of verifying it fully automatically.

8.3 Results of slicing

All results obtained show that applying slicing to the formulae to be verified resulted in large efficiency gains. The results are based on a set of approximately ten safety conditions which Invensys considered to be vital. Some analysis of the application of the slicing algorithm have shown that the following reductions were possible:

- For interlocking one, the number of rungs contained in the ladder logic formula, was reduced, on average from 331 rungs to around 60 rungs.
- For interlocking two, the number of rungs contained in the ladder logic formula, was reduced, on average from 238 rungs to around 25 rungs.

Obviously, the resultant formula size is dependant on the safety condition being verified. Hence it would be interesting to see the effect slicing has on more complicated, larger interlockings.

9 Conclusion

We have completed a feasibility study into various techniques for SAT-based model checking of Westrace interlockings. We have provided a formal model 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 bounded model checking and unbounded model checking via 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. This tool has been applied to verify real-world interlockings, with the main results being:

- 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.

Future work will include the removal of functional dependencies [JB04] and the verification of further interlockings.

Acknowledgements: We wish to thank Invensys for support and good cooperation; the Swansea Railway Verification Group, especially Faron G Moller, Anton G Setzer, and Monika Seisenberger for many helpful discussions; the reviewers for their helpful suggestions; and, finally, Erwin R. Catesbeiana (Jr) for guiding us along the correct route.

Bibliography

[ADK+05] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, K. L. McMillan. An Analysis of SAT-Based Model Checking Techniques in an Industrial Environment. In Borrione and Paul (eds.), *CHARME*. Springer, 2005.

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu. Symbolic Model Checking without BDDs. In Cleaveland (ed.), *TACAS '99*. Springer-Verlag, 1999.

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

[BHMW09] A. Biere, M. J. H. Heule, H. van Maaren, T. Walsh (eds.). *Handbook of Satisfiability*. IOS Press, 2009.

[Bjø09] D. Bjørner. Towards a Domain Model of Transportation. In *Domain Engineering – Technology Management, Research and Engineering*. JAIST Press, 2009.

[CESS08] K. Claessen, N. Een, M. Sheeran, N. Sörensson. SAT-solving in practice. In Lennartson et al. (eds.), *Proceedings of Workshop on Discrete Event Systems*. IEEE, May 2008.

[ES03] N. Een, N. Sörensson. Temporal Induction by Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science* 89(4), 2003. BMC'2003.

[FH98] W. Fokkink, P. Hollingshead. Verification of Interlockings: from Control Tables to Ladder Logic Diagrams. In Groote et al. (eds.), *FMICS'98*. CWI, 1998.

[GKV95] J. Groote, J. Koorn, S. Van Vlijmen. The safety guaranteeing system at station Hoorn-Kersenboogerd. In Danner et al. (eds.), *Compass'95, Computer Assurance*. IEEE, 1995.

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

[Jac04] R. Jacquart (ed.). *IFIP 18th World Computer Congress, Topical Sessions*. Chapter TRain: The Railway Domain - A Grand Challenge. Kluwer, 2004.

[Jam10] P. James. SAT-based Model Checking and its applications to Train Control Software. Master's thesis, Swansea University, 2010.

[JB04] J.-H. R. Jiang, R. K. Brayton. Functional Dependency for Verification Reduction. In Alur and Peled (eds.), *CAV*. Springer, 2004.

[JR10] P. James, M. Roggenbach. SAT-based Model Checking of Train Control Systems. In *Calco-Jnr 2009*. March 2010.

[Kan08] K. Kanso. Formal Verification of Ladder Logic. Master's thesis, Swansea University, 2008.

[KMS08] K. Kanso, F. Moller, A. Setzer. Verification of Safety Properties in Railway Interlocking Systems Defined with Ladder Logic. In Calder and Miller (eds.), *AVOCS08*. Glasgow 2008.

[Kul08] O. Kullmann. The OKlibrary: A generative research platform for (generalised) SAT solving. Technical report CSR 1-2008, Swansea University, 2008.

[SSS00] M. Sheeran, S. Singh, G. Stalmarck. Checking safety properties using induction and a SAT-solver. *Lecture Notes in Computer Science*, 2000.

[upp10] Uppaal Tool. Webpage, last accessed in July 2010. `http://www.uppaal.com/`.

[wes10] Westrace. Webpage, last accessed July 2010. `http://www.wrsl.com/assets/files/Interlocking/westrace/WESTRACE%20Intorduction.pdf`.

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

[ZRK03] B. Zoubek, J.-M. Roussel, M. Kwiatowska. Towards Automatic Verification of Ladder Logic Programs. In *Proceedings of IMACS-IEEE (CESA'03)*. 2003.