Verification of solid state interlocking programs

Phillip James[1], Andy Lawrence[1], Faron Moller[1], Markus Roggenbach[1], Monika Seisenberger[1], Anton Setzer[1], Karim Kanso[2], and Simon Chadwick[3]

[1] Swansea Railway Verification Group, Swansea University, Wales, UK
[2] Critical Software Technologies, Southampton, England, UK
[3] Invensys Rail, Chippenham, England, UK

Abstract. We report on the inclusion of a formal method into an industrial design process. Concretely, we suggest carrying out a verification step in railway interlocking design between programming the interlocking and testing this program. Safety still relies on testing, but the burden of guaranteeing completeness and correctness of the validation is in this way greatly reduced. We present a complete methodology for carrying out this verification step in the case of ladder logic programs and give results for real world railway interlockings. As this verification step reduces costs for testing, Invensys Rail is working to include such a verification step into their design process of solid state interlockings.

1 Introduction

Solid state interlockings represent one of many safety measures implemented in railways. In Vincenti's terminology [27], interlockings are *normal* designs: railway engineers have a clear understanding of their workings and customary features, and it is standard practice to design them and to bring them into operation.

The formal method we propose is a verification step between programming the interlocking using ladder logic [11] and testing of this program. The method we suggest is first to automatically translate the program as well as its desired properties and then to apply standard model checking approaches and tools to the resulting model checking problem.

Our work has been inspired by [8, 7]. [8] gives a detailed description of model checking railway interlockings and highlights the use of program slicing. [7] presents an approach to translate ladder logic programs into propositional logic and formulates model checking for sliced ladder logic programs. Alternative approaches include [28] who apply timed automata and UPPAAL or [9] who present a development framework for ladder logic, including verification by port-level simulation. Ladder logic programs for programable logic controllers in general have been verified using the symbolic model checker SMV [23]. Another type of interlocking program developed in so called "Safety Logic" has been verified using the SPIN model checker [3]. In [10], Haxthausen extracts a transition system (a SAL model) from circuit diagrams which are reminiscent of ladder logic programs. [15] verifies interlockings by interactive theorem proving, reducing the gap between verification of safety and safety in the real world.

This paper's contribution is, besides giving a precise formalisation of the translation from ladder logic into a model checking problem, to put known verification approaches into the context of a concrete engineering problem and, by providing a prototypical implementation, demonstrate that they work.

We first define interlockings and describe their design exemplified by the GRIP process and the realisation of GRIP's Detailed Design phase at Invensys Rail. We then detail our formal method, i.e., the verification step, and compile different technologies upon which the verification can be based. Finally, we present comparative results in terms of an industrial case study. This paper summarises results published in [15–18, 20].

2 Designing Solid State Interlockings

In railway systems, solid state interlockings provide a safety layer between the controller and the track. In order to move a train, the controller issues a request to set a route. The interlocking uses rules and track information to determine whether it is safe to permit this request: if so, the interlocking will change the state of the track (move points, set signals, etc.) and inform the controller that the request was granted; otherwise the interlocking will not change the track state. In this sense, an interlocking is like a Programmable Logic Controller (PLC). The standard IEC 61131 [11] identifies programming languages for such controllers, including the visual language ladder logic discussed below.

Interlockings applications are developed according to processes prescribed by Railway Authorities, such as Network Rail's *Governance for Railway Investment Projects* (GRIP) process. The first four GRIP phases define the track plan and routes of the railway to be constructed, while phase five – the detailed design – is contracted to a signalling company such as Invensys which chooses appropriate track equipment, adds control tables to the track plan, and implements the solid state interlocking. It is for part of this phase, namely for the correct implementation of a control table in a solid state interlocking, that our paper offers support in terms of a formal method.

Signalling handbooks (e.g. [21]) describe how to design control tables for the routes of a track plan. Technical data sheets provide information of how to control the selected hardware such as points, signals and track circuits. It is a complex programming task to implement the control tables for the selected hardware elements. For a larger railway station, the resulting program can involve thousands of tightly coupled variables, so thorough testing for safety is a must. To this end, programs are run on a rig which simulates the physical railway, and it can take any number of iterations of testing and debugging for a program to pass all prescribed tests. This testing cycle is cost intensive, as it is hardly automated due to its interactive nature and concerns about the safety integrity of any automated testing environment: the tester has to run the program through various scenarios developing over time. Furthermore, debugging is time consuming as there is little support for producing counter examples.

It is at this point that the formal method described below is able to reduce costs in the design process. Rather than testing an interlocking program, we automatically transform the program and the safety property that the test shall establish into a model checking problem. Tool support then allows to automatically check if the property is fulfilled. In case it is not, a counter example is produced, possibly in the form of a trace of controller requests and train movements. This allows the programmer to obtain intelligible feedback. This process is fast and far less involved than testing the program. For these reasons, based on our research, Invensys Rail is working to include such a verification step into their design process of solid state interlockings.

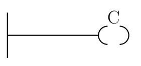
3 From Ladder Logic to Model Checking

We first introduce the programming language ladder logic, show how ladder logic programs can be represented in propositional logic, and give them a semantics in terms of transition systems. We then discuss how typical safety properties from the railway domain expressed in first order logic can be specialised to propositional logic. These two steps result in a model checking problem: is the specialisation of a safety property satisfied w.r.t. the labelled transition system gained from the ladder logic program? We discuss how to apply standard model checking approaches to this question and address the problem of false positives.

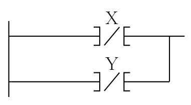
3.1 Ladder Logic

Ladder logic gets its name from its graphical "ladder"-like form (see Fig. 1) reminiscent of relay circuits. Each rung of the ladder computes the current value of an output. A ladder logic program is executed top-to-bottom, and an interlocking executes such a program indefinitely.

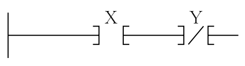
A ladder logic rung consists of the following entities. *Coils* represent boolean values that are stored for later use as output variables from the program. A coil is always the right most entity of the rung and its value is computed by executing the rung from left to right. *Contacts* are the boolean inputs of a rung, with *open* and *closed* contacts representing the values of un-negated and negated variables, respectively. The value of a coil is calculated when a rung fires, making use of the current set of inputs – input variables, previous output variables, and output variables already computed for this cycle – following the given connections. A horizontal connection between contacts represents logical conjunction and a vertical connection represents logical disjunction. For example:



(a) A coil



(b) Disjunction with closed contacts



(c) Conjunction with an open and a closed contact

As a running example we model a Pelican crossing, consisting of: two buttons at each side of a road, allowing pedestrians to make a request to cross; and four sets of lights (2 pedestrian lights, pla and plb, and 2 traffic lights, tla and tlb) controlling the flow of pedestrians and traffic. This is modelled by a boolean input variable *pressed* and 8 variables *plar*, *plag*, *plbr*, *plbg*, *tlar*, *tlag*, *tlbr*, *tlbg*, modelling the aspect of the light, 'r' for 'red', 'g' for 'green'. We also have two internal variables: *req* represents whether one of the pedestrian buttons has been pressed in a previous iteration of the program and whether there is already a request to cross; and *crossing* models the fact that a pedestrian is allowed to cross the road. Fig. 1 presents a ladder logic program for such a Pelican crossing.
The execution model for a ladder logic program is an infinite repetition of the

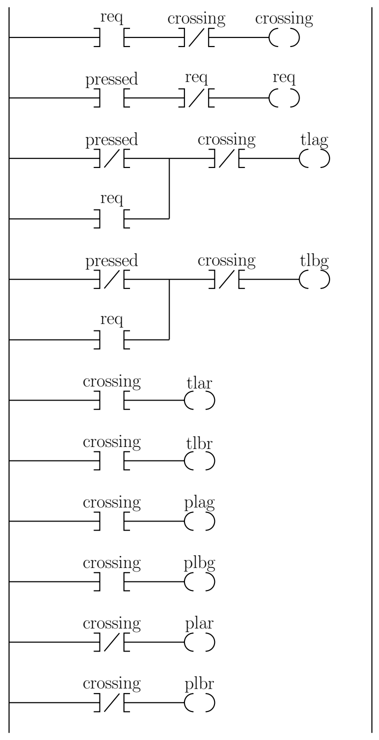


Fig. 1. The ladder logic program for the pelican crossing

sense-think-act cycle common in the design of embedded systems. *Sense*: all inputs are read; *think*: the program is executed; *act*: the outputs are all written. It thus makes sense to speak about consecutive execution cycles; and we discuss program execution for the current cycle, depending on the values of the input and the internal variables at the end of the previous cycle. We explain our use of ladder logic by considering the example program in Fig. 1.

- If the state variable "crossing" becomes true in Rung 1, then as a consequence, at the end of the cycle the pedestrian lights will be green and the traffic lights will be red (by Rungs 3-10).
- For "crossing" to become true, a request "req" to cross must have been made and in the previous cycle pedestrians could not cross (see Rung 1).
- The state variable "req" is true if and only if at (the beginning of) the previous cycle the button was pressed and at (the end of) the previous cycle the pedestrian lights were red i.e. "req" was previously false (see Rung 2)
- Rungs 5 and 6 control the setting of the red light for the traffic depending on on the state variable "crossing". Rungs 6-9 control the pedestrian lights depending on the state variable "crossing".
- Rungs 3 and 4 deliberately use a complicated encoding in order to demonstrate later the difference between model checking approaches. However, they still encode the correct behaviour: namely, setting the green light for traffic exclusively to setting the green light for pedestrians.

3.2 From Ladder Logic to Propositional Logic

From an abstract perspective, ladder logic diagrams represent propositional formulae. However, the process of obtaining these requires special care. In [14] we detail of how to use the Tseitin Transformation [26] in order to prevent a blow-up in formula size regarding nested disjunctions. This avoids bad performance when translating formulae into CNF, which is the usual input format of SAT solvers, the verification technology we intend to use.

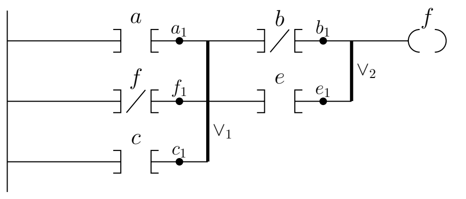


Fig. 2. Tracing back from coil f: Without auxiliary variables, the nested disjunction results in the large formula $f' \leftrightarrow (\neg b \wedge (a \vee \neg f' \vee c)) \vee (e \wedge (a \vee \neg f' \vee c))$.

The Tseitin Transformation traverses the formula from left to right, building up sub-formulae, each of which consisting of a conjunction or disjunction. The efficient use of sub-formulae requires the introduction of auxiliary variables. Fig. 2 shows an example and locations where variables are introduced. Here, a new variable is introduced for each step in the computation: After every contact x a new variable x_i is introduced (where i is fresh for x), and for each vertical connection (disjunction) a new variable $\vee_j$ is introduced (where j is fresh). The rung is then broken at each of the intermediate variables, resulting in a simplified ladder. Each rung in the simplified ladder consists of only conjunction or disjunction and at most one negation. By following the above procedure, applied to the ladder in Fig. 2, the below assignments and formulae are obtained:

$$
\begin{aligned}
a_1 &:= a \\
f_1 &:= \neg f \\
c_1 &:= c \\
\vee_1 &:= a_1 \vee f_1 \vee c_1 \\
b_1 &:= \vee_1 \wedge \neg b \\
e_1 &:= \vee_1 \wedge e \\
\vee_2 &:= b_1 \vee e_1 \\
f &:= \vee_2
\end{aligned}
$$

(a) Assignments of Fig. 2.

$$
\begin{aligned}
& (\ a_1' \leftrightarrow a \ ) \\
\wedge\ & (\ f_1' \leftrightarrow \neg f \ ) \\
\wedge\ & (\ c_1' \leftrightarrow c \ ) \\
\wedge\ & (\ \vee_1' \leftrightarrow a_1' \vee f_1' \vee c_1' \ ) \\
\wedge\ & (\ b_1' \leftrightarrow \vee_1' \wedge \neg b \ ) \\
\wedge\ & (\ e_1' \leftrightarrow \vee_1' \wedge e \ ) \\
\wedge\ & (\ \vee_2' \leftrightarrow b_1' \vee e_1' \ ) \\
\wedge\ & (\ f' \leftrightarrow \vee_2' \ )
\end{aligned}
$$

(b) Translation of Fig. 2.

The ladder logic of the Pelican logic Fig. 1 translates (for readability without the optimisation) into the conjunction of these formulae:

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

3.3 Ladder Logic Formulæ and their Semantics

A ladder logic program is constructed in terms of disjoint finite sets I and C of input and output variables, where internal variables are subsumed in C. In our example in Fig. 1, we have $I = \{\mathit{pressed}\}$ and $C = \{\mathit{crossing}, \mathit{req}, \mathit{tlag}, \mathit{tlbg}, \mathit{tlar}, \mathit{tlbr}, \mathit{plag}, \mathit{plbg}, \mathit{plar}, \mathit{plbr}\}$. We define $C' = \{c' \mid c \in C\}$ to be a set of new variables (intended to denote the output variables computed in the current cycle). In addition, we need a function $\text{unprime} : C' \to C, \text{unprime}(c') = c$.

Definition 1 (Ladder Logic Formulae) *A ladder logic formula ψ is a propositional formula of the form*

$$\psi \ \equiv \ (c_1' \leftrightarrow \psi_1) \ \wedge \ (c_2' \leftrightarrow \psi_2) \ \wedge \ \cdots \ \wedge \ (c_n' \leftrightarrow \psi_n)$$

such that the following holds for all $i, j \in \{1, \ldots, n\}$:

– $c_i' \in C'$;

– *$i \neq j \to c'_i \neq c'_j$; and*
– *$\mathrm{Vars}(\psi_i) \subseteq I \cup \{c'_1, \ldots, c'_{i-1}\} \cup \{c_i, \ldots, c_n\}$.*

Remark 1 *Note that the output variable c'_i of each rung ψ_i, may depend on $\{c_i, \ldots, c_n\}$ from the previous cycle, but not on c_j with $j < i$, due to the imperative nature of the ladder logic implementation. Those values are overridden.*

Remark 2 *In the formulae extracted from a ladder logic program equivalences $(c'_1 \leftrightarrow \psi_1) \wedge \cdots$ can be replaced by $(c'_1 = \psi_1) \wedge \cdots$. Both formulae are equivalent.*

Definition 2 (Semantics of Ladder Logic Formulae) *Let $\{0,1\}$ represent the set of boolean values and let*

$$\mathrm{Val}_I = \{\mu_I \mid \mu_I : I \to \{0,1\}\} = \{0,1\}^I$$
$$\mathrm{Val}_C = \{\mu_C \mid \mu_C : C \to \{0,1\}\} = \{0,1\}^C$$

be the sets of valuations for input and output variables. The semantics of a ladder logic formula ψ is a function that takes the two current valuations and returns a new valuation for output variables:

$$[\psi] \ : \ \mathrm{Val}_I \times \mathrm{Val}_C \to \mathrm{Val}_C$$
$$[\psi](\mu_I, \mu_C) \ = \ \mu'_C$$

where

$$\mu'_C(c_i) = [\psi_i](\mu_I, (\mu_C)_{\restriction\{c_i,\ldots,c_n\}}, (\mu'_C \circ \mathrm{unprime})_{\restriction\{c'_1,\ldots,c'_{i-1}\}})$$
$$\mu'_C(c) = \mu_C(c) \ \textit{\textbf{if}}\ c \notin \{c_1, \ldots, c_n\}$$

and $[\psi_i](\cdot,\cdot,\cdot)$ denotes the usual value of a formula under a valuation.

3.4 Labelled Transition Systems

We turn this into a transition system representing the ladder logic program.

Definition 3 (Labelled Transition System, reachability) *A Labelled Transition System (LTS) M is a four tuple (S, T, R, S_0) where*

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

We write $s \xrightarrow{t} s'$ for $(s, t, s') \in R$. A state s is called reachable *if*

$$s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} s_n = s,$$

for some states $s_0, \ldots, s_n \in S$, and labels $t_0, \ldots, t_{n-1} \in T$ where $s_0 \in S_0$.

Definition 4 (Ladder Logic Labelled Transition System) *We define the labelled transition system* $\mathrm{LTS}(\psi)$ *for a ladder logic formula* ψ *to be the four tuple* $(\mathrm{Val}_C, \mathrm{Val}_I, \rightarrow, \mathrm{Val}_0)$ *where*

– $\mu_C \xrightarrow{\mu_I} \mu'_C$ *iff* $[\psi](\mu_I, \mu_C) = \mu'_C$
– $\mathrm{Val}_0 \; = \; \{\mu_C \mid \mu_C$ *inital valuation*$\}$

Remark 3 *The standard initial valuation in the railway domain sets all red lights to* 1*, and all other variables to 0, i.e. this results in exactly one initial state. A variant proceeds as follows: First, all output variables are set to* 0 *and then all possible transitions are performed.* Val_0 *is then defined as the set of states obtained after this first transition. In the Pelican crossing example (see Fig. 3 below) this would lead to two initial states rather than one. In both cases, a formula Init characterises* Val_0*.*

3.5 Producing Verification Conditions

In order to guarantee safety, companies such as Invensys ensure through testing that interlockings fulfil certain properties. We formulate them as logical formulae, and call the result *safety conditions.* These conditions are the main example of *verification conditions*, which are formulae, for which we check using our tools whether they hold in an interlocking system. In our setting verification conditions are first-order formulae, with variables ranging over entities such as points, signals, routes, track segments, while referring to predicates. An example of a safety condition is the formula

$$\begin{aligned} &\forall rt, rt' \in \mathrm{Route}. \forall ts \in \mathrm{Segment}. \\ &\qquad \big(rt \neq rt' \;\wedge\; \mathrm{part\_of}(ts, rt) \;\wedge\; \mathrm{part\_of}(ts, rt')\big) \\ &\qquad\qquad\qquad \longrightarrow \neg\big(\mathrm{routeset}(rt) \;\wedge\; \mathrm{routeset}(rt')\big) \end{aligned}$$

expressing the property: *for all pairs of routes that share a track segment, at most one of them can be set to proceed.*

Note there are two kinds of predicates: *State* and *Topology*. State predicates express the state of entities at a given time; e.g., routeset(*rt26*) expresses that route *rt26* has been set. These predicates will unfold into variables in the ladder logic program, so in the previous example the predicate would—depending on the actual naming scheme—unfold to the variable *rt26ru*. *Topology* predicates express meta information relating to the topology of the railway yard. E.g. part_of(*ts54*, *rt26*) expresses that the track segment *ts54* is part of route *rt26*. These predicates unfold to *true* or *false*, depending on whether the property holds; thus, the previous example unfolds to *true* when *ts54* is actually part of *rt26*, otherwise *false*.

Some topology predicates are atomic and stated explicitly as true or false for given arguments. Other predicates can be computed in terms of these atomic predicates. E.g., signal *ms1* is a main signal guarding access to route *rt*, if there exists track segments *ts1* and *ts2* such that *ts1* is before route *rt*, *ts1* is connected with *ts2*, *ts2* is part of the route *rt*, and *ms1* is located directly between *ts1* and *ts2*. This can be expressed as follows:

$$\begin{aligned}&\text{route\_main\_signal}(\mathit{ms1}, \mathit{rt}) \ \leftrightarrow\ \exists \mathit{ts1}, \mathit{ts2} \in \text{Segment}.\\ &\qquad \text{before}(\mathit{ts1}, \mathit{rt}) \ \wedge\ \text{connected}(\mathit{ts1}, \mathit{ts2}) \ \wedge\ \text{part\_of}(\mathit{ts2}, \mathit{rt})\\ &\qquad\qquad\qquad \wedge\ \text{infrontof}(\mathit{ts1}, \mathit{ms1}) \ \wedge\ \text{inrearof}(\mathit{ts2}, \mathit{ms1})\end{aligned}$$

In [14, 16] Kanso introduces a translation of such formulae to propositional formulae which can then be verified using either SAT solving or model checking. This approach takes the following steps:

1. We first express the topology using a Prolog program that determines the truth of the topology predicates. The program consists of clauses such as
   - mainsignal(*ms1*) – signifying that *ms1* is a main signal, and
   - infrontof(*ts0a*, *ms1*) – signifying that signal *ms1* is in front of track segment *ts0a*.

   The above predicate route_main_signal(*ms1*, *rt*) is defined in Prolog as:
   route_main_signal(*ms1*, *rt*) :−
   $$\begin{aligned}&\text{before}(\mathit{ts}, \mathit{rt}),\ \text{connected}(\mathit{ts}, \mathit{tss}),\\ &\text{part\_of}(\mathit{tss}, \mathit{rt}),\ \text{infrontof}(\mathit{ts}, \mathit{ms1}),\ \text{inrearof}(\mathit{tss}, \mathit{ms1}).\end{aligned}$$
2. We then translate the formula into prenex form – i.e., a formula consisting of a block of quantifiers followed by a quantifier free formula – using standard techniques from logic.
3. Finally, we replace each occurrence of $\forall x{\in}A.\varphi(x)$ by $\varphi(a_1) \wedge \cdots \wedge \varphi(a_n)$ and each occurrence of $\exists x{\in}A.\varphi(x)$ by $\varphi(a_1) \vee \cdots \vee \varphi(a_n)$, where $a_1, \ldots, a_n$ are the elements of set A in the topology. φ is now instantiated to closed instances. Therefore the topological predicates evaluate to truth values that can then easily be omitted from the formula. Safety formulae can usually be translated into universally quantified formulae in prenex normal; the universally quantified formula is replaced by conjunctions, where most conjuncts reduce to false, since topology predicates such as connected(*ts1*, *ts2*) are false for most choices of arguments. Finally state predicates are replaced by the Boolean variables of the ladder logic. In the case of safety conditions we obtain a conjunction of instantiations of ψ. Since safety conditions usually become conjunctions, the validity of the conjuncts can be checked separately. This allows to identify problems relating specific objects of the railway yard.

A typical verification condition for our Pelican crossing example would for instance ensure that the traffic lights and the pedestrian lights are not green at the same time:

$$\varphi \ \equiv\ (\mathit{tlag} \wedge \mathit{tlbg} \wedge \neg \mathit{plag} \wedge \neg \mathit{plbg}) \ \vee\ (\neg \mathit{tlag} \wedge \neg \mathit{tlbg} \wedge \mathit{plag} \wedge \mathit{plbg})$$

3.6 The Model Checking Problem

We want to speak about the properties of the system that ensure safety – the so-called safety conditions – and then define what it means for a safety condition to hold in a labelled transition system. The following definition is motivated by the fact that safety conditions (tend to) describe properties which hold for two consecutive cycles of the ladder logic program.

Definition 5 (Safety Condition for a Ladder Logic Program) *Given a ladder logic formula ψ over the variables in $I \cup C$, a* ***verification condition*** *is a propositional formula formed from the variables in $I \cup C \cup C'$.*

Having defined the model of our system and the type of properties we want to speak about in that model, we must answer the following question: Given a model of our system and a safety condition, how do we check that the safety condition holds in that model. This motivates the following definition.

Definition 6 (Verification Problem for Ladder Logic Programs) *We define (and denote) the verification problem for a ladder logic formula ψ for a verification condition ϕ as follows:*

$\mathrm{LTS}(\psi) \models \phi$ *iff for all reachable transitions of the LTS – that is, triples μ_C, μ_I, μ'_C such that $\mu_C \xrightarrow{\mu_I} \mu'_C$, and μ_C is reachable in $\mathrm{LTS}(\psi)$ – we have $[\phi](\mu_C, \mu_I, \mu'_C) = 1$.*

Note that in most cases, as in our Pelican crossing, the verification condition ϕ only consists of variables in C, thus, the model checking problem simplifies to considering individual states, i.e. whether $[\phi](\mu_C) = 1$ at all times. Fig. 3 shows the labelled transition system for the Pelican crossing example. We have included one unreachable state in which both `required` and `crossing` are true.

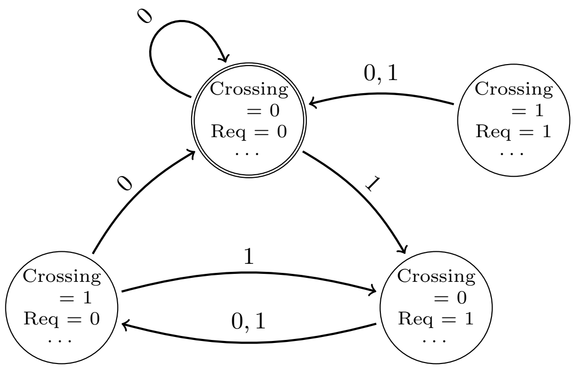


Fig. 3. Pelican crossing transition system

3.7 Model Checking Approaches

Target technology for the first three algorithms is SAT-solving; in the algorithms, execution terminates after a “return” statement has been performed.

Bounded Model Checking (BMC) BMC, see, e.g., [5], restricts the depth of the search space. Let the formulae ψ_n^{Init}, $n \geq 1$, be the unrolled transition relations which encode n steps with ψ from an initial state of the transition system. The following algorithm explores the transition system to a depth of up to K steps (we assume that ϕ uses the variables concerning the last transition):

```
if $\neg(Init \rightarrow \phi)$ is satisfiable, return error state
$n \leftarrow 1$
while $n \leq K$ do
    if $\neg(\psi_n^{Init} \rightarrow \phi)$ is satisfiable, return error trace
    $n \leftarrow n + 1$
return “K-Safe”
```

As BMC produces a counter example trace if the verification fails, it is especially interesting for debugging purposes.

Inductive Verification (IV) IV checks if an over approximation of the reachable state space is safe. In the following algorithm we assume that ϕ uses the variables concerning the current transition and ϕ' those concerning the last transition:

```
if $\neg(Init \rightarrow \phi')$ is satisfiable, return error state
if $\neg(\psi \wedge \phi \rightarrow \phi')$ is satisfiable, return pair of error states
return “Safe”
```

The over approximation happens in the second line of the algorithm: here one considers all safe transitions rather than the *reachable* ones. This idea makes IV a very efficient approach involving at most two calls to a SAT solver [14, 16].

Temporal Induction (TI) TI, see, e.g., [6], combines BMC and IV to allow for both complete verification and counter example production. For $n \geq 0$, let ψ_n be the unrolled transition relation encoding n steps with ψ; let LF_n be a formula encoding that all transitions of a sequence of n transitions are pairwise different; and $safe_n$ be a formula encoding that all these transitions fulfil the verification condition. Define

$Base_n \equiv \text{Init} \wedge \psi_n \rightarrow \phi$, and
$Step_n \equiv \psi_{n+1} \wedge LF_{n+1} \wedge safe_n \rightarrow \phi$, where ϕ uses the variables concerning the last transition.

We then have the following procedure.

```
$n \leftarrow 0$
while true do
    if $\neg Base_n$ is satisfiable, return error trace
    if $\neg Step_n$ is unsatisfiable, return “Safe”
    $n \leftarrow n + 1$
```

Stålmarck's Algorithm This algorithm has been developed and patented by Stålmarck [24]. It generally works well on industrial problems as – despite often being of a considerable size – they typically have a simple underlying structure.

Optimisation via Slicing Usually, the verification condition ϕ does not use all variables of the ladder logic formula ψ. This opens up the possibility to slice ψ with respect to ϕ, i.e., to compute a formula ψ_ϕ with $\psi \models \phi \Leftrightarrow \psi_\phi \models \phi$ where ψ_ϕ involves fewer variables and rungs than ψ. [8, 7] present an algorithm to compute ψ_ϕ, [12, 13] give a correctness proof. Here is the sliced ladder logic program of the Pelican crossing example for the condition $(tlag \vee tlar) \wedge \neg(tlag \wedge tlar) \wedge (tlbg \vee tlbr) \wedge \neg(tlbg \wedge tlbr)$:

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

Such slicing can be applied as a pre-processing step for all discussed approaches.

3.8 Excluding False Positives by Invariants

When verifying interlockings, often false positives are obtained. When discussing such false positives arising from the models with railway experts, they often state that in the physical system these situations do not occur because the specific value combination of the false positive is impossible, i.e., the false positive violates a system invariant. In [14] we identify two types of invariants.

Physical invariants are due to the fact that certain combinations of input variables are physically impossible. A typical example of this is a three way switch, modelled by 3 variables where each variable i indicates whether the switch is in position i or not. Physically it is impossible that such switch is in two positions simultaneously. This insight can be added to the system model as an invariant. However, in the real system it might happen that wet leaves fall on the three way switch and connect two of its contacts. This now puts the physical controller into a state that in the model was excluded by the physical invariant. Here, one has to decide if the system design and therefore its verification shall cater for such situations or not, i.e., physical invariants need to be carefully considered and validated by domain experts.

Mathematical invariants. In the case of IV, unreachable states may hinder verification through causing false positives. In this case one can identify invariants that hold in all reachable states. An example of such an invariant would be the equivalence $tlar \leftrightarrow tlbr$, which holds for the program given in Fig. 1.

3.9 Graphical Representation

In order to investigate counter examples a graphical representation of the error states was given. For our prototype, Kanso [14, 16] develops a latex document, which contains a scheme plan with signals, sets of points and routes, together with tables listing the state of all variables in question. The state of signals (red or green) and points and of all tables listed is determined by macros. It is now easy to compute from an error state a document setting these macros to the values in this state, and therefore present an easy to view document.

4 Technology & Case Studies

4.1 SAT solving with open software

An initial—successful—feasibility study was conducted using the open-source OKLibrary as underlying SAT solving framework to automate IV in order to establish safety properties. To this end, we used the Dimacs format as a target language. Note that this requires a representation in CNF.

Extending this implementation, we produced a framework of automatic translations of the formulae ψ, written in Haskell (about 8000 lines of code), and ϕ, written in Java (about 1000 lines of code), into the formulae required for the algorithms BMC, IV, and TI. As target format we chose TPTP [25], which is the input language of the Paradox tool [4]. Internally, the open source tool Paradox is based on the SAT solver Minisat [22], which is open source as well. Using Paradox has the advantage that the tool takes care of the translation into Dimacs format. The framework also includes a Haskell implementation of slicing (about 500 lines of code).

Using this framework, experiments on our Pelican crossing example with the above verification condition showed: with BMC the program is K safe for all $K \geq 0$ we tried; with IV, we obtain a pair of error states; TI gives the result "Safe". This example demonstrates that though IV is sound, it is not complete.

4.2 The *SCADE* Suite as an Industrial Tool

For comparison, we applied a tool widely used in industry, where however no control over the method applied is available. In *SCADE* (Safety Critical Applications Development Environment) [1] programs are verified using the *SCADE* language and Prover Technology based on Stalmarck's algorithm. The program to translate ladder logic programs into *SCADE* is based on the framework described above, it has a length of approximately 8000 lines of Haskell code [19].

The *SCADE* language is based on the synchronous dataflow language Lustre [2]. The flows which constitute a Lustre program are infinite sequences of values which describe how a variable changes over time. Flows are combined together to form nodes which can be seen as the Lustre equivalent of a function or procedure. There are two main temporal operations which can be applied to flows:

– The unary operator `pre` allows one to consider the previous value of a flow.
– The binary operator `->` allows one to express an initial value using the first operand and all subsequent values are computed using the second operand.

The following is the result of the automatic translation of the pelican crossing ladder logic to *SCADE*.

```
node PelicanLadderLogic1(pressed: bool)
returns (req, crossing, tlag, tlar, tlbg, tlbr, plag,
                        plar, plbg, plbr: bool)

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

4.3 Industrial Case Study

Using the approaches described above we automatically translated real world railway interlockings and safety properties into the Dimacs format (for IV), the TPTP language (for BMC, IV, and TI) and the *SCADE* language. The verification results gained have been positive. For every safety condition the tools have either given a successful verification, or a counter example (trace). All results have been obtained within the region of seconds.

In the following we report on the verification of a small, but real world interlocking which actually is in use on the London Underground. The ladder logic program consists of approximately six hundred variables and three hundred and fifty rungs. Concerning typical verification conditions, slicing reduces the number of rungs down to 60 rungs, i.e., the program size is reduced by a factor of 5. All experiments reported have been carried out on a computer with the operating system Ubuntu 9.04, 64-bit edition, an Intel Q9650, Quad core CPU with 3GHz, and a System Memory of 8GB DDR2 RAM.

Evaluation with an Open Source Tool The first condition encodes that if a point has been moved, it must have been free before. Here, the verification actually fails. IV yields a pair of states within 0.75s, while BMC produces an error trace of length 3 in 0.81s, TI produces the same trace. The rail engineers were able to exclude this counter example as a false positive. By adding justifiable invariants we could exclude this false positive. The second condition excludes that the program gives an inconsistent command, namely, that a point shall be set to normal and to reverse at the same time. IV proves this property in 0.71s; BMC yields K-safety for up to 1000 steps, after which we ran out of memory; BMC on the slided program is possible up to 2000 steps; TI does not terminate, neither for the original nor for the sliced version. Our experience is that IV can deal with real world examples. Slicing yields an impressive reduction of the size of the ladder logic program. It is beneficial when producing counter examples with BMC as it reduces the runtime and also helps with error localisation.

Verifying the Industrial Case Study using *SCADE* All above safety conditions take times less than 1s [19]. We attempted the verification of 109 safety conditions out of these 54 were valid and 55 produced counter examples. The latter are false positives and were eliminated by adding invariants as described above. The total time for the verification and production of counter examples for all of these safety conditions was under 10 seconds. This may be in part due to some support for multi-core processors allowing the *SCADE* suite to dispatch multiple verification tasks efficiently. Generally, in the process of removing false positives approximately one hundred invariants were added. Overall, this shows that *SCADE* is a viable option for the verification of railway interlockings.

5 Conclusion

The overall result is that the verification step described works out: the required translations can be automated, the current tools scale up to real world problems, the gained benefits are convincing enough for the company Invensys to change its practice. Concerning proof technology, it is a matter of taste / philosophy / further constraints if one prefers open software or commercial products.

Acknowledgement: Our thanks go to Ulrich Berger for advice on the semantics of ladder logic formulae.

References

1. P. Abdulla, J. Deneux, G. Stålmarck, H. Argen and O. Akerlund. Designing safe, reliable systems using SCADE. In *LNCS* **4314**:115-129, Springer 2006.
2. P. Caspi, D. Pilaud, N. Halbwachs and J. A. Plaice. LUSTRE: a declarative language for real-time programming. In *Proceedings of POPL'87*, pp178-188, 1987.
3. A. Cimatti, F. Giunchiglia, G. Mongardi, D. Romano. Formal verification of a railway interlocking system using model checking. *FACS* 10(4):361-380, Springer 1998.
4. K. Claessen and N. Sorensson. New techniques that improve mace-style finite model finding. In *Proceedings of CADE'03 Workshop: Model Computation*, 2003.
5. E. Clarke, A. Biere, R. Raimi and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design* 19(1):7-34, Kluwer, 2001.
6. N. Een and N. Sörensson. Temporal induction by incremental SAT solving. *ENTCS* 89(4):543-560, 2003.

7. W. Fokkink and P. Hollingshead. Verification of interlockings: from control tables to ladder logic diagrams. In *Proceedings of FMICS'98*, pp171-185, 1998.
8. J. Groote, J. Koorn and S. Van Vlijmen. The safety guaranteeing system at station Hoorn-Kersenboogerd. in Proceedings of *Compass'95*, pp57-68, 1995.
9. K. Han and J. Park. Object-oriented ladder logic development framework based on the unified modeling language. In *Computer and Information Science*, pp33-45, Springer, 2009.
10. A. Haxthausen. Automated generation of formal safety conditions from railway interlocking tables. *STTT*, Springer (to appear).
11. IEC 61131-3 edition 2.0 2003-01. International standard. Programmable controllers. Part 3: Programming languages. January 2003.
12. P. James. SAT-based model checking and its applications to train control software. MRes Thesis, Swansea University, 2010.
13. P. James and M. Roggenbach. Automatically verifying railway interlockings using SAT-based model checking. in *Proceedings of AVoCS'10*. Electronic Communications of EASST 35, 2010.
14. K. Kanso. Formal verification of ladder logic. MRes Thesis, Swansea University, 2009.
15. K. Kanso. Agda as a platform for the development of verified railway interlocking systems. PhD Thesis, Swansea University, 2012.
16. K. Kanso, F. Moller and A. Setzer. Automated verification of signalling principles in railway interlocking systems. *ENTCS* 250:19-31, 2009.
17. K. Kanso and A. Setzer. Specifying railway interlocking systems. In *Proceedings of AVoCS'09*, pp233-236, 2009.
18. K. Kanso and A. Setzer Integrating automated and interactive theorem proving in type theory. In *Proceedings of AVoCS'10*, 2010.
19. A. Lawrence. Verification of railway interlockings in SCADE. MRes Thesis, Swansea University, 2011.
20. A. Lawrence and M. Seisenberger. Verification of railway interlockings in SCADE. In Proceedings of *AVoCS'10*, 2010.
21. M. Leach (editor). *Railway Control Systems: a sequel to Railway Signalling.* A & C Black, 1991.
22. Minisat. `http://minisat.se`.
23. M. Rausch and B. Krogh. Formal verification of PLC programs. In *Proceedings of the American Control Conference.* IEEE, 1998.
24. G. Stålmarck. System for determining propositional logic theorems by applying values and rules to triplets that are generated from boolean formula. US Patent: 5,276,897, 1994.
25. The TPTP problem library for automated theorem proving. `http://www.cs.miami.edu/ tptp/`.
26. G.S. Tseitin. On the complexity of derivation in propositional calculus. Ina *Structures in Constructive Mathematics and Mathematical Logic*, Steklov Mathematical Institute, 1968.
27. W.G. Vincenti. *What engineers know and how they know it.* The Johns Hopkins University Press, 1990.
28. B. Zoubek, J.-M. Roussel and M. Kwiatkowska. Towards automatic verification of ladder logic programs. In *Proceedings of CESA'03*, Springer 2003.