Verification of solid state interlocking programs

Phillip James, Andy Lawrence
Faron Moller, Markus Roggenbach,
Monika Seisenberger, Anton Setzer
Swansea Railway Verification Group
Swansea University, Wales, UK

Karim Kanso
Critical Software Technologies
Southampton, England, UK

Simon Chadwick
Invensys Rail Northern Europe
a Siemens company
Chippenham, England, UK

***Abstract*—We report on the inclusion of a formal method into a design process in industry. 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 verfication 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.**

I. Introduction

Solid state interlockings represent one of many safety measures implemented in railways. In Vincenti's terminology [1], 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 and the testing of this program. On the one hand we have interlocking programs, their representation in propositional logic, and their semantics in terms of a labelled transition system; whilst on the other hand we have general safety properties expressed in first order logic, their specialization to propositional logic, and their satisfaction relative to the labelled transition system. Both representation and specialization can be automatically derived. The method we suggest is to apply standard model checking approaches and tools to the resulting model checking problem.

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 detail our formal method, i.e., the verification step, and compile different technologies upon which the verification can be based, giving comparative results in terms of a case study. We conclude with a brief discussion of related work and future research. This paper summarizes results published in [2]–[10].

II. Designing Solid State Interlockings

In railways 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. [12]) describe how to design control tables for the routes of a track plan selected for signalling. 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.

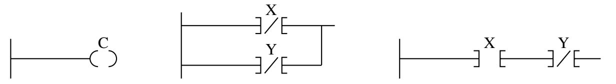
It is at this point that the formal method described below is able to reduce costs in the design process. Rather than testing an interlocking program, we automatically transform the program and the safety property that the test shall establish into a model checking problem. Tool support then allows to 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.

III. From Ladder Logic to Model Checking

A. 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 from the values of one or more inputs in the rung one time step (i.e. one cycle) earlier. 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.

B. From Ladder Logic to Propositional Logic

From an abstract perspective, ladder logic diagrams represent propositional formulae. However, the process of obtaining these formulae as described in [2] requires special care to prevent a blow-up in formula size regarding nested disjunctions, which would result in bad performance for CNF translation[1]. This is achieved by traversing 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.

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).

[1]Required when interfacing with theorem provers.

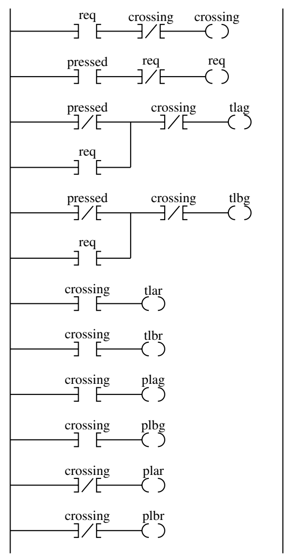


Fig. 1. The ladder logic program for the pelican crossing

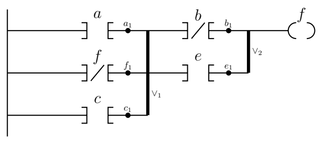


Fig. 2. Try tracing back from coil f: It is clear that the nested disjunction results in the large formula $f' \leftrightarrow (\neg b \wedge (a \vee \neg f' \vee c)) \vee (e \wedge (a \vee \neg f' \vee c))$.

The 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 are obtained.

Translating the assignments from (a) below is canonical with respect to the operators, giving the formula in (b):

$$\begin{array}{rl@{\qquad}ll}
a_1 & := a & & (\ a_1' \leftrightarrow a\ ) \\
f_1 & := \neg f & \wedge & (\ f_1' \leftrightarrow \neg f\ ) \\
c_1 & := c & \wedge & (\ c_1' \leftrightarrow c\ ) \\
\vee_1 & := a_1 \vee f_1 \vee c_1 & \wedge & (\ \vee_1' \leftrightarrow a_1' \vee f_1' \vee c_1'\ ) \\
b_1 & := \vee_1 \wedge \neg b & \wedge & (\ b_1' \leftrightarrow \vee_1' \wedge \neg b\ ) \\
e_1 & := \vee_1 \wedge e & \wedge & (\ e_1' \leftrightarrow \vee_1' \wedge e\ ) \\
\vee_2 & := b_1 \vee e_1 & \wedge & (\ \vee_2' \leftrightarrow b_1' \vee e_1'\ ) \\
f & := \vee_2 & \wedge & (\ f' \leftrightarrow \vee_2'\ )
\end{array}$$

(a) Assignments of Fig. 2. (b) Translation of (a).

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

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

$$\begin{array}{ll}
tlar' \leftrightarrow crossing', & tlbr' \leftrightarrow crossing', \\
plag' \leftrightarrow crossing', & plbg' \leftrightarrow crossing', \\
plar' \leftrightarrow \neg\, crossing', & plbr' \leftrightarrow \neg\, crossing'
\end{array}$$

C. Ladder Logic Formulae and their Semantics

A ladder logic program is constructed in terms of disjoint finite sets I and C of input and output variables. In our example in Fig. 1, we have $I = \{pressed\}$ and $C = \{crossing, req, tlag, tlbg, tlar, tlbr, plag, plbg, plar, plbr\}$. 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 $\mathrm{unprime} : C' \to C, \mathrm{unprime}(c') = c$.

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

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

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

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

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 since for Boolean values b and c the truth values of $b \leftrightarrow c$ and $b = c$ are the same. The use of $\leftrightarrow$ is suitable for the input language of SAT solvers, which require logical formulae (in our example combined with verification conditions) to be checked for satisfiability. The use $=$ is suitable for the input language of model checkers, which require equations defining the variables of the next state in terms of the current one.*

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

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

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

D. Labelled Transition Systems

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

Definition 3 (Labelled Transition System)**.** *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.*
- *$S_0 \subseteq S$ 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 $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} s_n$, for some states $s_0, \ldots, s_n \in S$, and labels $t_0, \ldots, t_{n-1} \in T$ such that $s_0 \in S_0$ and $s_n = s$.*

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 $(\mathrm{Val}_C, \mathrm{Val}_I, \to, \mathrm{Val}_0)$ 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 transistions 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 characterizes* Val_0.

E. 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 signalling principle is the formula

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

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. $\mathrm{routeset}(rt26)$ 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. $\mathrm{part\_of}(ts54, rt26)$ 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:

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

In [2], [6] Kanso introduced a translation of such formulae to propositional formulae which then can be verified using SAT solving or model checking. He took the following steps:

(1) Expressed the topology as a Prolog program, which determined the truth value of the topology predicates. It consisted of clauses such as $\text{mainsignal}(\mathit{ms1})$ ($\mathit{ms1}$ is a main signal), $\text{infrontof}(\mathit{ts0a}, \mathit{ms1})$ (signal $\mathit{ms1}$ is in front of track segment $\mathit{ts0a}$). The above predicate $\text{route\_main\_signal}(\mathit{ms1}, \mathit{rt})$ is defined in Prolog as:

$$\begin{aligned}&\text{route\_main\_signal}(\mathit{ms1}, \mathit{rt}) :-\\ &\quad \text{before}(\mathit{ts}, \mathit{rt}), \text{connected}(\mathit{ts}, \mathit{tss}),\\ &\quad \text{part\_of}(\mathit{tss}, \mathit{rt}), \text{infrontof}(\mathit{ts}, \mathit{ms1}),\\ &\quad \text{inrearof}(\mathit{tss}, \mathit{ms1}).\end{aligned}$$

(2) Translated using standard techniques from logic the formula into prenex form, i.e. a formula starting with a block of quantifiers followed by a quantifier free formula.

(3) Now $\forall x \in A.\varphi(x)$ is replaced by $\varphi(a_1) \wedge \cdots \wedge \varphi(a_n)$ and $\exists x \in A.\varphi(x)$ 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 valuese true or false, which can then easily be omitted from the formula. Safety formulae can usually be translated into universally quantified formulae in prenex normal form[2]. The universally quantified formula is replaced by conjunctions, where most conjuncts reduce to false, since topology predicates such as $\text{connected}(\mathit{ts1}, \mathit{ts2})$ are false for most choices of arguments. Finally state predicates are replaced by the Boolean variables of the ladder logic. In 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 for validity. 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:

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

F. The Model Checking Problem

Definition 5 (Safety Conditions 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 $I \cup C \cup C'$.*

Definition 6 (The Verification Problem for Ladder Logic Programs)**.** *We define the verification problem for a ladder logic formula ψ for a verification condition ϕ*

$$\text{LTS}(\psi) \models \phi$$

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

Note that in most cases, as in our Pelican crossing example, the verification condition ϕ only consists of variables in C, therefore the model checking problem simplifies to considering individual states, i.e. whether $[\phi](\mu_C) = 1$ at all times. Fig. 3[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.

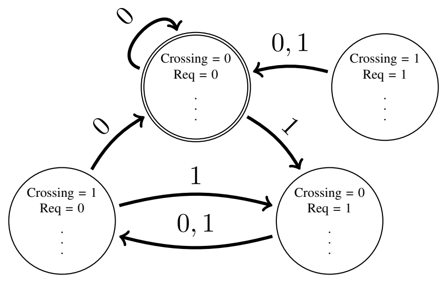


Fig. 3. Pelican crossing transition system

G. 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.

1) Bounded Model Checking (BMC): BMC, see, e.g., [13], restricts the depth of the search space. Let the formulae ψ_n^{Init}, $n \geq 1$, be unrolled transition relations which encode n steps with ψ from an initial state of the automaton. The following algorithm explores the automaton to a depth of up to K steps (we assume that ϕ uses the variables concerning the last state):

```
if $\neg(Init \rightarrow \phi)$ satisfiable, return error state
$n \leftarrow 1$
while $n \leq K$ do
    if $\neg(\psi_n^{Init} \rightarrow \phi)$ 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.

2) 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 state and ϕ' those concerning the last state:

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

The over approximation happens in the second line of the algorithm: here one considers all safe states rather than the *reachable* ones. This idea makes IV a very efficient approach involving at most two calls to a SAT solver [2], [6].

[2]$\forall x_1 \in A_1, \ldots, x_n \in A_n.\varphi(x_1, \ldots, x_n)$, where φ is quantifier free.

[3]The transition labelled 0,1 is in fact two transitions, one labelled with 1 and the other labelled with 0.

3) Temporal Induction (TI): TI, see, e.g, [14], combines BMC and IV to allow for both: complete verification and counter example production. Let ψ_n be the unrolled transition relation encoding n steps with ψ, let LF_n be a formula encoding that all n states of a sequence of states are pairwise different and $safe_n$ be a formula encoding that all these states fulfil the verification condition, $n \geq 0$. Define $Base_n \equiv \text{Init} \wedge \psi_n \rightarrow \phi$ and $Step_n \equiv \psi_{n+1} \wedge LF_{n+1} \wedge safe_n \rightarrow \phi$, $n \geq 0$, where ϕ uses the variables concerning the last state.

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

4) Stålmarck's Algorithm: This algorithm has been developed and patented by Stålmarck [15]. It usually works well on industrial problems as they are often of considerable size, but with a simple underlying structure. This is due to its ability to merge the conclusion of branches in a proof tree which can be seen as a form of learning. Its underlying theory was influenced by sequent calculus and semantic tableaux which inspired the branch and merge dilemma rule and the simple proof rules respectively. The algorithm makes use of equivalence classes in the form of data structures known as triplets.

5) Optimization 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. [16], [17] present an algorithm to compute ψ_ϕ, [4], [9] give a correctness proof. Here is the sliced ladder logic program of the Pelican crossing example for the condition $(tlag \vee tlar) \wedge \neg(tlag \wedge tlar) \wedge (tlbg \vee tlbr) \wedge \neg(tlbg \wedge tlbr)$:

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

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

H. Excluding False Positives by Invariants

When verifying concrete examples, often false positives were obtained. When discussing these counter examples with railway experts, one obtains usually that these examples do not occur because a certain combination of values for variables is not possible. This means that a certain invariant was violated. We identified [2] two kinds of invariants, *physical* invariants and *mathematical* invariants. *Physical invariants* are due to the fact that certain combinations of input variables are physically impossible. An example is a three way switch, which is modelled by 3 variables where each variable i indicates whether the switch is in position i or not.[4] It is physically impossible for this switch to be in two positions simultaneously. Physical invariants need to be carefully investigated by domain experts. One example could be a paper clip falling into a three way switch, which connects then two contacts, and one might want the railway yard to be safe even if a paper clip has falled into the switch.

[4]One could easily model it by 2 variables; however having 3 variables makes it easier to compute the next state from the current state.

Mathematical invariants. When using IV one might obtain states which violate the safety condition, but are not reachable from the initial state. In this case one can identify invariants, which hold in all reachable states but not in the false positive. In many cases one can prove now using the tool that the invariant holds in all cases, and then prove again using the tool that the verification condition holds provided the invariant holds.

I. Graphical Representation

In order to investigate counter examples a graphical representation of the error states was given. For our prototype Kanso [2], [6] developed a latex document, which contained 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 was determined by macros. It was 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.

IV. Technology & Case Studies

A. 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 requieres 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 [18], which is the input language of the Paradox tool [19]. Internally, the open source tool Paradox is based on the SAT solver Minisat [20], 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.

B. 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) [21] programs are verified using the *SCADE* language and Prover Technology based on Stalmarck's algorithm. The program to translate ladder logic programs into *SCADE* language is based on the framework described above, it has a length of approximately 8000 lines of Haskell code [5].

The *SCADE* language is based on the synchronous dataflow language Lustre [22]. 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 operator `pre` allows one to speak about the previous value of a flow.
- The operator `->` allows one to speak about the initial value of a flow and its successive values.

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
```

C. 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, 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.

1) 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 sliced 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 localization.

2) Verifying the Industrial Case Study using SCADE: All above safety conditions take times less than 1s [5]. 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.

V. Conclusion

The overall conclusion 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. In terms of the underlying proof technology, it is a matter of taste / philosophy / further constraints if one wants to employ open software tools or a commercial product.

Our work on verifying ladder logic programs has been inspired by [16], [17]. Alternative approaches include [23] who apply timed automata and UPPAAL or [24] who present a development framework for ladder logic, including verification by port-level simulation. Our contribution is to put known verification approaches into the context of a concrete engineering problem and, by providing a prototypical implementation, demonstrating that they work.

Putting the context even wider, in his PhD thesis [3] Kanso shows how to fully verify railway interlockings by interactive theorem proving. This work greatly reduces the gap between formal verification of safety and safety in the real world.

References

[1] W. G. Vincenti, *What engineers know and how they know it.* The Johns Hopkins University Press, 1990.

[2] K. Kanso, "Formal verification of ladder logic," 2010, MRes Thesis, Swansea University.

[3] ——, "Agda as a platform for the development of verified railway interlocking systems," 2012, PhD Thesis, Swansea University.

[4] P. James, "SAT-based model checking and its applications to train control software," 2010, MRes Thesis, Swansea University.

[5] A. Lawrence, "Verification of railway interlockings in SCADE," 2011, MRes Thesis, Swansea University.

[6] K. Kanso, F. Moller, and A. Setzer, "Automated verification of signalling principles in railway interlocking systems," *ENTCS*, vol. 250, pp. 19–31, 2009.

[7] K. Kanso and A. Setzer, "Specifying railway interlocking systems," in *PreProceedings of AVoCS'09*, 2009, pp. 233 – 236.

[8] ——, “Integrating automated and interactive theorem proving in type theory,” in *Proceedings of AVOCS 2010*, 2010.

[9] P. James and M. Roggenbach, “Automatically Verifying Railway Interlockings using SAT-based Model Checking,” in *Proceedings of AVoCS’10*. Electronic Communications 35 of EASST, 2010.

[10] A. Lawrence and M. Seisenberger, “Verification of railway interlockings in SCADE,” in *Proceedings of AVOCS 2010*, 2010.

[11] IEC, “IEC 61131-3 edition 2.0 2003-01. international standard. programmable controllers. part 3: Programming languages,” January 2003.

[12] M. Leach, Ed., *Railway Control Systems: a sequel to Railway Signalling*. A & C Black, 1991.

[13] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, “Bounded model checking using satisfiability solving,” in *Formal Methods in System Design*. Kluwer Academic Publishers, 2001, p. 2001.

[14] N. Een and N. Sörensson, “Temporal induction by incremental SAT solving,” *ENTCS*, vol. 89, no. 4, 2003.

[15] G. Stålmarck, “System for determining propositional logic theorems by applying values and rules to triplets that are generated from boolean formula,” 1994, US Patent: 5,276,897.

[16] J. Groote, J. Koorn, and S. Van Vlijmen, “The safety guaranteeing system at station Hoorn-Kersenboogerd,” in *Compass’95*. IEEE, 1995.

[17] W. Fokkink and P. Hollingshead, “Verification of interlockings: from control tables to ladder logic diagrams,” in *FMICS’98*, 1998.

[18] “The TPTP problem library for automated theorem proving,” `http://www.cs.miami.edu/ tptp/`.

[19] K. Claessen, “New techniques that improve mace-style finite model finding,” in *CADE-19*, 2003.

[20] “Minisat,” `http://minisat.se`.

[21] P. Abdulla, J. Deneux, G. Stålmarck, H. Argen, and O. Akerlund, “Designing safe, reliable systems using SCADE,” in *Springer LNCS 4313*, 2006, pp. 115–129.

[22] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, “LUSTRE: a declarative language for real-time programming,” in *POPL’87*, 1987.

[23] B. Zoubek, J.-M. Roussel, and M. Kwiatowska, “Towards automatic verification of ladder logic programs,” in *CESA’03*. Springer, 2003.

[24] K. Han and J. Park, “Object-oriented ladder logic development framework based on the unified modeling language,” in *Computer and Information Science*. Springer, 2009.