Verification of train control systems: Reducing the complexity

Phillip James

Department of Computer Science
Swansea University, Wales.

BCTCS – April 2010

In co-operation with Invensys

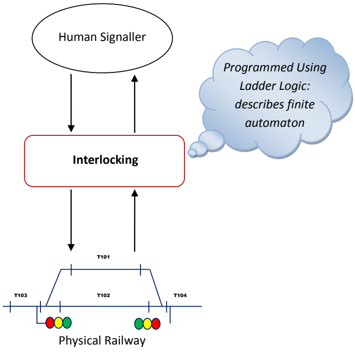
Overview

- Verification within the Railway Domain.
- Our Approach.
  - Modelling.
  - Slicing.
  - Reachability Algorithms.
- Implementation and Results.

Verification within the Railway Domain

Safety within the Railway Domain

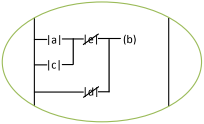
An interlocking is major system responsible for enforcing safety.



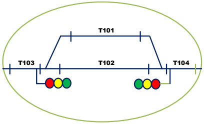
- Interface between signaller and the physical track.
- Implemented as single control loop.

Successful Railway Verification – Kanso 2008

Interlocking Ladder Logic



Railway Topology



Informal Safety Condition

Initial Condition - $I(\mu)$

and **Transition Formula** - $T(\mu, \mu')$

$\varphi(\mu)$

Safety Condition

μ, μ' : *Represent system states.*

(i.e. - Valuations of propositional variables)

Successful Automated Verification:

1. $\neg(I(\mu) \Rightarrow \varphi(\mu))$
2. $\neg(\varphi(\mu) \wedge T(\mu, \mu') \Rightarrow \varphi(\mu'))$

If 1 & 2 *are not satisfiable* then output “safe”

Overcoming Limitations and Our Aims

Limitations of Kanso'08

- Violations that are unreachable (Invensys).
- Production of counterexample trace is not possible.
- Invariants require domain knowledge.

Overcoming Limitations and Our Aims

Limitations of Kanso'08

- Violations that are unreachable (Invensys).
- Production of counterexample trace is not possible.
- Invariants require domain knowledge.

Our aims:

- A verification method which only considers reachable states.
- If a counterexample is found, produce an error trace.
- Validate techniques: encode and verify a new interlocking.
- Implement these techniques into a usable verification tool.

Our Approach

Automata Definition

Definition: Ladder Logic Automaton

Given a ladder logic propositional formula ψ_P over $I \cup C$, define

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

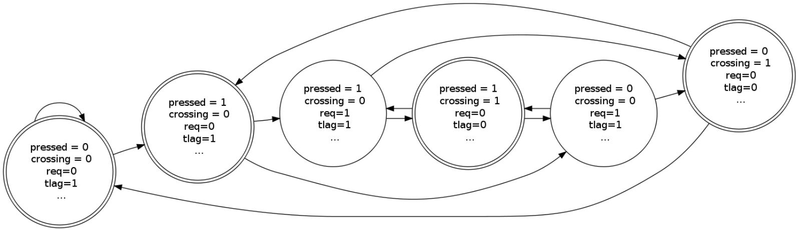
where

- $S = \{\mu \mid \mu : I \cup C \rightarrow \{0,1\}\}$,
- $\mu \rightarrow \mu'$ if $\mu \,\mathrm{⨾}\, \mu' \models \psi_P$,
- $I_s = \{\mu' \mid \mu \models (\bigwedge_{i \in I} \neg i), \mu \,\mathrm{⨾}\, \mu' \models \psi_P\}$

Definition: Satisfaction (verification)

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

An example automaton



Program Slicing Example

Slicing a ladder with regard to a safety condition:

$$(tlag1 \vee tlar1) \wedge \neg(tlag1 \wedge tlar1) \wedge (tlbg1 \vee tlbr1) \wedge \neg(tlbg1 \wedge tlbr1).$$

```
while(true){
crossing1 = (req0 && ...
req1 = (pressed0 && ...
tlag1 = ((not crossing1) ...
tlbg1 = ((not crossing1) ...
tlar1 = crossing1;
tlbr1 = crossing1;
plag1 = crossing1;
plbg1 = crossing1;
plar1 = (not crossing1);
plbr1 = (not crossing1);
audio1 = crossing1;
}
```

```
while(true){
crossing1 = (req0 && ...
req1 = (pressed0 && ...
tlag1 = ((not crossing1) ...
tlbg1 = ((not crossing1) ...
tlar1 = crossing1;
tlbr1 = crossing1;
}
```

Algorithm by Fokkink'98 gives new sliced transition formula ψ_{P_φ}.

New Program Slicing Theorem

Correctness differs to Fokkink'98:

We explicitly consider the reachable states of an automaton.

Theorem: Correctness of Slicing

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

$$A(\psi_P) \models \varphi \Leftrightarrow A(\psi_{P_\varphi}) \models \varphi.$$

One Verification Algorithm

Definition: Formulae for Temporal Induction

Define:

- $Base_n = I(W_0) \wedge T_n \Rightarrow \varphi_n$.
- $Step_n = T_{n+1} \wedge LF_{n+1} \wedge \varphi_n \Rightarrow \varphi(W_n, W_{n+1})$

Temporal Induction Algorithm

```
n ← 0
while true do
    if ¬Base_n is satisfiable return trace
    if ¬Step_n is unsatisfiable return "Safe"
    n ← n + 1
od
```

Further Algorithms Studied

Along with Temporal Induction, the following have been explored and implemented:

- Bounded and unbounded model checking via:
  - Forward and backward iteration.
  - Formulating inclusion checks.
- Applying slicing to each approach:
  - Reduction from 600 to 60 rungs (approx).

Implementation and Results

Improvements and Verification Results

Overall the tool from Kanso'08 has been improved:

- Overall software architecture has been simplified.
- Extended to allow verification of new interlocking.
- Extended with various verification techniques.
- Improved verification time (From minutes to seconds).

The tool has been used to verify 2 interlockings where:

- Verification times were in the region of seconds.
- All safety properties were
  1. verified, or
  2. a counterexample trace was generated.

Counter Example Traces

```
.
.
v8253_1__EFM_1 <=> $false
v8253_1__EFM_2 <=> $false
v8253_1__F_0 <=> $false
v8253_1__F_1 <=> $false
v8253_1__F_2 <=> $true
v8253_1__F_3 <=> $false
v8253_1__FM_0 <=> $false
v8253_1__FM_1 <=> $true
v8253_1__FM_2 <=> $true
v8253_1__FM_3 <=> $false
.
.
```

Summary and Future Work

Overall the main results have been:

- The successful verification of 2 interlockings.
- Improved verification tool (Speed and Architecture).
- Correctness result for slicing.

In the future we wish to explore:

- Further reduction via functional dependency removal.
- Using a higher level language with domain specific data types.
- Compositional verification and tool integration.

Functional Dependency Example

```
while(true){
crossing1 = (req0 && ...
req1 = (pressed0 && ...
tlag1 = ((not crossing1) ...
tlbg1 = ((not crossing1) ...
tlar1 = crossing1;
tlbr1 = crossing1;
plag1 = crossing1;
plbg1 = crossing1;
plar1 = (not crossing1);
plbr1 = (not crossing1);
audio1 = crossing1;
}
```

```
while(true){
crossing1 = (req0 && ...
req1 = (pressed0 && ...
}
```

Finally re-write safety condition in terms of these.

Thanks!