SAT-based Model Checking of Train Control Software.

Phillip James    Markus Roggenbach

Department of Computer Science
Swansea University, United Kingdom

CALCO-jnr’09

In co-operation with Invensys.
Overview

- Verification Within The Railway Domain.
- Reachable State Algorithms.
- Real World Interlockings.
Verification Within The Railway Domain
Motivation


25 people killed, over 100 people injured.
A major system responsible for ensuring safety within the railway is the interlocking system.

- Interlockings control aspects such as signals and points.
- Interlockings are written by Invensys in a logic similar to propositional logic.
Verification Within The Railway Domain
Our Approach
Real World Interlockings

Railway Verification in Propositional Logic – Kanso 2008

Ladder Logic

Railway Topology

Informal Safety Condition

Safety Condition

¬(occ(t1)) ∧ (occ(t2))
...

For all track segments...

Verification Via SAT Solver
Successful Automated Verification:
1. ¬(I(μ) ⇒ φ(μ))
2. ¬(φ(μ) ∧ T(μ,μ') ⇒ φ(μ'))

Phillip James
Verifying Train Control Software
Discussion of Kanso’08

Positive:
- Successful verification of some safety properties of a real interlocking.

Problematic:
- Unclear: Is a violation reachable?
- Costly human interaction required.
Our Aims

- If a counterexample is found, produce an error trace to the counterexample.
- Devise a verification method which ignores unreachable states.
- Implement these techniques into a useable verification tool which works on real world interlockings.
Our Approach

Phillip James

Verifying Train Control Software
**SAT-based Model Checking**

**k-bounded Model Checking**

\[
i \leftarrow 0
\]
\[
B_0 \leftarrow \{\mu \mid I(\mu)\}
\]

**while** \(i \leq k\) **do**

\[
\text{for } \mu \in B_i, \text{ if } \neg(\varphi(\mu)) \in \text{SAT} \text{ return } \text{“unsafe” + trace; stop}
\]
\[
B_{i+1} \leftarrow \{\mu' \mid T(\mu, \mu'), \mu \in B_i\}
\]
\[
i \leftarrow i + 1
\]

**return** “safe”

**Unbounded Model Checking**

Change \(i \leq k\) to \(B_{i+1} \subseteq B_0 \cup \cdots \cup B_i\).
Some Definitions

**Definition: Series of transitions.**

We define a series of $n$ transitions $T_n$ in an automaton as follows:

$$T_n = \bigwedge_{0 \leq i \leq n-1} T(S_i, S_{i+1})$$

where $T(S_i, S_{i+1})$ is a transition from state $S_i$ to state $S_{i+1}$.

Formula size: $O(kn)$, $k$ number of rungs, $n$ number of iterations.
Knowing when to stop?

**Definition: Inclusion Property**

We define an inclusion check as:

\[
P = I_n \land T_{n+1} \Rightarrow (LF_n \Rightarrow \bigvee_{i \leq n+1} S_{n+1} \Leftrightarrow S_i)
\]

If \( P \Leftrightarrow true \) then inclusion has been reached.

**Definition: Loop Freedom**

An Automaton \( A \) is loop free for \( n \) transitions if the following holds:

\[
LF_n = T_n \land \bigwedge_{0 \leq i \leq j \leq n-1} \neg (S_i = S_j)
\]

Formula size: \( O(kn^2) \), \( k \) number of rungs, \( n \) number of iterations.
Real World Interlockings
Problem size:
- Ladder Logic for small train station – about 550 variables.
- 1 iteration = (approx) 1 second of run-time.

Experiments:
- Without inclusion:
  - Only 500 iterations possible due to state space explosion.
  - Verification time – 523(s), more iterations: out of memory.
- With inclusion:
  - Only 50 iterations possible due to large formulae.
  - Verification time – 652(s), more iterations: out of memory.

Slicing needed!
Program Slicing

Main Idea: Construct a program slice by removing variables/rungs which have no effect on the safety condition.

- Algorithm thanks to Fokking et al.
- New correctness statement and proof:
  consider reachable states only!
Slicing a ladder with regard to a safety condition:

\[(\text{tlag}_1 \lor \text{tlar}_1) \land \neg(\text{tlag}_1 \land \text{tlar}_1) \land (\text{tlbg}_1 \lor \text{tlbr}_1) \land \neg(\text{tlbg}_1 \land \text{tlbr}_1)\].

```java
1 while (true) {
2  crossing1 = (req0 && ...)
3  req1 = (pressed0 && ...)
4  tlag1 = ((not crossing1) ...)
5  tlbg1 = ((not crossing1) ...)
6  tlar1 = crossing1;
7  tlbr1 = crossing1;
8  plag1 = crossing1;
9  plbg1 = crossing1;
10  plar1 = (not crossing1);
11  plbr1 = (not crossing1);
12  audio1 = crossing1;
13 }

Figure: Original Ladder

1 while (true) {
2  crossing1 = (req0 && ...)
3  req1 = (pressed0 && ...)
4  tlag1 = ((not crossing1) ...)
5  tlbg1 = ((not crossing1) ...)
6  tlar1 = crossing1;
7  tlbr1 = crossing1;
8 }

Figure: Sliced Ladder
```
Correctness Theorem

Theorem:

Given a ladder logic program $P$ and a safety condition $\varphi$,

$$A(P) \models \varphi \iff A(P|\varphi) \models \varphi.$$  

Proof Sketch: Argue on reachability of states in each automaton.

Implementation of slicing in Haskell.
Our Results on Real World Interlockings:

- Ladder with approx 550 variables reduced to ladder with 62 variables.
  - Without Inclusion:
    - Up to 2000 iterations 4553(s), more iterations: out of memory.
  - With Inclusion:
    - Up to 200 iteration 1554(s), more iterations: out of memory.

Underlying prover: Equinox.

Commercial Tool: about 100 iterations.
k-bounded Model Checking:

<table>
<thead>
<tr>
<th>Property</th>
<th>Kanso’08</th>
<th>k-bounded MC</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varphi_1$</td>
<td>Safe</td>
<td>Safe</td>
</tr>
<tr>
<td>$\varphi_2$</td>
<td>Unsafe</td>
<td>Counterexample (4 iterations)</td>
</tr>
<tr>
<td>$\varphi_3$</td>
<td>Unsafe</td>
<td>Counterexample (3 iterations)</td>
</tr>
<tr>
<td>$\varphi_4$</td>
<td>Unsafe</td>
<td>Counterexample (1 iteration)</td>
</tr>
</tbody>
</table>

Unbounded Model Checking:

- Inclusion not reached in 200 iterations.
- Current slices: $\sim 60$ variables.
- Experiments show: $\sim 30$ variables work out.
Conclusion
New slicing Theorem w.r.t. reachable states only.

Slicing works very well to reduce formulae size.

Verified successfully two real interlockings:
- For all given safety conditions we either -
  - proved safety, or
  - returned counter example.

Open problem (with no impact to practice?): Inclusion not reached, formulae still too big.
Future Work

- Remove functional dependencies: to reduce formulae size further.
- Look at modelling using First Order logic.
- Explore compositional reasoning of ladder logic templates used by Invensys.
Thanks!
Definition: Automaton

Given a ladder logic program $P$ over $V = I \cup O \cup O'$. An automaton is a triple $(S, I, \rightarrow)$, where

- $S = \{\nu | \nu : I \cup U \rightarrow \{0, 1\}\}$.
- $I = \{\nu' | \nu \models \neg l_{cond}, \nu \cup \nu' \models \psi_P\}$
- $\nu \rightarrow \nu'$ iff $\nu \cup \nu' \models \psi_P$. 

Phillip James  
Verifying Train Control Software