Verifying Railway Interlockings Using *SCADE*

Andy Lawrence
Swansea University

7th April 2010

A project in cooperation with Invensys Rail UK.

An Overview of the Presentation

Overview:

- *SCADE*
- Pelican Crossing: to demonstrate modelling and use of *SCADE*'s model checking capabilities.
- Case Study: A Real Interlocking.
- Project programme.

In this talk we will concentrate on *SCADE* and its application, rather than on the underlying theory and techniques.

The following presents the progress since the start of the project in November.

Railway Interlockings and Ladder Logic

Railway engineers use a programming language called Ladder Logic:

- A graphical language for programming logic controllers.
- Part of the **IEC 61131** standard.
- Sequentially executed
- The subset used here is similar to propositional logic.

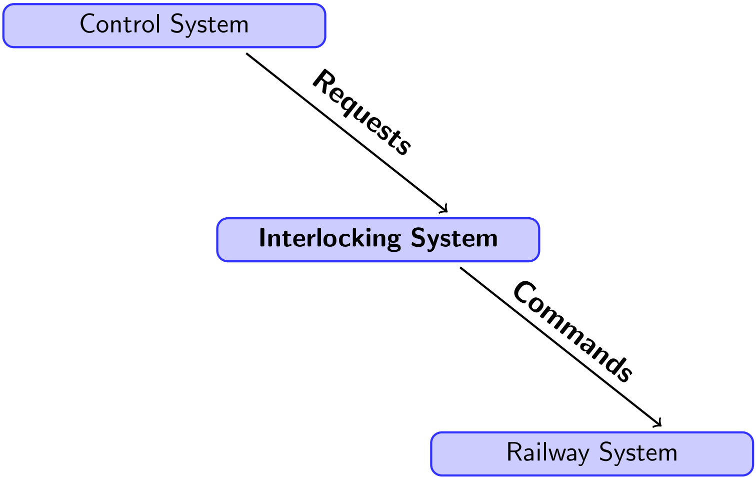
The three main stages in an execution cycle of an interlocking are:

- Reading of Inputs
- Internal Processing
- Committing of Outputs

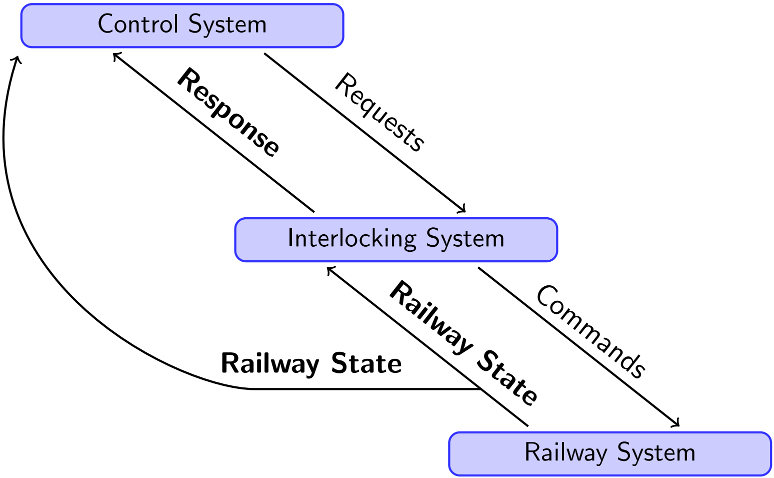
Control System

Railway System

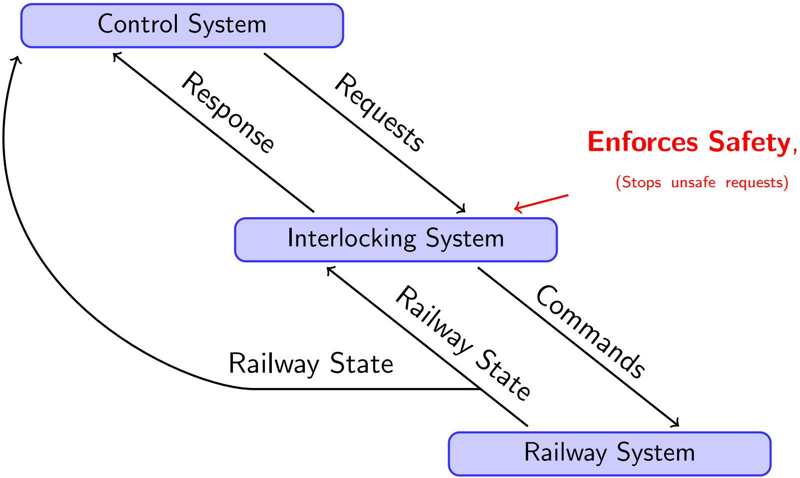
Railway Interlockings and Ladder Logic



Railway Interlockings and Ladder Logic



Railway Interlockings and Ladder Logic

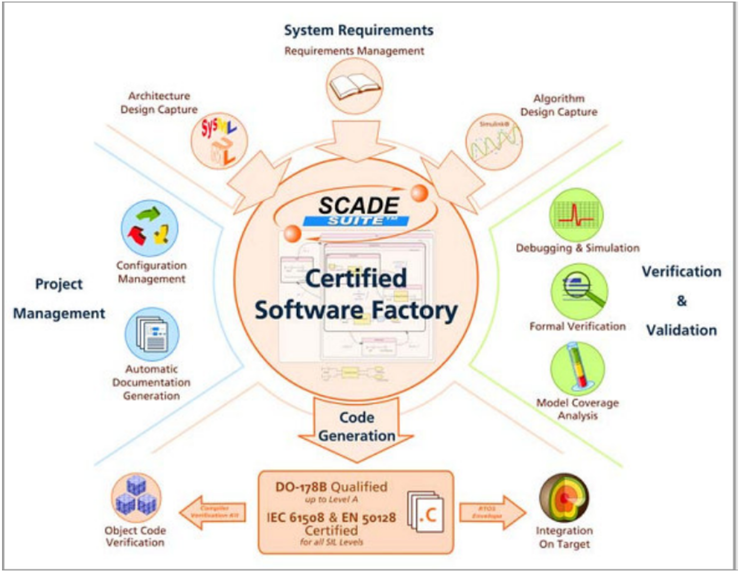


SCADE Suite

The *SCADE* Suite by Esterel Technologies is a IDE for developing safety critical embedded systems.

SCADE moto: Design, Verify, Generate.

Certified compiler: EN 50128 Software for railways and protection systems.


System Requirements
Requirements Management
Architecture
Design Capture
Algorithm
Design Capture
SCADE
SUITE
Certified
Software Factory
Configuration
Management
Project
Management
Automatic
Documentation
Generation
Debugging & Simulation
Formal Verification
Model Coverage
Analysis
Verification
&
Validation
Code
Generation
DO-178B Qualified
up to Level A
IEC 61508 & EN 50128
Certified
for all SIL Levels
.C
Object Code
Verification
Integration
On Target

Model Checking Techniques Used By *SCADE*

Verification Techniques applied in *SCADE*.

- Bounded model checking.
- Induction over time.
- ...

The following techniques are applied in *SCADE*'s built in model checker in order to decide the Satisfiability of formulas:

- Stålmarck's saturation method.
- Davis-Putman-Loveland-Logemann.
- Reduced ordered binary decision diagrams.
- ...

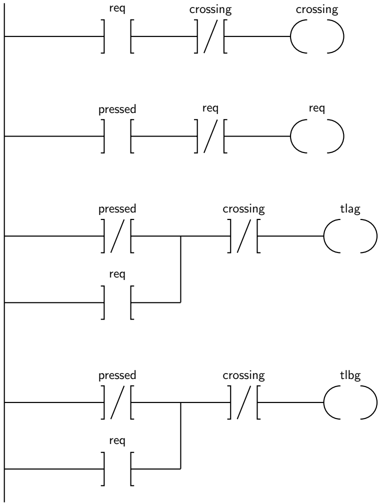
A Simple Pelican Crossing in *SCADE*

Example created/used by Kanso and James

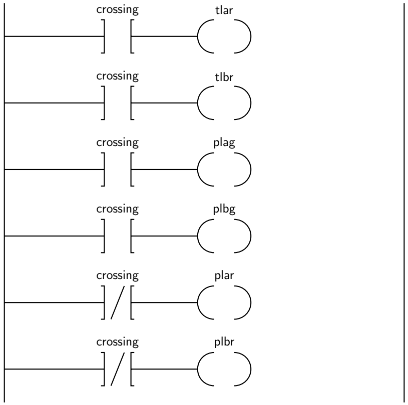
It consists of 4 two aspect lights that control the flow of pedestrians and traffic and a button for pedestrians that indicates that a pedestrian would like to cross the flow of traffic.

- 1 input variable "pressed".
- 2 variables representing an internal state: "crossing" and "required"
- 8 variables representing some external state, 2 variables for each of the 4 lights: "tlag", "tlar", ... "plag", "plar", ...

Pelican Crossing Ladder Logic 1



Pelican Crossing Ladder Logic 2



pre — The pre operator allows us to access the value of a variable used in the previous cycle of a ladder logic program.

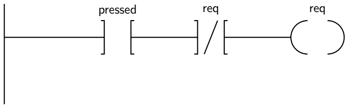
-> — The -> operator allows us to express that a variable has a certain value in the initial cycle of a ladder logic program as well as what it subsequent values depend on.

Example:

```
A = False ->  (not (pre A))
```

Ladder Logic / *SCADE* Comparison

Ladder Logic:



SCADE Language:

```
req = false -> pressed and (not pre req);
```

Model as a node in *SCADE*

```
node PelicanLadderLogic(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
```

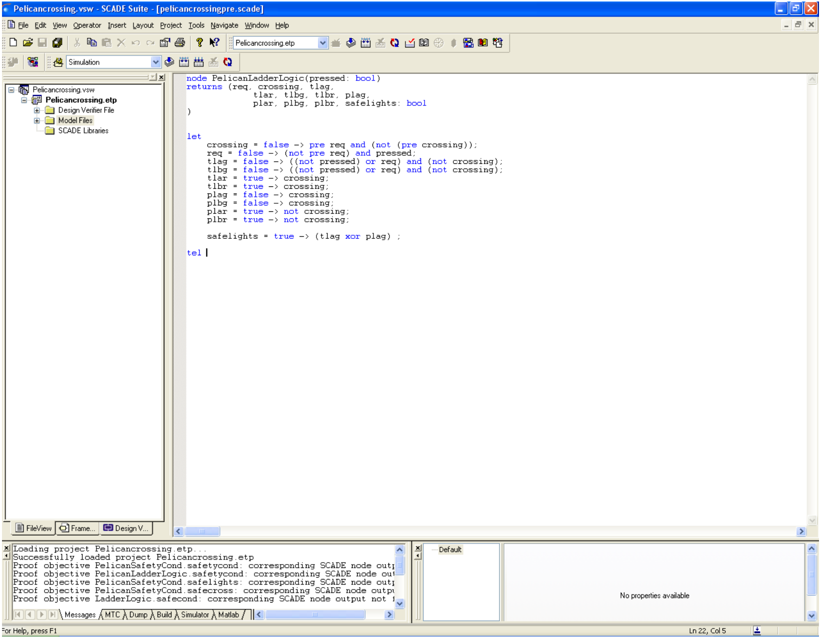
Safety Condition

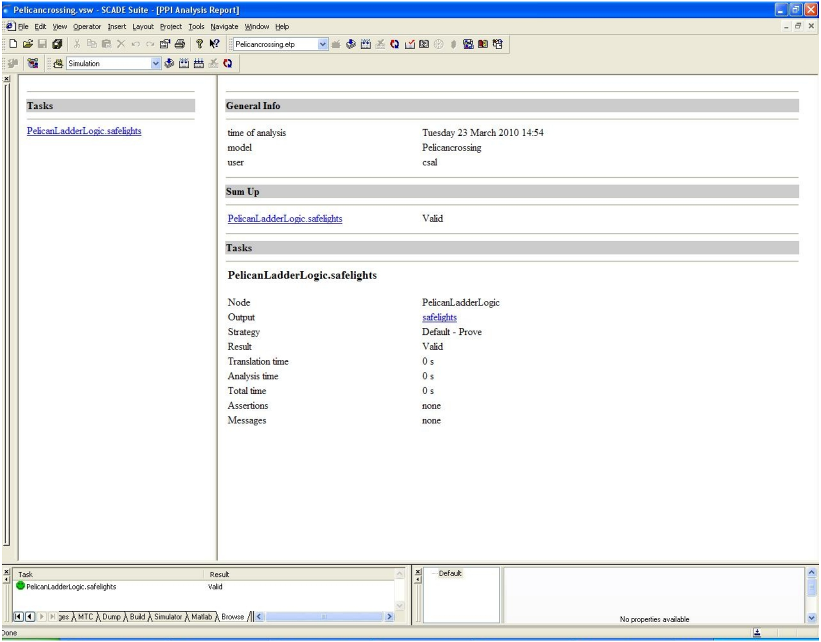
A safety condition for the pelican crossing:

```
safelights = true -> (tlag  xor  plag)
```

It should be the case that either a green light is showing for the traffic or the pedestrians; but never both at the same time.

SCADE will check that the variable `safelights` always has value `true`.


Pelicancrossing.vsw - SCADE Suite - [pelicancrossingpre.scade]
File Edit View Operator Insert Layout Project Tools Navigate Window Help
Pelicancrossing.etp
Simulation
Pelicancrossing.vsw
Pelicancrossing.etp
Design Verifier File
Model Files
SCADE Libraries
node PelicanLadderLogic(pressed: bool)
returns (req, crossing, tlag,
tlar, tlbg, tlbr, plag,
plar, plbg, plbr, safelights: 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;
safelights = true -> (tlag xor plag) ;
tel
FileView
Frame...
Design V...
Loading project Pelicancrossing.etp...
Successfully loaded project Pelicancrossing.etp
Proof objective PelicanSafetyCond.safetycond: corresponding SCADE node out
Proof objective PelicanLadderLogic.safetycond: corresponding SCADE node out
Proof objective PelicanSafetyCond.safelights: corresponding SCADE node out
Proof objective PelicanSafetyCond.safecross: corresponding SCADE node outp
Proof objective LadderLogic.safecond: corresponding SCADE node output not
Messages
MTC
Dump
Build
Simulator
Matlab
Default
No properties available
For Help, press F1
Ln 22, Col 5


Pelicancrossing.vsw - SCADE Suite - [PPI Analysis Report]
File Edit View Operator Insert Layout Project Tools Navigate Window Help
Pelicancrossing.etp
Simulation
Tasks
PelicanLadderLogic.safelights
General Info
time of analysis
Tuesday 23 March 2010 14:54
model
Pelicancrossing
user
csal
Sum Up
PelicanLadderLogic.safelights
Valid
Tasks
PelicanLadderLogic.safelights
Node
PelicanLadderLogic
Output
safelights
Strategy
Default - Prove
Result
Valid
Translation time
0 s
Analysis time
0 s
Total time
0 s
Assertions
none
Messages
none
Task
Result
PelicanLadderLogic.safelights
Valid
MTC
Dump
Build
Simulator
Matlab
Browse
Default
No properties available
Done

Case study: Interlocking A: 331 rungs, 599 variables.

The ladder logic program was automatically translated into the *SCADE* language using a modification of the tool by Kanso and James.

$$Ladder\ logic \underset{Kanso/James/Lawrence}{\overset{Tool\ \ by}{\Longrightarrow}} Scade\ language$$

Verification of a Safety Condition for Interlocking A

We verified a safety condition: "**if** a green light is set **and** a route is selected **then** the green bulb has not blown" .

```
'S1.D' & 'R1(2).RU' -> 'R1(2).UEC'
```

This was formalised as the following in the *SCADE* language.

```
safe =  (not (vS1_D_1 and vR1_2__RU_1)
        or vR1_2__UEC_1);
```

This produces a counter example after 4 steps.

Tasks

[redacted]

General Info

time of analysis	Saturday 20 March 2010 15:06
model	Pelicancrossing
user	csal

Sum Up

[redacted]	Falsifiable

Tasks

[redacted]Logic.safe

Node	[redacted]Logic
Output	safe
Strategy	Default - Prove
Result	Falsifiable
Scenario	[redacted]Logic.safe_s0.sss | **[Load Scenario]**
Translation time	0 s
Analysis time	0 s
Total time	0 s
Assertions	none
Messages	none

Var	1	2	3	4
Inputs				
	true	true	true	false
	false	false	false	false
	false	false	false	false
	false	false	false	false
	true	false	false	false
	false	true	true	false
	false	true	true	false
	true	false	true	false
	true	true	true	false
	true	false	false	false
	false	false	true	false
	true	true	true	false
	false	false	false	false
	false	false	false	false
	false	false	false	false
	true	true	true	false
	true	true	false	false
	true	true	false	false
	false	false	false	false
	false	true	false	false
	false	false	false	false
	false	false	false	false
	false	false	false	false
	false	true	true	false
	true	false	false	false
	false	true	true	false
	true	true	true	false
	false	false	false	false
	false	false	false	false
	false	true	true	false
	false	false	false	false
	false	false	false	false
	false	false	false	false
	false	true	true	false
	false	false	false	false
	true	false	false	false
	false	true	true	false
	true	false	false	false
	false	false	false	false
	false	false	false	false
	false	false	false	false
	false	false	false	false
	true	true	false	false
	false	false	false	false
	true	true	true	false
	false	true	false	false
	true	true	false	false
	false	false	false	false

Comparison with results from previous projects with Invensys (using SAT-solving and different model checkers.) So far no quantifiable results, but our outcomes suggest that *SCADE* is faster.

Comparison with results from previous projects with Invensys (using SAT-solving and different model checkers.) So far no quantifiable results, but our outcomes suggest that *SCADE* is faster.

Analyse applicability of *SCADE*:

- Work so far already demonstrates that *SCADE* can be used in the railway domain.
- **Advantage**: The combination of methods makes it fast.
- **Disadvantage**: Hidden underlying methods make it difficult to trace what is really happening.

Study how to add domain specific knowledge, specifically we need to exclude false negatives (i.e. false counter examples.)

Investigate:

- Limits of Railway Interlocking examples in *SCADE*: How many variables and rungs can *SCADE* handle.
- Further safety conditions and liveness conditions.
- Further functionality of *SCADE*: explore and control other capabilities (eq code generation).
- Is a combination of first order theorem proving and model checking applicable?

Thank you for listening to my talk!