Application of Logic in Railway Verification

Monika Seisenberger

Swansea University
Prifysgol Abertawe

ASL, Boise, 22 March 2017
Specifically, Program extraction from formal proofs:

It is wellknown that from a formal proof we can obtain a program via (Modified) Realisability.

**Proof Term**

**Proof (automatic)**

**Program (automatic)**

**Example (Quotient and Remainder)**

\[ \forall a, b. 0 < b \rightarrow \exists q, r. a = q \cdot b + r \land r < b \].

Proof: Ind(a).

Base \(a = 0\): take \(q = 0, r = 0\).

Step: Given \(q, r\) such that \(a = q \cdot b + r\) find new \(q', r'\) for \(a + 1\).

Program extraction yields a program with

**Input**: two numbers \(a,b\)

**Output**: a pair of numbers \((q,r)\) with the desired behavior.
Specifically, Program extraction from formal proofs:

Example (Quotient and Remainder)

∀ a, b. 0 < b → ∃ q, r. a = q * b + r ∧ r < b

Proof: Ind(a). Base a = 0: take q = 0, r = 0.

Step: Given q, r such that a = q * b + r find new q′, r′ for a + 1

Program extraction yields a program with

Input: two numbers a, b
Output: a pair of numbers (q, r) with the desired behavior.
Specifically, Program extraction from formal proofs: It is wellknown that from a formal proof we can obtain a program via (Modified) Realisability.

Proof (interactive) $\rightarrow$ Proof Term (automatic) $\rightarrow$ Program (automatic)

Example (Quotient and Remainder)

$$\forall a, b. 0 < b \rightarrow \exists q, r. a = q \times b + r \land r < b.$$
Specifically, Program extraction from formal proofs: It is wellknown that from a formal proof we can obtain a program via (Modified) Realisability.

Proof (interactive) $\rightarrow$ Proof Term (automatic) $\rightarrow$ Program (automatic)

Example (Quotient and Remainder)

$$\forall a, b. \ 0 < b \rightarrow \exists q, r. \ a = q \times b + r \land r < b.$$
Specifically, Program extraction from formal proofs: It is wellknown that from a formal proof we can obtain a program via (Modified) Realisability.

**Proof** (interactive) → **Proof Term** (automatic) → **Program** (automatic)

Example (Quotient and Remainder)

∀ \(a, b\). 0 < b → \(\exists q, r. a = q \cdot b + r \land r < b\).

Proof: Ind(a). Base \(a = 0\): take \(q = 0, r = 0\).
Research I: Proof Theory

Specifically, Program extraction from formal proofs: It is wellknown that from a formal proof we can obtain a program via (Modified) Realisability.

Proof (interactive) → Proof Term (automatic) → Program (automatic)

Example (Quotient and Remainder)

\[ \forall a, b. \ 0 < b \rightarrow \exists q, r. \ a = q \times b + r \wedge r < b. \]

Proof: Ind(a). Base \( a = 0 \): take \( q = 0, \ r = 0 \).
Step: Given \( q, r \) such that \( a = q \times b + r \) find new \( q', r' \) for \( a + 1 \)
Specifically, Program extraction from formal proofs: It is wellknown that from a formal proof we can obtain a program via (Modified) Realisability.

Proof (interactive) → Proof Term (automatic) → Program (automatic)

Example (Quotient and Remainder)

\[ \forall a, b. \ 0 < b \rightarrow \exists q, r. \ a = q \times b + r \land r < b. \]

Proof: Ind(a). Base \( a = 0 \): take \( q = 0, \ r = 0 \).
Step: Given \( q, r \) such that \( a = q \times b + r \) find new \( q', r' \) for \( a + 1 \)

Program extraction yields a program with

Input: two numbers \( a, b \)
Output: a pair of numbers \( (q, r) \) with the desired behavior.
Agenda: Establish Program Extraction from Formal Proofs as a Competitive Method for Program Development and Verification

The concrete steps are:

- Extend theory to cover a large range of proofs (→ Inductive and coinductiveDefs, Classical reasoning, Imperative programs,...)
- Demonstrate feasibility via applications in various areas: infinite combinatorics, constructive mathematics, computable analysis, ...
- Demonstrate advantages over other methods of program development and verification: get algorithms for free.
- Provide suitable tool support -all case studies done in Interactive Proof System: Minlog.
The interactive proof assistant: Minlog

- **Formal system** = Heyting Arithmetic in finite types $HA^\omega$
  - Functional term language with structural recursion
  - Intuitionistic logic + Induction
  - Constants, free predicate variables,
  - Inductive types, Inductively defined predicates
  - New: extension to Coinduction

- **Model**: partial continuous functionals in finite types.

- **Proofs** are represented as lambda terms (Curry-Howard)
  - Can be checked,
  - Normalized (Normalisation-by-Evaluation)
  - Manipulated for program development, etc

- **Automatisation** available.
Research II: Logic in Railway Verification

Overview:
1. A semantics for Ladder Logic
   From Ladder Logic to a SAT solving problem.
2. Verification of Interlockings
   A real world example
   Extraction of a verified SAT Solver in the Minlog System.
3. Verification of Train Control Systems
   ERTMS versus PTC
   Generic Modelling: ERTMS in Real-Time Maude
   Simulation & Verification results
Overview:

1. A semantics for Ladder Logic
   - From Ladder Logic to a SAT solving problem.

2. Verification of Interlockings
   - A real world example
   - Extraction of a verified SAT Solver in the Minlog System.

3. Verification of Train Control Systems
   - ERTMS versus PTC
   - Generic Modelling: ERTMS in Real-Time Maude
   - Simulation & Verification results
Ladder Logic

Explanation: Ladder logic is used to develop software for programmable logic controllers; represents a program by a graphical diagram based on the circuit diagrams of relay logic hardware; used in industrial control applications.

Example (Door motor that requires two switches.)

Example (System with start and stop button)
Ladder Logic: a simple Crossing Example

- **pressed**: input variable
- "crossing" and "required": internal state variables
- traffic light variables: tlag, tlar, etc

Read e.g.: if button is pressed and a crossing was currently not requested, then a crossing will be requested in the next cycle.
Translation (automated) to Propositional Logic

\[
\begin{align*}
crossing' & \iff req \land \neg crossing, \\
req' & \iff pressed \land \neg req, \\
tlag' & \iff (\neg pressed \lor req') \land \neg crossing', \\
tlbg' & \iff (\neg pressed \lor req') \land \neg crossing', \\
tlar' & \iff crossing', \\
tlbr' & \iff crossing', \\
plag' & \iff crossing', \\
plbg' & \iff crossing', \\
plar' & \iff \neg crossing', \\
plbr' & \iff \neg crossing'
\end{align*}
\]
Translation (automated) to Propositional Logic

crossing' $\iff$ req $\land \neg$ crossing,

req' $\iff$ pressed $\land \neg$ req,

tlag' $\iff$ $(\neg$ pressed $\lor$ req') $\land \neg$ crossing',

tlbg' $\iff$ $(\neg$ pressed $\lor$ req') $\land \neg$ crossing',

tlar' $\iff$ crossing',

tlbr' $\iff$ crossing',

plag' $\iff$ crossing',

plbg' $\iff$ crossing',

plar' $\iff$ $\neg$ crossing',

plbr' $\iff$ $\neg$ crossing'

- crossing', req',... are new variables, but intended to denote the state of the variable one time step later.
- primed variables on left sides need all to be different.
- a primed variable may depend on earlier computed primed variables, but not on the unprimed ones.
Definition Ladder Logic Formulae

Let $I$ input variables, $C$ output variables, and $C' = \{c' \mid c \in C\}$ to be a set of new variables (intended for output vars computed in the current cycle). In the example:

$I = \{\text{pressed}\}$ and $C = \{\text{req, crossing, tlag, tlbg, plag, plbg, . . .}\}$. 
Definition Ladder Logic Formulae

Let $I$ input variables, $C$ output variables, and $C' = \{c' | c \in C\}$ to be a set of new variables (intended for output vars computed in the current cycle). In the example:

$$I = \{pressed\} \text{ and } C = \{req, crossing, tlag, tlg, plag, plbg, \ldots\}.$$ 

A ladder logic formula $\psi$ is a propositional formula of the form

$$\psi \equiv ((c'_1 \leftrightarrow \psi_1) \land (c'_2 \leftrightarrow \psi_2) \land \ldots \land (c'_n \leftrightarrow \psi_n))$$
Definition Ladder Logic Formulae

Let $I$ input variables, $C$ output variables, and $C' = \{ c' \mid c \in C \}$ to be a set of new variables (intended for output vars computed in the current cycle). In the example:

$I = \{ \text{pressed} \}$ and $C = \{ \text{req}, \text{crossing}, \text{tlag}, \text{tlbg}, \text{plag}, \text{plbg}, \ldots \}$.

A ladder logic formula $\psi$ is a propositional formula of the form

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

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

- $c'_i \in C'$
- $i \neq j \rightarrow c'_i \neq c'_j$ (pairwise distinct)
- $\text{Vars}(\psi_i) \subseteq I \cup \{c'_1, \ldots, c'_{i-1}\} \cup \{c_i, \ldots, c_n\}$
Semantics Ladder Logic Formulae

The semantics of a ladder logic formula \( \psi \) is a function that takes the current valuations for input and output variables

\[
\text{Val}_I = \{ \mu_I \mid \mu_I : I \to \{0, 1\} \}
\]
\[
\text{Val}_C = \{ \mu_C \mid \mu_C : C \to \{0, 1\} \}
\]

and returns a new valuation for output variables (one time cycle later).

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

where

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

and \([\psi_i](\cdot, \cdot, \cdot)\) denotes the usual value of a propositional formula under a valuation. \(\text{unprime} : C' \to C, \text{unprime}(c') = c\).
We define the labelled transition system $\text{LTS}(\psi)$ for a ladder logic formula $\psi$ to be the four tuple $(\text{Val}_C, \text{Val}_I, \rightarrow, \text{Val}_0)$ where

- $\mu_C \xrightarrow{\mu_I} \mu'_C$ iff $[\psi](\mu_I, \mu_C) = \mu'_C$
- $\text{Val}_0 = \{\mu_C | \mu_C \text{ initial valuation}\}$

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 $s_0, \ldots, s_n$, and labels $t_0, \ldots, t_{n-1}$ such that $s_0 \in \text{Val}_0$ and $s_n = s$. 
Crossing transition system

- Included one unreachable state where both crossing and req are true.
Definition (Safety Conditions)

Given a ladder logic formula $\psi$ 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 (The Verification Problem)

We define the verification problem for a ladder logic formula $\psi$ for a verification condition $\phi$

\[
\text{LTS}(\psi) \models \phi
\]

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

Example

In crossing not all lights not green at the same time:

\[\phi = \neg(tlag \land plag)\]
Part 2: Verification of Interlockings

Application to Real World Case Studies

Siemens Rail Automation, formerly Invensys Rail UK, provided Ladder Logic Programs for several stations.
Size: 600 variables, 350 rungs, for a small London Underground Station.
largest case study: 8166 variables, 14726 clauses.

SAT solving using an industrial Tool: SCADE (Prover).
- Tool that translates ladder logic formulae into SCADE language.
- Several optimization methods, no control on methods.
- All 109 safety conditions proven.
- 55 produced counter examples.
- eliminated by added further invariants.
Siemens Automation UK (formerly Invensys Rail UK) provided Ladder Logic Programs for several stations. Size: 600 variables, 350 rungs, for a small London Underground Station. Largest: 8166 variables, 14726 clauses

1. SAT solving using an industrial Tool: SCADE (Prover).

2. Verification via an SAT solver, extracted ourselves in the Minlog system.
   - Formalisation of SAT solver was available in Isabelle/Coq.
   - Extracted SAT solver can easily be integrated in the Minlog system, i.e. will allow for a combination of SAT solving and interactive theorem proving.
   - Provides in each case either a model or a derivation why not satisfiable.
   - Can deal with all the above safety conditions
   - Variant: Extension to backtracking and clause learning (CDCL)
2.1. Extraction of a SAT solving algorithm

Basic definitions:

- A **literal** $l$ is either a positive variable $+v$ or a negative variable $-v$. The **opposite** value of a literal is defined as: $+v = -v$, $-v = +v$.

- A **clause** $C$ is defined as a set of literals $\{l_1, \ldots, l_k\}$ (representing their disjunction).

- A **formula** $\Delta$ is a set of clauses (representing their conjunction).

An example of a formula:

$$\Delta = \{\{l_{11}\}, \{l_{21}\}, \{\overline{l}_{11}, \overline{l}_{21}\}\}$$

to be read as

$$l_{11} \land l_{21} \land (\neg l_{11} \lor \neg l_{21})$$

**SAT problem:** is there a valuation for these variables satisfying the formula?
Most modern SAT solvers are based on DPLL algorithm (Davies, Putnam, Logemann, Loveland 1960/1962).
Part 2: Verification of Interlockings

DPLL Proof System $\Gamma \vdash \Delta$

Most modern SAT solvers are based on DPLL algorithm (Davies, Putnam, Logemann, Loveland 1960/1962).

We use the DPLL proof system, consisting of 5 rules:

1. **Unit**
   \[
   \frac{\Gamma, l \vdash \Delta}{\Gamma \vdash \Delta, \{l\}}
   \]

2. **Red**
   \[
   \frac{\Gamma, l \vdash \Delta, C}{\Gamma \vdash \Delta, (\overline{l}, C)}
   \]

3. **Elim**
   \[
   \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, \overline{l}, C}
   \]

4. **Conflict**
   \[
   \frac{\Gamma \vdash \Delta, \emptyset}{\Gamma \vdash \Delta}
   \]

5. **Split**
   \[
   \frac{\Gamma, l \vdash \Delta}{\Gamma \vdash \Delta}
   \]

Here, $\Gamma$ is a valuation (set of literals) and $\Delta$ is a formula (clause set). $\Gamma \vdash \Delta$ means: there is a proof that $\Delta$ is not satisfiable, using valuation $\Gamma$. 
Valuations and Models

- A valuation $\Gamma$, i.e. set of literals $\{l_1, \ldots, l_k\}$, is consistent iff $l \in \Gamma \rightarrow \overline{l} \notin \Gamma$. Let Cons be the set of all consistent Valuations.

- A *model* is a total function $M$ which maps literals to booleans and satisfies the following property $\forall l. M. l \leftrightarrow \neg(M \overline{l})$

Two abbreviations:

- For a given valuation $\Gamma$, $\forall l \in \Gamma M. l$ is abbreviated as $M \models \Gamma$.
- For a given formula $\Delta$, $\forall C \in \Delta \exists l \in C M. l$ is abbreviated as $M \models \Delta$.

We call a valuation $\Gamma$ and a formula $\Delta$ *compatible* if there exists a model satisfying both, i.e.

$$\exists M. M \models \Gamma \land M \models \Delta$$
Formalising and Proving Completeness

We proved the following classically equivalent but constructively stronger statement:

\[ \text{compatible}(\Gamma; \Delta) \lor \Gamma \vdash \Delta \]

Program extraction yields a program that either yields a model if \( \Gamma \) and \( \Delta \) are compatible (\( \exists M. M \models = \Gamma \land M \models = \Delta \)) or a derivation if incompatible.
Soundness:

\[ \Gamma \vdash \Delta \rightarrow \text{incompatible}(\Gamma; \Delta) \]
Formalising and Proving Completeness

Soundness:
\[ \Gamma \vdash \Delta \rightarrow \text{incompatible}(\Gamma; \Delta) \]

Completeness:
\[ \text{incompatible}(\Gamma; \Delta) \rightarrow \Gamma \vdash \Delta \]
Formalising and Proving Completeness

Soundness:
\[ \Gamma \vdash \Delta \rightarrow \text{ incompatible}(\Gamma; \Delta) \]

Completeness:
\[ \text{incompatible}(\Gamma; \Delta) \rightarrow \Gamma \vdash \Delta \]

We proved the following classically equivalent but constructively stronger statement:

\[ \text{compatible}(\Gamma; \Delta) \lor \Gamma \vdash \Delta \]
Part 2: Verification of Interlockings

Formalising and Proving Completeness

Soundness:

$$\Gamma \vdash \Delta \rightarrow \text{incompatible}(\Gamma; \Delta)$$

Completeness:

$$\text{incompatible}(\Gamma; \Delta) \rightarrow \Gamma \vdash \Delta$$

We proved the following classically equivalent but constructively stronger statement:

$$\text{compatible}(\Gamma; \Delta) \lor \Gamma \vdash \Delta$$

Program extraction yields a program that either yields a model if $\Gamma$ and $\Delta$ are compatible ($\exists M. M \models \Gamma \land M \models \Delta$) or a derivation if incompatible.
Proof of Completeness Theorem

**Theorem**

\[ \forall \Gamma \in \text{Cons}, \forall \Delta, \Theta. \emptyset \notin \Theta \land \text{Var}(\Gamma) \cap \text{Var}(\Theta) = \emptyset \rightarrow \]

\[ (\Gamma \vdash \Delta \cup \Theta) \lor \exists M. M \models \Gamma \land M \models \Delta \cup \Theta, \]

We aim to perform the proof in such a way that an efficient program is extracted:

1. Since performing a split is the only computational expensive operation, we only apply it when it is absolutely necessary.

2. We perform an optimisation on the proof level by partitioning the clauses into 'clean' and 'unclean' clauses, where a clause is called clean if we cannot apply Elim, Reduce or Unit to that clause.
The proof has been formalised in the Interactive Proof System Minlog, and - via modified realisability - a program has been extracted.
Program Extraction - Extracted Solver

The proof has been formalised in the Interactive Proof System Minlog, and - via modified realisability - a program has been extracted.

Example run: We run the extracted solver using pigeon hole formulae

\[ \text{PHP}(n, m) := \{\{l_i,1, \ldots, l_i,m\}|1 \leq i \leq n\} \]

\[ \cup\{\{l_i,k, l_j,k\}|1 \leq i < j \leq n, 1 \leq k \leq m\} \]

Intuitively, e.g. \( \text{PHP}(n, n - 1) \): ”is it possible to put \( n \) pigeons into \( n - 1 \) holes and only have one pigeon in each hole?”
Various improvements:

<table>
<thead>
<tr>
<th>Formula</th>
<th>Minlog $\forall$</th>
<th>Minlog $\forall_{nc}$</th>
<th>Haskell</th>
<th>Haskell ($-fllvm$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Witness</td>
<td>Witness</td>
<td>Yes/No</td>
<td>Witness</td>
</tr>
<tr>
<td>PHP(4,3)</td>
<td>33.62s</td>
<td>11.61s</td>
<td>0.019s</td>
<td>0.006s</td>
</tr>
<tr>
<td>PHP(4,4)</td>
<td>5.45s</td>
<td>5.25s</td>
<td>0.019s</td>
<td>0.010s</td>
</tr>
<tr>
<td>PHP(5,4)</td>
<td>13m54s</td>
<td>2m41s</td>
<td>0.055s</td>
<td>0.020s</td>
</tr>
<tr>
<td>PHP(5,5)</td>
<td>26.09s</td>
<td>25.03s</td>
<td>0.024s</td>
<td>0.015s</td>
</tr>
<tr>
<td>PHP(6,5)</td>
<td>5h35m41s</td>
<td>37m25s</td>
<td>0.367s</td>
<td>0.066s</td>
</tr>
<tr>
<td>PHP(6,6)</td>
<td>1m34.11s</td>
<td>1m24.88s</td>
<td>0.035s</td>
<td>0.025</td>
</tr>
<tr>
<td>PHP(8,8)</td>
<td>-</td>
<td>-</td>
<td>0.054s</td>
<td>0.029s</td>
</tr>
<tr>
<td>PHP(9,8)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1m21.915s</td>
</tr>
<tr>
<td>PHP(9,9)</td>
<td>-</td>
<td>-</td>
<td>0.064s</td>
<td>0.042s</td>
</tr>
<tr>
<td>PHP(10,9)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>102m 16s</td>
</tr>
</tbody>
</table>
Versat was formalized and verified in the dependently typed programming language Guru and translated into C-code.

<table>
<thead>
<tr>
<th>Formula</th>
<th>$\forall_{nc}$ compiled (Yes/No)</th>
<th>Versat</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP(7,6)</td>
<td>0.226s</td>
<td>0.089s</td>
</tr>
<tr>
<td>PHP(8,7)</td>
<td>2.42s</td>
<td>0.794s</td>
</tr>
<tr>
<td>PHP(9,8)</td>
<td>32.062s</td>
<td>17.217s</td>
</tr>
<tr>
<td>PHP(10,9)</td>
<td>15m 5s</td>
<td>15m 46s</td>
</tr>
</tbody>
</table>
Part 3: Verification of Railway Control Systems

In Europe: European Rail Traffic Management System (ERTMS)
In North America: Positive Train Control

Aim: To investigate such a Centralized Traffic Control System, can be modelled and verified using the Real-Time-Maude system

Overview:

I: Centralized Control Systems– how they work
II: Modelling in Real-Time Maude
III: Validation and Verification results
Main Responsibilities:
Trains - communicate position/speed, and receive movement authorities.
RBC - grants MAs/denies MA requests, consults with Interlocking
Interlocking - allows for setting new routes, responsible for safety.
Main Responsibilities:
- Trains - communicate position/speed, and receive movement authorities.
- RBC - grants MAs/denies MA requests, consults with Interlocking
- Interlocking - allows for setting new routes, responsible for safety.
- Controller (not in picture) - requests new routes.
Information flow in ERTMS, level 2

- Controller
  - Acknowledge
  - Route Request / Cancel
- Interlocking
  - Routes Available / Proceed
  - Request to Proceed
- RBC
  - Movement Authority Request
  - Movement Authority
- Track Equipment
  - Track Occupation
  - Point Setting
- Trains
  - Movement Request
  - Guidance/Beacons
Object Oriented Modelling in Real-Time-Maude

- Real-Time Maude (Peter C. Ölveczky and José Meseguer 2004) is a language and tool extending Maude, that allows for simulation and formal analysis of real-time and hybrid systems.

- Object based systems are modelled as multisets of objects and messages of a sort Configuration, a subset of Maude’s built-in in sort System.
Object Oriented Modelling in Real-Time-Maude

- Real-Time Maude (Peter C. Ölveczky and José Meseguer 2004) is a language and tool extending Maude, that allows for simulation and formal analysis of real-time and hybrid systems.

- Object based systems are modelled as multisets of objects and messages of a sort Configuration, a subset of Maude’s built-in in sort System.

- A real-time specification consists of
  - the sort Time (in our case NNegRat),
  - the constructor \( \{\_\} : \text{System} \rightarrow \text{Globalsystem} \)
  - instantaneous rewrite rules,
  - a so-called tick rule that defines how time elapses.
Object Oriented Modelling in Real-Time-Maude

- Real-Time Maude (Peter C. Ölveczky and José Meseguer 2004) is a language and tool extending Maude, that allows for simulation and formal analysis of real-time and hybrid systems.

- Object based systems are modelled as multisets of objects and messages of a sort Configuration, a subset of Maude’s built-in in sort System.

- A real-time specification consists of
  - the sort Time (in our case NNegRat),
  - the constructor \{\_\} : System -> Globalsystem
  - instantaneous rewrite rules,
  - a so-called tick rule that defines how time elapses.

\[
\text{crl \ [tick] : \{CURRENT\} => \{delta(CURRENT,T)\} in time \ T}
\[
\text{if } T <= mte(CURRENT) .
\]

where \text{delta} defines the effect of time elapse on a configuration. 
\text{mte} defines the maximal possible time elapse.
Modelling 1: location specific data & messages

Encoding of the rail topology:

\begin{verbatim}
sort Track . ops AA AB AC ... : -> Track .
sort Point . ops P1 P2 : -> Point .
sort RouteName . ops RouteName1A ... : -> RouteName .
\end{verbatim}

Messages to be exchanged between the ERMTS components:

\begin{verbatim}
msg routerequest : RouteName -> Msg .
msgmarequest : Oid Track -> Msg .
\end{verbatim}
Modelling 2: Instantaneously reacting sub-systems

Interlocking – a class with internal states telling which tracks are occupied, which routes and points are set,…

Here, ignoring a route request:

\[
\text{crl routerequest}(\text{RN1}) \quad < 0 : \text{Inter} \mid \text{occ} : \text{MAPTB1}, \text{pointslocked} : \text{MAPPB3} > \\
\quad => \\
\quad < 0 : \text{Inter} \mid > \text{if (not checkClear(RN1, MAPTB1)) or pointsLocked(RN1, MAPPB3)} .
\]
Modelling 3: Trains with ERTMS equipment

```latex
ceq delta(<O : Train | state : acc,
  dist : DT,
  speed : S,
  ac : A,
  ma : MA, ..... >, T)
=>

< O : Train | state : if (S + T * A == MAX)
  then cons
  else (if T == mteMA(DT,S,A,MA)
    then brake
    else acc fi) fi,
  dist : DT + S * T + A * T * T * 1/2,
  speed : S + A * T > ) .
```
Validation through Simulation

We have validated our model through exploring various train movements.

For example, rewriting a train starting on track AA:

\[
\text{(trew } \{ < \text{inter1} : \text{Inter} | \text{pointPositions} : (P1 \rightarrow \text{normal}, P2 \rightarrow \text{normal}), ... \} < \text{train1} : \text{Train} \mid \text{state} : \text{acc}, \text{dist} : 2, \text{speed} : 0, \text{ac} : 1, \text{ma} : 1498, \text{tseg} : \text{AA}, \text{maxspeed} : 60 \} \text{ in time } \leq 39 .)\]

shows that it accelerates until it is required to begin braking due to its MA:

\[
... < \text{train1} : \text{Train} \mid \text{ac} : 1, \text{dist} : 1499446241/2000000, \text{ma} : 1498, \text{maxspeed} : 60, \text{speed} : 38671/1000, \text{state} : \text{brake}, \text{tseg} : \text{AA} >... \text{ in time } 38671/1000
\]
Validation through Simulation (2)

It then makes a movement authority request:

```plaintext
marequest(train1, AA)
< inter1 : ...>
< train1 : Train | speed : 37671/1000, ... >
in time 39671/1000
```

However at this point the system will not progress until we add an RBC to deal with the request...
Our modelling is able to find errors, for example:

Decelleration for used for computation: 1; physical deceleration: 8/10.

< train1 : Train | ... dist : 3249, ac : 1, ma : 6499, tseg : AD , maxspeed : 20 > < train2 : Train | ... ac : 8/10, ma : 1, tseg : Entry , maxspeed : 60 > ...

Model checking is able to produce a counter example

Defining collision freedom

\[
eq \{ \text{REST} \\
\quad < \text{train1} : \text{Train} \mid \text{tseg} : \text{T1}, \text{dist} : \text{DT1}, \text{mb} : \text{MBA} > \\
\quad < \text{train2} : \text{Train} \mid \text{tseg} : \text{T2}, \text{dist} : \text{DT2}, \text{mb} : \text{MBB} > \} \\
\implies \text{nocrashDistance(train1, train2)} \\
= \\
\quad (\ (\ \text{not (T1 == Entry)} \ \text{and} \ \text{not (T2 == Entry)} \ \text{and} \\
\quad \text{not (T1 == Exit)} \ \text{and} \ \text{not (T2 == Exit)} \ ) \\
\quad \text{and (T1 == T2 or} \\
\quad \quad \text{T1 == next(T2, mb : MBB, normal) or} \\
\quad \quad \text{T1 == next(T2, mb : MBB, reverse) or} \\
\quad \quad \text{T2 == next(T1, mb : MBA, normal) or} \\
\quad \quad \text{T2 == next(T1, mb : MBA, reverse)} \ )) \\
\text{implies} \\
\quad \text{distance(DT1, MB1, DT2, MB2) >= minDist} .
\]
Safety Verification through Model-checking

Verification that trains cannot be within minDist metres of each other:

\[
\text{mc initState |=t [] nocrashDistance(train1,train2)} \quad \text{in time } \leq 300.
\]

<table>
<thead>
<tr>
<th>Scheme Plan</th>
<th>Round Robin Controller Unbounded</th>
<th>Random Controller in Time 300</th>
</tr>
</thead>
<tbody>
<tr>
<td>Junction</td>
<td>2.4s / 5,767,435 rewrites</td>
<td>268.3s / 208,715,358 rewrites</td>
</tr>
<tr>
<td>Pass-through Station</td>
<td>3.0s / 7,135,987 rewrites</td>
<td>439.2s / 308,629,500 rewrites</td>
</tr>
<tr>
<td>Three Platform Station</td>
<td>2.8s / 6,624,578 rewrites</td>
<td>2697.1s / 729,201,878 rewrites</td>
</tr>
</tbody>
</table>

**Table**: Verification results of model checking three scheme plans.
Completeness

**Head-to-head**

\[ \text{DistA} < \text{Pos(MBA)} \quad \text{DistB} < \text{Pos(MBB)} \]

**Head-to-tail**

\[ \text{DistA} < \text{Pos(MBA)} \quad \text{DistB} < \text{Pos(MBB)} \]
Conclusion: Extraction of a Sat Solver

1. We presented a conceptually new approach to the synthesis and verification of SAT algorithms.
   - does not require the formalisation of the algorithm, but obtains it by program extraction.
   - interesting point: do optimisations not on the programme level, but on the proof level.
   - Future work: Extension to include backtracking and clauselearning.

2. Application of Logic to Traditional Interlockings: all translation processes can be automated; method included in industrial process between design and testing. Industry will still do testing, but the burden of guaranteeing is completeness and correctness greatly reduced.
Conclusion: Railway Verification

Summary:

- Semantics for Ladder Logic Programs
- First use of Real-Time Maude in the railway domain.
- First formal model comprising of all ERTMS subsystems required for the control cycle.
- Rail control modelled as a hybrid system,
- Safety properties in physical rather than in logical terms.
Conclusion: Railway Verification

Summary:
- Semantics for Ladder Logic Programs
- First use of Real-Time Maude in the railway domain.
- First formal model comprising of all ERTMS subsystems required for the control cycle.
- Rail control modelled as a hybrid system,
- Safety properties in physical rather than in logical terms.

Future work:
- Improving models: Further controller strategies.
- More complex train progression behaviour.
- Include further safety properties.
- Develop abstractions to increase in verification speed.
- Proof: Sufficient to consider two trains only.
References

Berger, U., Lawrence, A., Nordvall Forsberg, F., Seisenberger, M. 
Extraction of Verified Decision Procedures. 
LMCS 11(1:6), 2015.

James, P., Lawrence, A., Moller, F., Roggenbach, M., Seisenberger, M. and Setzer, A. Chadwick, S., P. Kanso, K., 
Verification of solid state interlocking programs. 
In SEFM’13, LNCS 8368, Springer 2014.

James, P., Lawrence, A., Roggenbach, M., Seisenberger, M. 
Towards Safety Analysis of ERTMS/ETCS Level 2 in Real-Time Maude. 

U. Berger, James, P., Lawrence, A., Roggenbach, M., Seisenberger, M., 
Verification of the European Rail Traffic Management System in Real-Time Maude. SCP, submitted.