

Developing user propagators for graph-based SMT reasoning

Alec Critten
Swansea University

(supervisors: Dr. Monika Seisenberger, Dr. Anton Setzer)

BCTCS 2025
University of Strathclyde, Glasgow, Scotland

15 April 2025

Overview

- 1 Background
- 2 A tale of two solvers
- 3 Introducing the propagator
- 4 Propagation for graphs
- 5 Conclusion

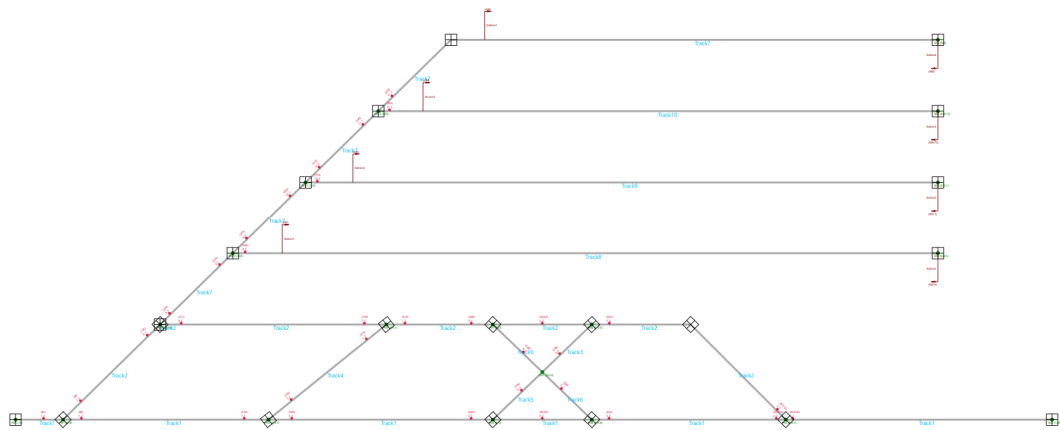
Project Aims

- Improve the **formal verification process** of geographic scheme data for railways
- Develop a custom **graph theory** for the Satisfiability Modulo Theories (SMT) solver Z3
- Integrate graph theory support into a tool-chain for scheme plan verification [1]

Two key definitions

- *Scheme plans* are formal geographic representations of railway systems.
- *Design rules* specify safety-critical constraints which must be verified to hold in the scheme plan.

A scheme plan visualisation

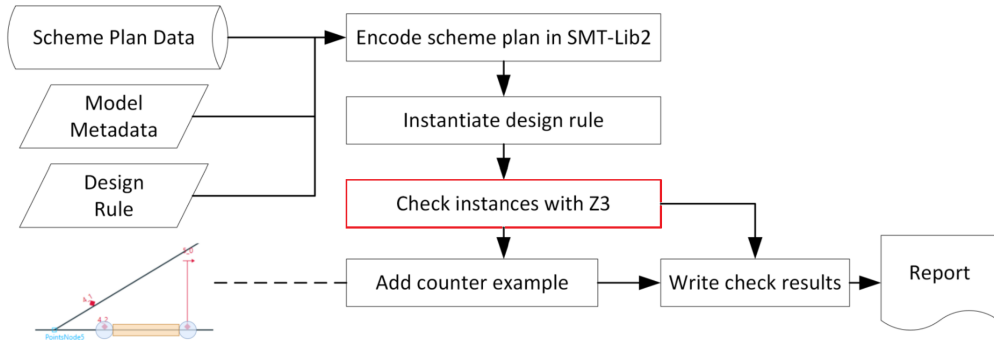


A sample design rule

*The BG-03 design rule*³ states that design placement of balises should avoid points and crossings. Designed spacing shall be constrained by:

1. $\geq 1.0\text{m}$ between balise and point toes.
2. $\geq 1.0\text{m}$ between balise and point frog.
3. $\geq 1.4\text{m}$ lateral separation between a balise on one path and the centre line of the other path.
4. No balises between the toe and frog of set of points.

The verification tool-chain [1]



What is SMT?

Satisfiability Modulo Theories (SMT):

- generalises propositional satisfiability (SAT) to first-order logic
- uses decision procedures to check satisfiability with respect to background theories

Why SMT?

- Industry demand
- Fully automated theorem proving
- A vision for an expressive language w.r.t. scheme plan verification

On to the modelling!

Approach 1: Modelling with MonoSAT

We started with the MonoSAT SMT solver [2] (with some customisation) because of

- its status as the only known SMT solver which inherently supports graph structures
- its Python interface which allows graph construction in the naïve sense

“Small-Happy” encoding in MonoSAT

```
25  RBCN11 = small_happy.addNode("SimpleNode", 5340, "RBCN11")
26  RBC_N111 = small_happy.addNode("SimpleNode", 5342, "RBC_N111")
27  RBC_N2 = small_happy.addNode("SimpleNode", 5969, "RBC_N2")
28
29  simpleNodes = [RBC_S2, RBC_S1, EN_3635, RBC_N1, RBCN11, RBC_N111, RBC_N2]
30
31  # points nodes
32  PN_Extra = small_happy.addNode("PointsNode", 400, "PN_Extra")
33
34  points_nodes = [PN_Extra]
35
36
37  # balises
38  balise_16_0 = small_happy.addNode("Balise", 4348, "balise_16_0")
39  balise_15_0 = small_happy.addNode("Balise", 4386, "balise_15_0")
40  balise_14_0 = small_happy.addNode("Balise", 4430, "balise_14_0")
```

```

105 #####
106 # Translating BG-05 into MONOSAT #
107 #####
108
109 #Designed minimum spacing between adjacent balise groups shall be constrained by:
110 #1.  $\geq$  MIN_BG_SEPARATION between adjacent end balises, one at each end of the two groups.
111 MIN_BG_SEPARATION = 20
112
113 # sort the list of balises
114 balises_sorted = small_happy.sort_nodes(balise_set)
115 print(balises_sorted)
116
117 # adjacency approach
118 for i in range(len(balises_sorted)):
119     # check if two end-balises are adjacent
120     if (small_happy.get_adjacent_balises(balises_sorted[i]) != None):
121         if Solve([Not(small_happy.distance_lt(balises_sorted[i], small_happy.get_adjacent_balises(balises_sorted[i]), MIN_BG_SEPARATION))]) == False:
122             print("BG-05 not satisfied")
123
124         Assert(Not(small_happy.distance_lt(balises_sorted[i], small_happy.get_adjacent_balises(balises_sorted[i]), MIN_BG_SEPARATION)))
125
126 print("BG-05 checked")

```

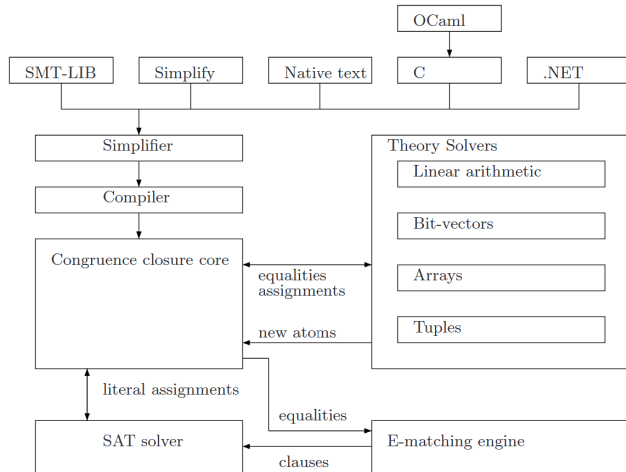
Results from MonoSAT

- Encoded a scheme plan with approx. 50 nodes (“Small-Happy”)
- Encoded and instantiated several design rules
- Performed design rule verification in MonoSAT, specifically for balise spacing
- Automated the translation process from scheme plan to graph for MonoSAT

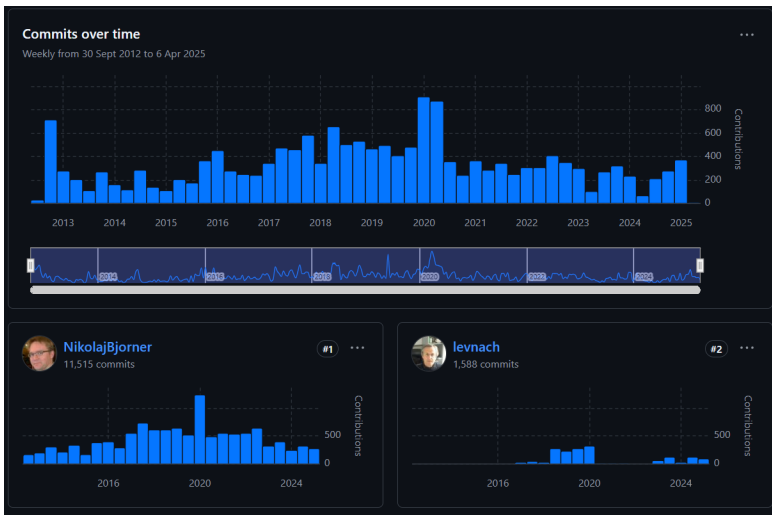
Why pivot away from MonoSAT?

- Bespoke solver not compatible with Z3/general SMT
- Black-box functionality below API
- Difficult to ensure correctness of verification for large scheme plans

Z3 is a deeply complicated system... [3]



...only well-known by a few experts...



...but there is a solution!

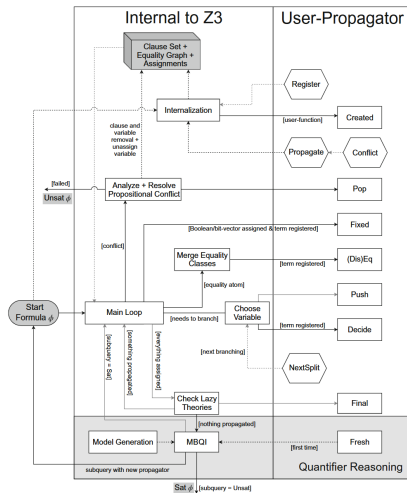
Introducing: user propagators!

What is a user propagator?

A **user propagator** (Eisenhofer et. al. 2023) [4]:

- is an **interface** for customised decision procedures
- uses the **EUF** (equality with uninterpreted functions) theory
- operates in an external code-base, exposing **callbacks** to override Z3 behaviour

User propagators with Z3 architecture [4]



User propagator callbacks

Some callbacks of interest:

- `Created` - instantiate expressions to be eventually assigned to true/false
- `Push/Pop` - save/revert the state of the SAT solver when it branches on a decision, particularly so we can backtrack
- `Fixed` - when the solver decides an expression is true/false
- `Final` - when the solver has assigned all expressions to true/false

The EUF theory

- Always included in SMT
- Everything is **uninterpreted** - just symbols
- **Congruence closure** to determine satisfiability
- Only $=$ comes preloaded with meaning!
- edge is a **user-function**

The interplay between SAT and theory solvers

Suppose we have $\phi = (x < 3) \wedge (x > 4)$. Obviously there is no such x , but the solver deduces this way:

- 1 **Boolean abstraction:** First, the solver abstracts the first-order formula into propositional logic, so

$$\underbrace{(x < 3)}_A \wedge \underbrace{(x > 4)}_B$$

The interplay between SAT and theory solvers, cont.

- ② **Boolean assignment:** The SAT solver makes a decision to set $A \rightarrow \top$ and $B \rightarrow \top$.

The interplay between SAT and theory solvers, cont.

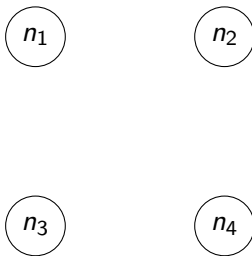
- ③ **Theory symbols:** SMT solver observes $x < 3$ and $x > 4$ on a purely symbolic level, then determines they belong to the theory of integer arithmetic (LIA).

The interplay between SAT and theory solvers, cont.

- ④ **SAT modulo LIA:** Solver checks satisfiability of $A \wedge B$ w.r.t. LIA and returns **unsat**.

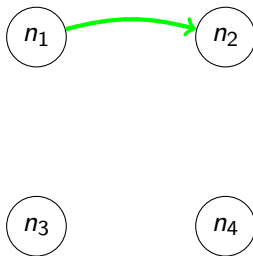
Visualising a graph propagation

A simple propagator enforcing the transitive closure relation over a graph
 $G = \{n_1, n_2, n_3, n_4\}$:



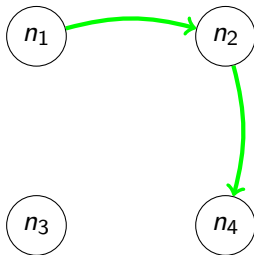
Visualising a graph propagation, cont.

If, for example, we are given that $n_1 \rightarrow n_2 \dots$



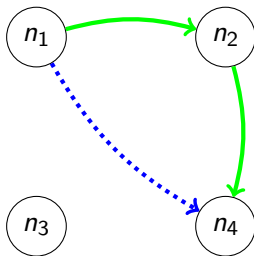
Visualising a graph propagation, cont.

...and $n_2 \rightarrow n_4$...



Visualising a graph propagation, cont.

...then the solver will propagate that $n_1 \rightarrow n_4$:



A comment on transitivity

Constructing the propagator for even a simple relation like TC is not trivial!

Merely programming $(a \rightarrow b) \wedge (b \rightarrow c) \Rightarrow (a \rightarrow c)$ doesn't work.

The key to transitivity is handling the **permutations** of the order of the edges!

Antireflexivity and symmetry

Unlike transitivity, these are easy to model:

- **Antireflexivity**: for any propagation, check first that the two nodes are distinct
- **Symmetry**: given a propagating edge $a \rightarrow b$:
 - Check if $b \rightarrow a$ is assigned.
 - If not, propagate $b \rightarrow a$ with the single justification $a \rightarrow b$.

.smt2 input (non-sym TC)

```
1  (declare-const n1 Node)
2  (declare-const n2 Node)
3  (declare-const n3 Node)
4  (declare-const n4 Node)
5
6  (assert (edge n1 n2))
7  (assert (edge n2 n3))
8  (assert (edge n3 n4))
9
10 (check-sat)
```

Custom model generation for graphs...

Graph model (16 assignments in total):

```
[  
(edge n1 n1) -> false  
(edge n1 n2) -> true  
(edge n1 n3) -> true  
(edge n1 n4) -> true  
(edge n2 n1) -> false  
(edge n2 n2) -> false  
(edge n2 n3) -> true
```

```
(edge n2 n4) -> true  
(edge n3 n1) -> false  
(edge n3 n2) -> false  
(edge n3 n3) -> false  
(edge n3 n4) -> true  
(edge n4 n1) -> false  
(edge n4 n2) -> false  
(edge n4 n3) -> false  
(edge n4 n4) -> false  
]
```

...instead of the default Z3 model

```
MODEL FROM Z3:
;; universe for Node:
;;   Node!val!0 Node!val!1 Node!val!3 Node!val!2
;;   -----
;; definitions for universe elements:
(declare-fun Node!val!0 () Node)
(declare-fun Node!val!1 () Node)
(declare-fun Node!val!3 () Node)
(declare-fun Node!val!2 () Node)
;; cardinality constraint:
(forall ((x Node))
  (or (= x Node!val!0) (= x Node!val!1) (= x Node!val!3) (= x Node!val!2)))
;; -----
(define-fun n4 () Node
  Node!val!3)
(define-fun n1 () Node
  Node!val!0)
(define-fun n2 () Node
  Node!val!1)
(define-fun n3 () Node
  Node!val!2)
END OF MODEL
```

Logging propagation steps

List of unique propagations:

1: (edge n1 n2) = true, derived from input assumption

2: (edge n2 n3) = true, derived from input assumption

3: (edge n1 n3) = true, derived from (ast-vector
 (edge n2 n3)
 (edge n1 n2))

Weighted edges

We can also model weighted edges as well!

- Similar propagation process to before
- Extend the edge predicate to also take an integer-sort argument

Building paths

A path predicate $(\text{path } n1 \ n2 \ l)$ is defined inductively:

- $(\text{edge } n1 \ n2 \ l) \rightarrow (\text{path } n1 \ n2 \ l)$
- $(\text{path } n1 \ n2 \ l) \wedge (\text{edge } n2 \ n3 \ l') \rightarrow (\text{path } n1 \ n3 \ (l + l'))$

The path propagator: input

```
1  (declare-const n1 Node)
2  (declare-const n2 Node)
3  (declare-const n3 Node)
4  (declare-const n4 Node)
5
6  (assert (edge n1 n2 10))
7  (assert (edge n2 n3 5))
8  (assert (edge n3 n4 4))
9
10 (check-sat)
11 (get-model)
```


The path propagator: output

```
Graph model (9 assignments in total):  
[  
  (edge n1 n2 10) -> true  
  (edge n2 n3 5) -> true  
  (edge n3 n4 4) -> true  
  (path n1 n2 10) -> true  
  (path n1 n3 15) -> true  
  (path n1 n4 19) -> true  
  (path n2 n3 5) -> true  
  (path n2 n4 9) -> true  
  (path n3 n4 4) -> true  
]
```

Naïve shortest path

- Current work
- Naïve approach: generate all paths and take shortest at Final

Naïve shortest path: input/output

```
1 (declare-const n1 Node)
2 (declare-const n2 Node)
3 (declare-const n3 Node)
4
5 (assert (edge n1 n3 10))
6 (assert (edge n1 n2 1))
7 (assert (edge n2 n3 2))
```

```
Graph model (10 assignments in total):
[
  (edge n1 n2 1) -> true
  (edge n1 n3 10) -> true
  (edge n2 n3 2) -> true
  (path n1 n2 1) -> true
  (path n1 n3 10) -> true
  (path n1 n3 3) -> true
  (path n2 n3 2) -> true
  (shortest-path n1 n2 1) -> true
  (shortest-path n1 n3 3) -> true
  (shortest-path n2 n3 2) -> true
]
```

Next steps





Currently working on a graph user propagator for Z3 with verified algorithms. Current and future work planned for this year:

- **Optimising** path-generation over graphs
- **Applying** the path propagator to larger, realistic graphs
- **Integrating the propagator** with our partner's graph tools
- **Verifying scheme plans** with the railway-oriented propagator

Summary

- Develop **graph theory** for scheme plan verification
- **User propagators** to customise Z3
- Program **decision procedures** for graph traversal
- **Groundwork** for expressing design rule constraints in SMT

References

-  Banerjee, M., Cai, V., Lakshmanappa, S., Lawrence, A., Roggenbach, M., Seisenberger, M., Werner, T.: A Tool-Chain for the Verification of Geographic Scheme Data. RSSRail 2023. LNCS 14198. Springer (2023).
-  Bayless, S., Bayless, N., Hoos, H., Hu, A.: SAT Modulo Monotonic Theories. AAAI Conference on Artificial Intelligence, **29**(1), (2015).
-  de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. TACAS 2008, LNCS 4963. Springer (2008).
-  Bjørner, N., Eisenhofer, C., Kovács, L.: Satisfiability Modulo Custom Theories in Z3. VMCAI 2023. LNCS 13881. Springer (2023).

Thanks for listening!

