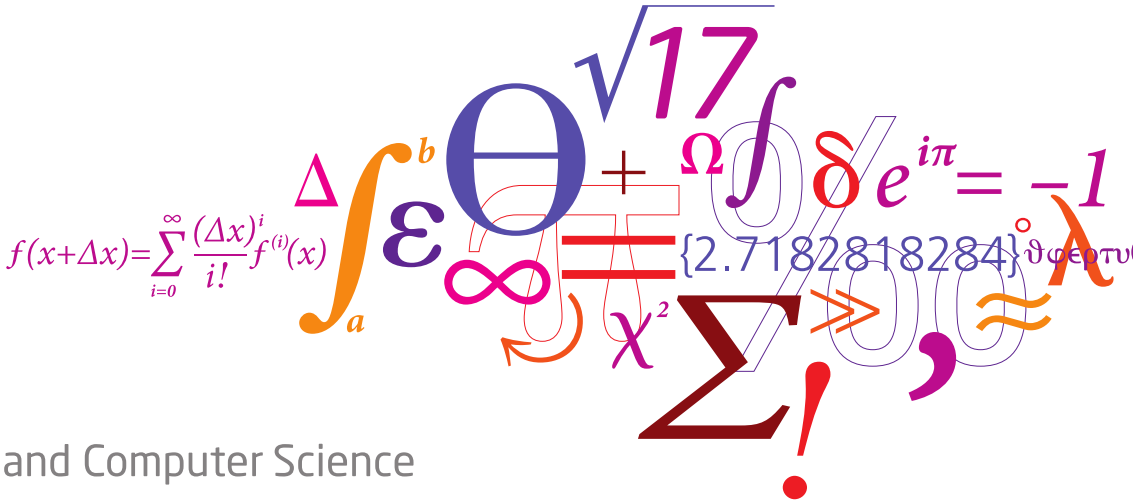
Automated Compositional Verification of Interlocking Systems

Anne E. Haxthausen[1], Alessandro Fantechi[2], Gloria Gori[2], Óli Kárason Mikkelsen[1], and Sofie-Amalie Petersen[1]

[1] DTU Compute, Technical University of Denmark

[2] DINFO, University of Florence



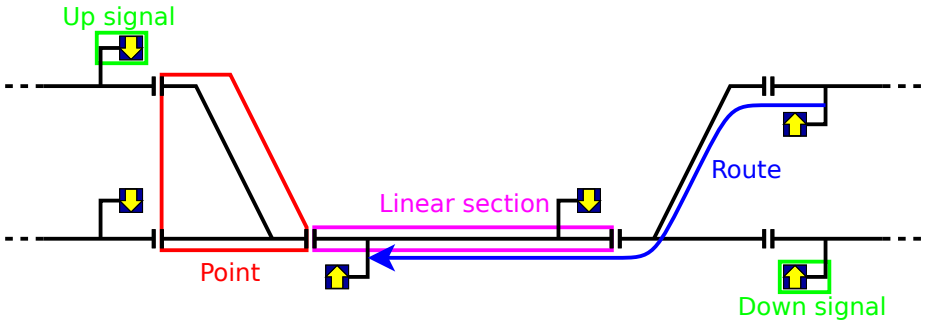
DTU Compute

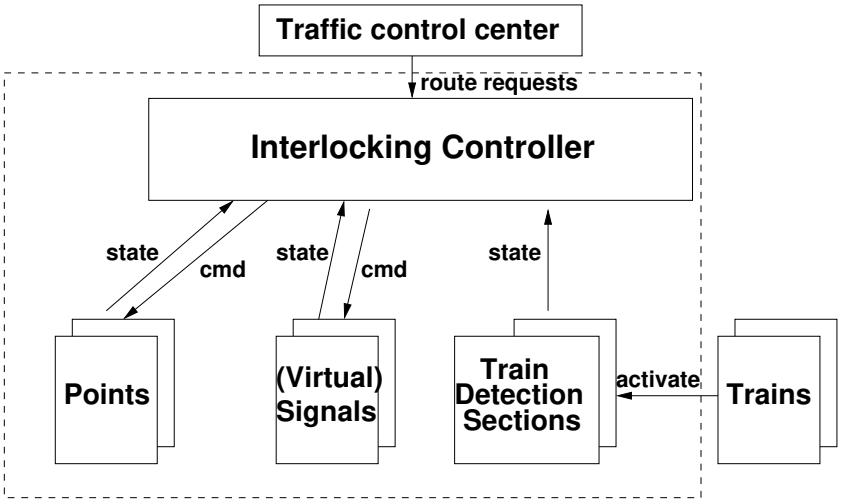
Department of Applied Mathematics and Computer Science

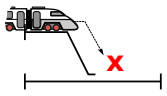
Outline

1. Introduction
2. Compositional Verification
3. A Strategy for Automated Decomposition
4. Formal Specification and Implementation of a Decomposer
5. Experimental Results
6. Conclusions

Principles of Interlocking Systems

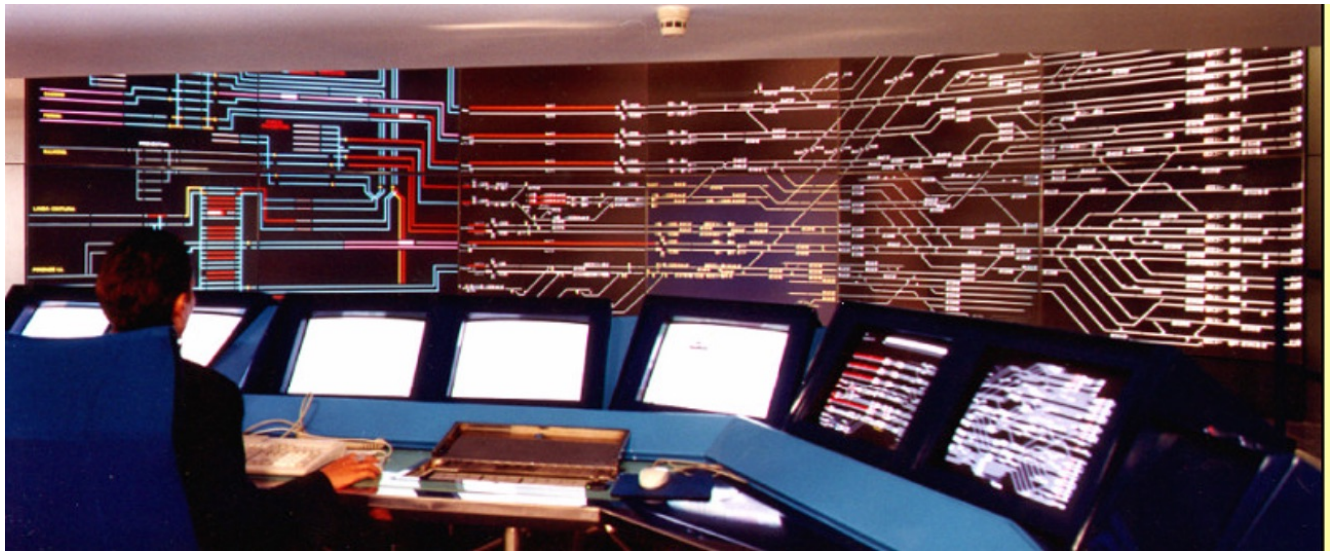




- The task of an **interlocking system** IXL is to ensure *safe* routing of trains in the railway network under its control.
  - safe = *no collisions*  and *no derailments* 
- Tracks are divided into train detection sections.
- IXL reserves routes for exclusive use by a single train.
- IXL switches and locks points in correct positions for routes when reserving them.
- IXL uses (virtual) signals to regulate the train access to sections.

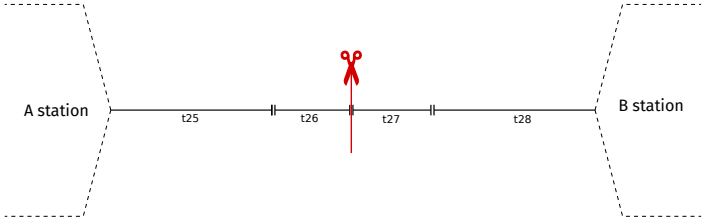
The Challenge of Model Checking Large Interlockings

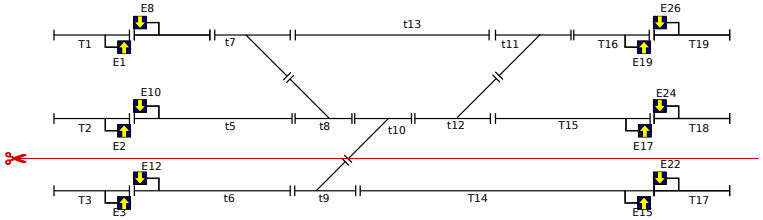
- Interlockings are *safety-critical* systems (SIL4).
- *CENELEC 50128* strongly recommends formal verification for SIL4 software.
- *Model checking* has raised interest as it is *fully automated*.
- However, for large interlocking systems it typically incurs in *state space explosion* making it infeasible to reach a verification result.
- *"Large"* is often referred to the number of physical elements in the track layout and to the number of routes that are defined on the track layout.



Can Compositional Verification be Used?

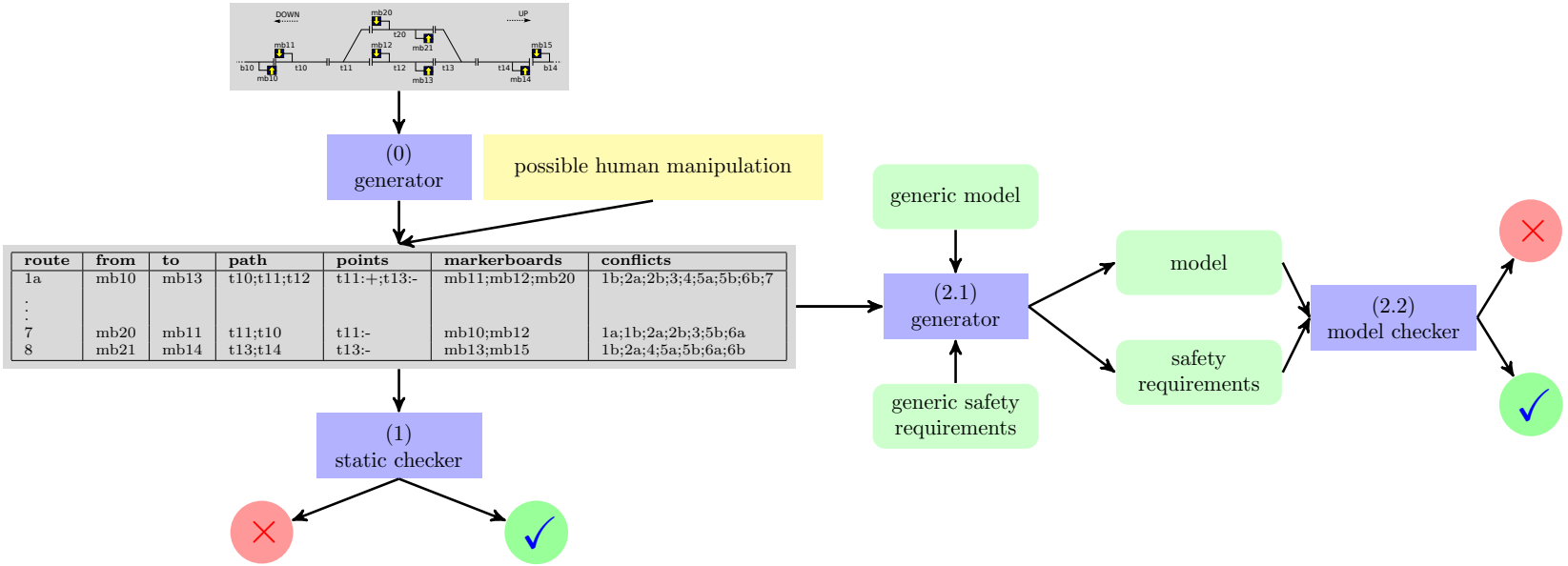
- Can we address the verification challenge by *decomposing* the interlocking logic of large layouts *into separate, more manageable, parts*, so that proving safety of the parts implies safety of the whole?





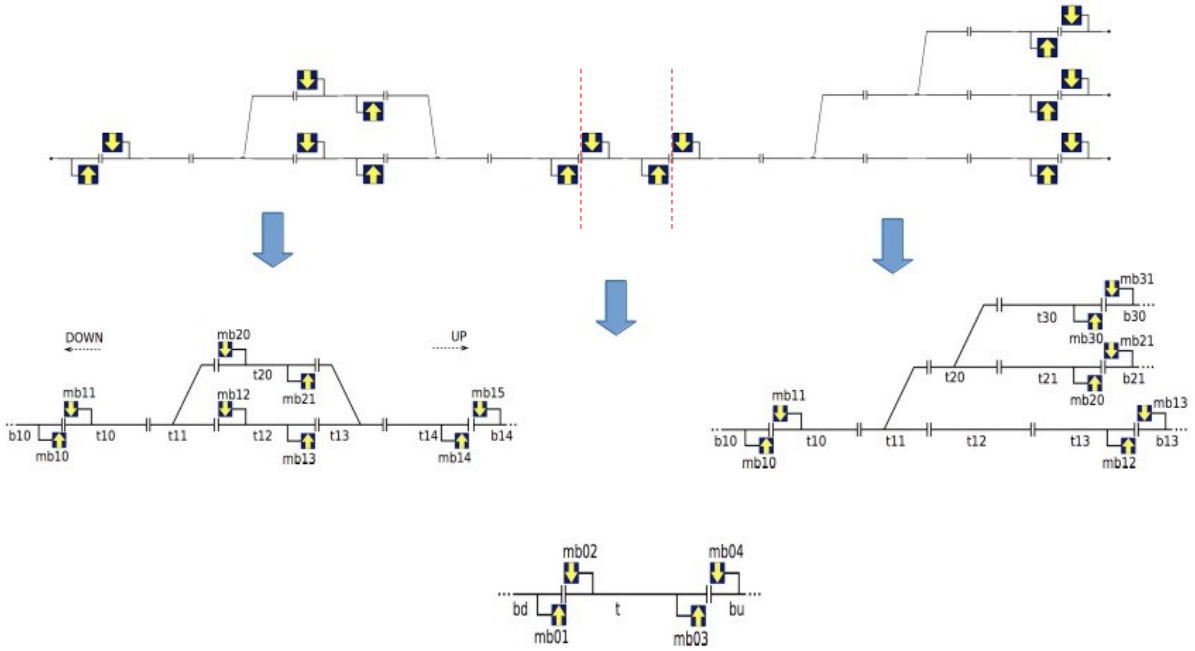
- To answer this, we have developed a *compositional method* and implemented it for the **RobustRailS** interlocking verification environment.

RobustRailS Tool Chain for Interlocking Verification[1]



[1] The method and tools were developed by Anne Haxthausen, Jan Peleska, and Linh H. Vu in collaboration with the Danish railways in the RobustRailS project, 2012-2017, `http://www.robustrails.man.dtu.dk/`

A Method for Compositional Verification



1. Decompose the network into n sub-networks by applying $n-1$ *valid* network cuts.
2. Verify each sub-network using the RobustRailS tools verification steps.

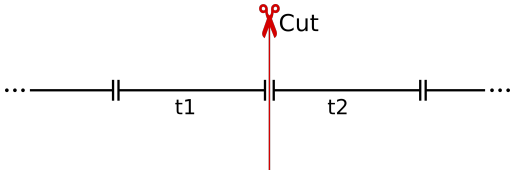
This method is *sound* and *complete*:

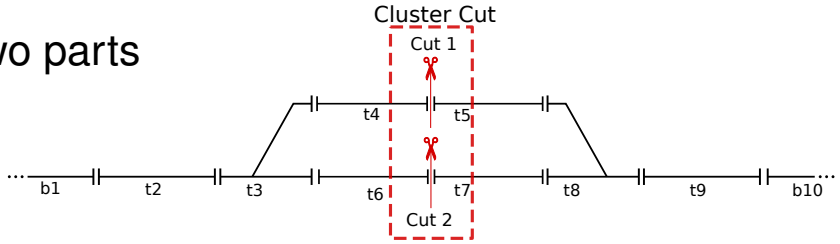
Haxthausen & Fantechi. Compositional verification of railway interlocking systems. FAC 35 (1), 2023

Valid Network Cuts

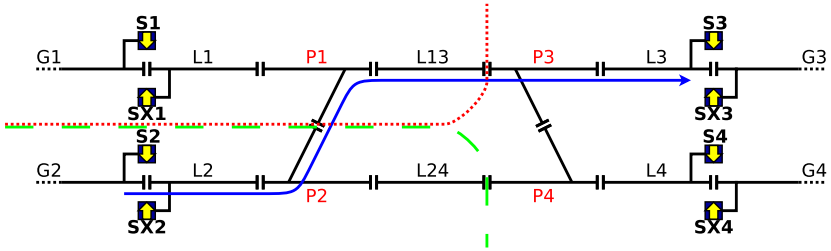
A cut is *valid* if:

- it divides the network into exactly two parts

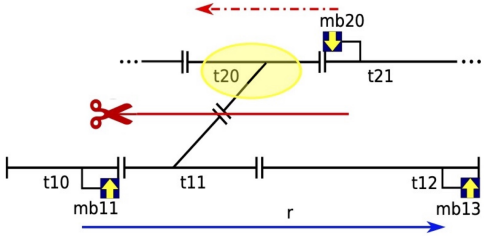




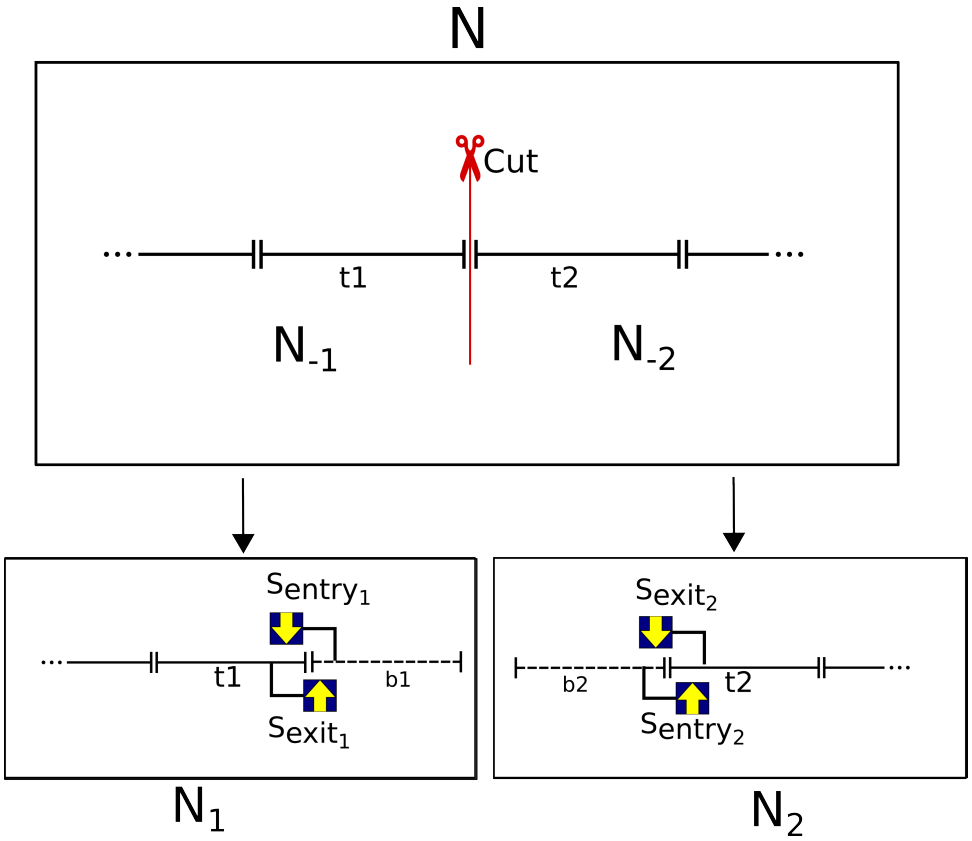
- no route is divided into more than two parts



- no flank protecting elements are separated by the cut from the sections they protect



Decomposition by Applying a Valid Cut



A Strategy for Automated Decomposition

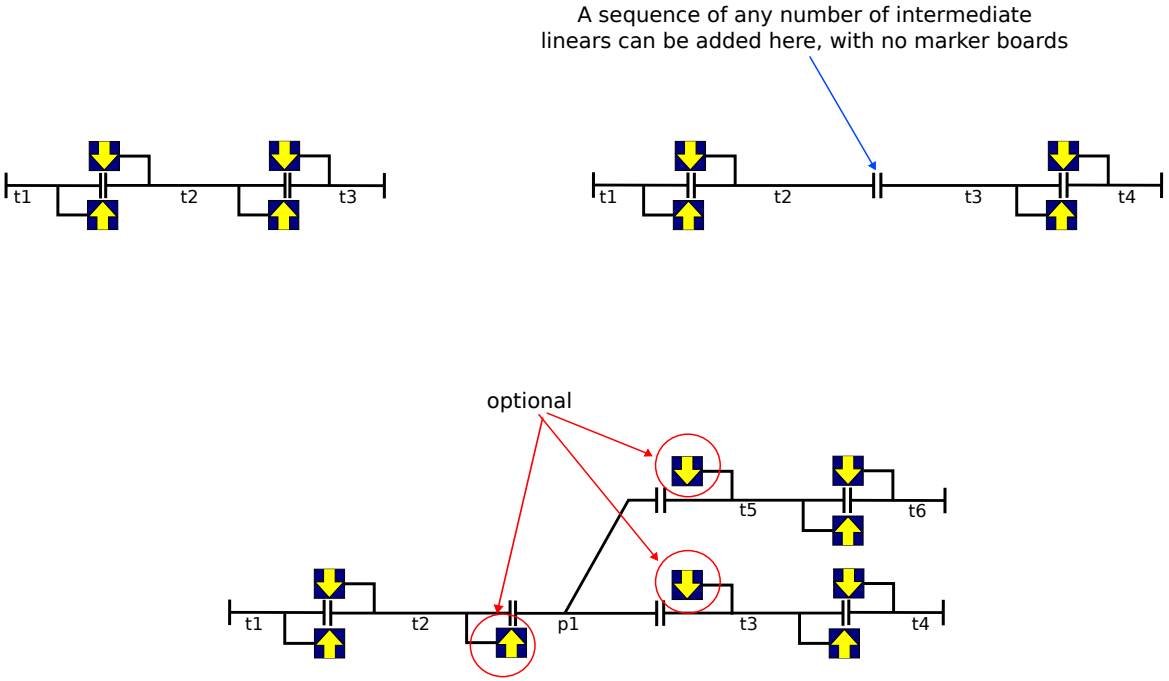
Question:

- Which cuts should be made?

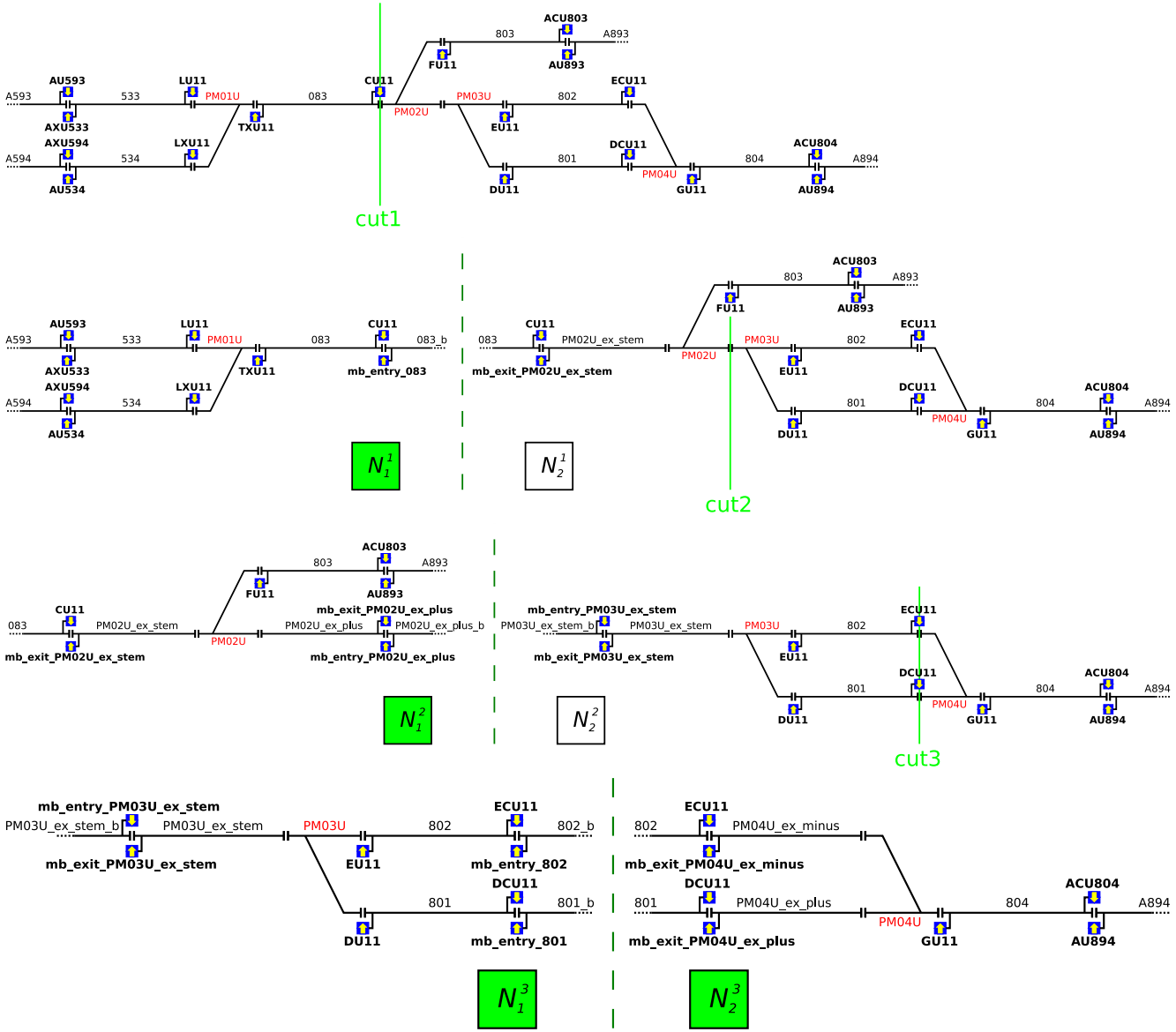
Solution Idea:

- Provide a library of pre-verified, elementary networks.
- Devise an algorithm for decomposing a network into elementary sub-networks.

Elementary Networks

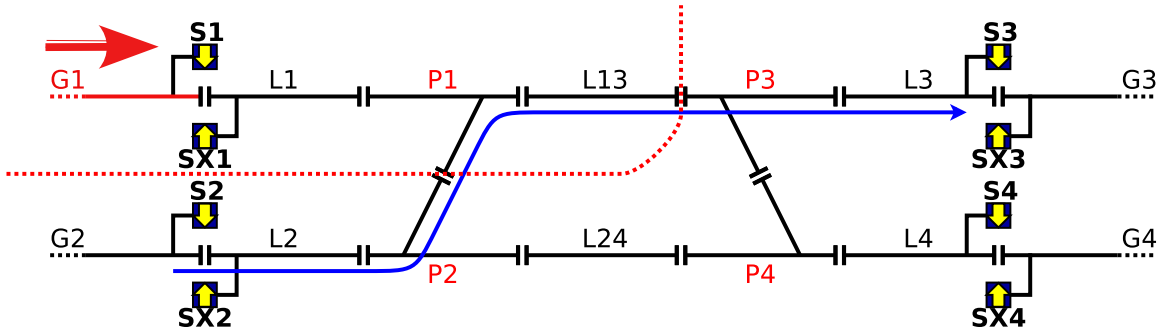


Example: Decomposition Steps for Leval-Binche

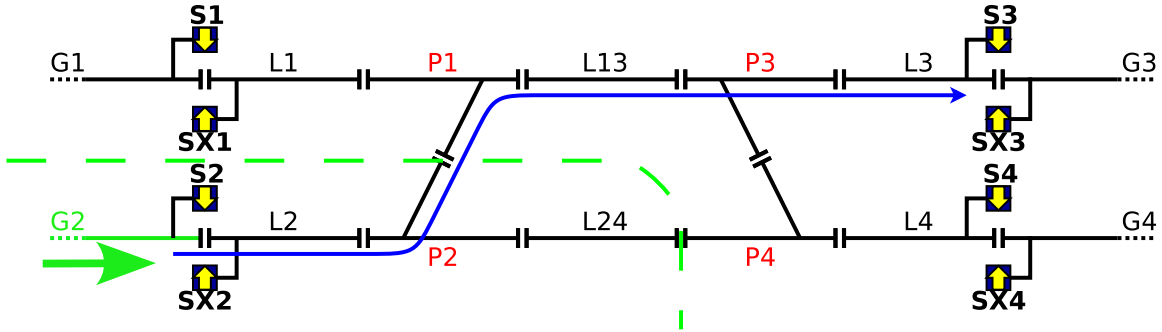


Cuts Found when Searching from Different Borders

From G1: invalid cut

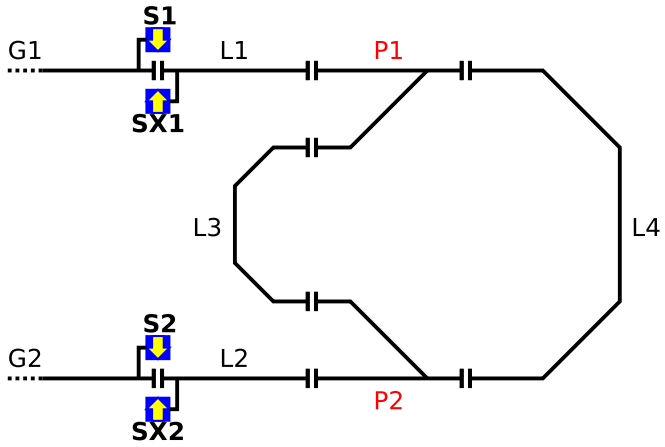


From G2: valid cut



Decomposability

A network containing a loop:



- This network can't be decomposed into elementary networks.

Conjecture:

- For any network N: if N is *loop-free* and no flank protection is adopted, decomposition results in elementary networks $N_1, \ldots, N_n$
  $\rightarrow$ no model checking is needed.

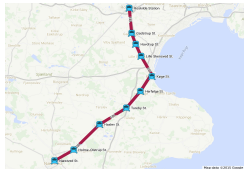
Specification & Implementation of a Decomposer Tool

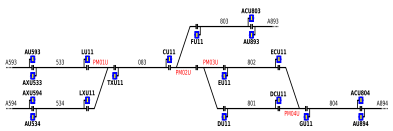
- Formal RSL specification of a decomposer algorithm.
- C++ implementation.

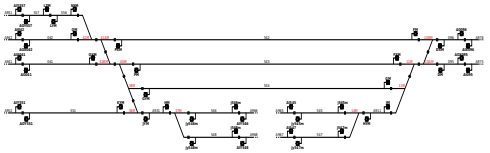
```
decompose : NetworkLayout × SecId × SecId-set →
        NetworkLayout-set × NetworkLayout-set
decompose(N, b, bs) ≡
  let cutset = find_cuts(N,b) in
    if cutset = { {} } -- means N is elementary
      then ({N}, {})
    elsif cutset = {} -- means N is unbreakable from b
      then -- try to search from another border
        if bs = {} then ({}, {N}) -- no other borders to search from
        else let b2 • b2 ∈ bs in
               decompose(N, b2, bs \ {b2}) -- search from border b2
             end
        end
    elsif card cutset = 1 -- found one non-empty cluster cut
      then
        let
          ccut • ccut ∈ cutset, -- ccut is the found cluster set
          (N1, N2) = divide(N, ccut) -- divide N into two nets N1 and N2
                                     -- ccut is defined s.t. N1 becomes elementary
          added_borders = ..., -- set of borders added to N2 during division
          b2 • b2 ∈ added_borders,
          (e_ns, u_ns) =
            decompose(N2, b2, (bs ∩ borders(N2)) ∪ added_borders\{b2})
        in
          ({N1} ∪ e_ns, u_ns)
        end
    else -- found two cluster cuts having one single cut each
      -- see text for explanation of what is then done
      ...
    end
  end
```

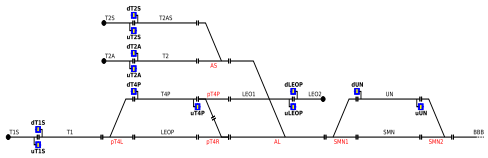
```
find_cuts : NetworkLayout × SecId → ClusterCut-set
find_cuts(N, b) ≡
  let
    dir = find_direction_towards_neighbor_of_border(b, N),
    l = next_from_linear(b, dir, N) -- l is the neighbor of b
  in
    find_cuts_from_linear(l, dir, N)
  end,

find_cuts_from_linear : SecId × Direction × NetworkLayout → ClusterCut-set
find_cuts_from_linear(l, dir, N) ≡
  let next = next_from_linear(l, dir, N) in -- next section to visit
    if is_linear(next, N)
    then
      if is_border(next, N) then {{}} -- case 2(a)
      elsif has_signal(l, dir, N) ∨ has_signal(next, opposite_direction(dir), N)
        then { { (l , next) } } -- case 2(b)
      else -- no signals between l and next
        find_cuts_from_linear(next, dir, N) -- continue search from next
      end
    else -- is_point(next, N), i.e. case 2(c)
      -- further search depends on from which side the point is met:
      case get_pointend_entry_given_neighbor(next, l, N) of
        STEM → find_cuts_from_stem(next, dir, N),
        PLUS → find_cuts_from_branch(next, dir, PLUS, N),
        MINUS → find_cuts_from_branch(next, dir, MINUS, N)
      end
    end -- if
  end -- let
```

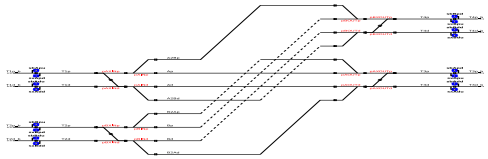
Experimental Results: applying *RR-T model checker* and *RSL & C++ decomposers*











	Number of Elements				Execution Time in seconds			Number of Sub-Nets
Network	Linears	Points	Signals	Routes	RR-T	RSL	C++	Elementary
EDL line (DK)	111	39	126	179	*22863*	*219*	*1,5*	68
Leval-Binche (B)	11	4	18	18	*91*	*7*	*0,5*	4
Piéton (B)	26	12	42	48	*49813*	*9*	*0,5*	13
Tramway Line (I)	22	12	20	62	*43184*	*8*	*0,5*	12
Flying Junction	24	16	16	40	*62172*	*9*	*0,7*	16

Conclusions

Contributions

- a sound and complete method for *compositional verification*
- a *library* of verified, elementary networks
- formally specified and implemented a tool for *automated decomposition*:
  - decomposes a network N into sub-networks $N_1, \ldots, N_n$
  - **conjecture:** if N is *loop-free* & no flank protection, $N_1, \ldots, N_n$ are *elementary* $\rightarrow$ no model checking is needed
  - experiments: execution time of decomposer is a very small fraction of that for model checking

Future work

- prove correctness of the $decompose$ algorithm
- prove the conjecture
- extend decomposer to also handle flank protection
- study the applicability of our approach to other verification frameworks and tools