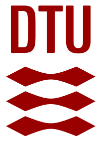
Downloaded from orbit.dtu.dk on: Oct 21, 2021



DTU Library

# Synthesis of railway-signaling plans using reachability games

**Kasting, Patrick Frederik Soelmark; Hansen, Michael Reichhardt; Vester, Steen**

*Published in:*
28th Symposium on the Implementation and Application of Functional Programming Languages,

*Link to article, DOI:*
10.1145/3064899.3064908

*Publication date:*
2016

*Document Version*
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

*Citation (APA):*
Kasting, P. F. S., Hansen, M. R., & Vester, S. (2016). Synthesis of railway-signaling plans using reachability games. In *28th Symposium on the Implementation and Application of Functional Programming Languages,* (Vol. Part F127410). Association for Computing Machinery. Acm Int. Conf. Proc. Ser https://doi.org/10.1145/3064899.3064908

**General rights**
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

# Synthesis of Railway-Signaling Plans using Reachability Games


Patrick Kasting
DTU Compute
Technical University of Denmark
DK-2800, Lyngby, Denmark
s124313@student.dtu.dk

Michael R. Hansen
DTU Compute
Technical University of Denmark
DK-2800, Lyngby, Denmark
mire@dtu.dk

Steen Vester*
Flow Robotics
DK-2300, Copenhagen, Denmark
sv@flow-robotics.com


## ABSTRACT


In this work, we show the feasibility of using functional programming (more specifically F#) in connection with game-based methods for synthesis of correct-by-construction controllers (also called *signaling plans*) for railway networks. This is a massively resource-demanding application.

A model for railway networks comprising trains, signals, linear sections, and points is established together with a domain-specific language capturing the important concepts in the model. A translation from railway network models to two-player reachability games is provided. In these games, the existential player (the control system) controls signals and points and the universal player (the antagonistic environment) controls movement of trains. A winning strategy for the existential player provides a signaling plan that will safely guide trains through the network.

The concepts from the railway network model and the two-player reachability game are captured, in a natural manner, by type declarations in F#. Furthermore, the F# translation functions are formulated in a manner that is close to the mathematical formulations. This increases confidence in the correctness of the implementation and it decreases the development time. Imperative features of F# proved useful in two places: Hash tables and arrays were used in the representations of the railway network model and the reachability game. This allowed for more compact representations and a more efficient game solver (providing the winning strategy).

Experiments show that we are able to synthesize signaling plans for real railway networks of substantial size.


## CCS CONCEPTS


• **Theory of computation** → **Algorithmic game theory**; • **Software and its engineering** → **Functional languages**; Domain-specific languages;


*Part of this work was done while Steen Vester was at Technical University of Denmark.


Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.


*IFL 2016, Leuven, Belgium*


© 2016 ACM. 978-1-4503-4767-9/16/08...$15.00


DOI: http://dx.doi.org/10.1145/3064899.3064908


## KEYWORDS


Domain-specific language, functional programming, railway networks, reachability games, signaling plans, synthesis

**ACM Reference format:**
Patrick Kasting, Michael R. Hansen, and Steen Vester. 2016. Synthesis of Railway-Signaling Plans using Reachability Games. In *Proceedings of IFL 2016, Leuven, Belgium, August 31-September 02, 2016,* 13 pages.
DOI: http://dx.doi.org/10.1145/3064899.3064908


## 1 INTRODUCTION

Railway network systems are often mentioned as examples of safety-critical systems, for obvious reasons, and formal methods have a significant role in achieving confidence in the correctness of such systems. Predominant approaches to achieving this are based on verification techniques, in particular model checking, but these techniques are geared towards finding bugs in designs – they are of limited use in the design process. We investigate the feasibility of using game-based methods for *synthesis* of correct-by-construction controllers [14, 21, 22].

The challenge of verification and synthesis techniques for railway-network problems is state-space explosion: The number of positions of trains in a network grows exponentially with the number of trains, the number of combinations of signals in a network grows exponentially with the number of signals, and so on. The goal of this work, which is based on [17], is to show that

- game-based techniques can be used to synthesize signaling plans for real railway networks of a substantial size (comparable to the size of the problems that can be handled by verification techniques), and that
- functional programming provides an efficient development and implementation platform for a tool synthesizing signaling plans from railway network models.

### 1.1 The Problem

A train's movement in a railway network must be guided by points and signals, such that it reaches its destination without risk of derailment or collision. The points must direct the trains along their respective routes and the signals must halt certain trains in order to give way to others.

A simple example is found in Figure 1. It contains four *sections* *s10*, *s11*, *s12* and *s20*, where *s10*, *s11* and *s20* are linear sections and *s11* is a *point*. The point can switch

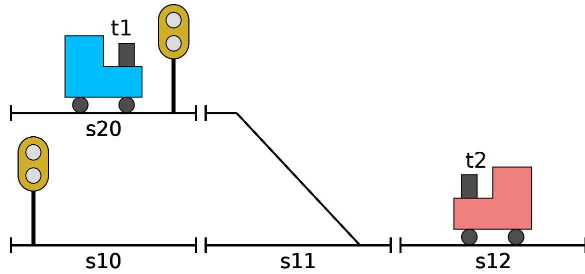



**Figure 1: A railway network (inspired by [24]) containing four sections, two signals, two trains, and the point *s11*.**

between connecting the sections *s20* and *s12* or the sections *s10* and *s12*. Sections allow traffic in both directions.

The train *t1* occupies section *s20* and the train *t2* occupies the section *s12*. If the signal along section *s20* is closed, it prevents *t1* from entering *s11*. The train *t2* is free to enter *s11* and continue to *s10* or *s20* depending on the state of *s11*.

Based on the current positions of the trains, a control system must select the right configuration of points and signals. For this, the system needs a mapping from train positions to configurations. Such a mapping is also called a *signaling plan* and it can be considered an algorithm for controlling the traffic in a network. A signaling plan is *correct*, if it ensures that the trains reach their destinations without crashing. We consider the following problem:

*Synthesis of correct signaling plans for railway networks*

## 1.2 Game-Based Synthesis

The synthesis problem is solved using game-based methods: In a reachability game, two players: Adam (the universal player) and Eve (the existential player), take turns moving a token along the edges of a graph. Eve wins the game, if the token reaches a designated set of vertices. Otherwise, Adam wins. We model a railway network as such a game.

The states of the network act as the vertices of the game and we make changes to the current state by moving the game token. Eve takes the role of the control system and is in charge of updating points and signals. Adam moves the trains, obeying the signals. Eve wins if the trains reach their destinations without crashing and Adam wins if he can move a train away from its route or crash it. If Eve can force Adam to lose, even if Adam acts as a perfect adversary, then we can get the trains correctly through the network. A correct signaling plan is obtained from Eve's winning strategy.

Previously, game-based techniques have been used for synthesis of control algorithms for *concrete examples* of railway-networks (e.g. [9]). However, to the best of the authors' knowledge, at the time of writing, there have been no game-based attempts at solving the synthesis problem for *arbitrary* railway networks. For other examples of game-based synthesis, see [3, 4, 18].

## 1.3 The functional approach

A functional programming language with a strong type system, in this case F$^{\#}$ [15, 23], was used as implementation platform. The concepts from a mathematical model of railway networks [17] are captured in a natural manner by type declarations and so are the concepts from reachability games. The F$^{\#}$ translation functions from network models to games are formulated in a manner that is close to the mathematical formulation. This increases confidence in the correctness of the implementation and it decreases the development time. Imperative features of F$^{\#}$ proved useful in two places: Hash tables and arrays were used in the representations of the railway network model and the reachability game. This allowed for more compact representations and a more efficient game solver (providing the winning strategy).

The most interesting experiment conducted so far is that a correct signaling plan for (approximately one fourth of) Florence Station with four trains is generated in 8.1 seconds. The considered part of Florence Station has 69 linear sections, 23 points, and 46 signals. This is a size approaching the limit of what state-of-the-art verification approaches, e.g. [24], can handle.

## 1.4 Background

In Europe, the predominant standard for railway signaling is the *European Rail Traffic Management System* (ERTMS), see for example [24]. ERTMS comprises a standard ETCS for in-cab signaling and automatic train protection systems [10]. ETCS has three application levels ranging from Level 1 to Level 3, where Level 3 is the most autonomous. In 2009, the Danish parliament decided to replace the entire railway-signaling system with an implementation of ERMTS with ETCS Level 2 [1].

In ETCS, signaling is controlled according to *interlocking tables*. An interlocking table contains one entry for each *elementary route* in the network. An elementary route starts at a signal, ends at another signal, and there is no signal between the two. *Compound routes* are formed by concatenating elementary routes. Signaling works according to the principle that a route is locked exclusively for one train at a time [24].

This static generation of routes has the advantage that a train entering a route can drive to the end of that route without stopping, thereby saving time and energy – frequent stop-and-go is energy consuming. However, the statically generated routes are inflexible and of no use if, for example, a malfunctioning train blocks a track and alternative routes are needed.

Correct signalling plans provide a controlling mechanism that can be used at run-time: In any traffic situation the controller makes a safe decision that brings the trains closer to their destinations. But signaling plans do not guarantee "good" routes: In the current form, there is no attempt at avoiding stop-and-go situations.

*PGSolver.* The PGSolver framework [12] provides an efficient platform for solving *parity games*, that is, a kind of

two-player games of perfect information and infinite duration. PGSolver is implemented in the functional programming language OCaml and our solver for reachability games is implemented in F$^{\#}$, which is closely related to OCaml. With respect to choices of data structures, our implementation is inspired by PGSolver.

## 2 GAMES FOR RAILWAY NETWORKS

In a *turn-based two-player game*, or from now on a *game*, the players Eve and Adam move a shared token along the edges of a shared *game graph*. The vertices of the game graph are partitioned into $V_E$ and $V_A$. Eve chooses the next move from vertices in $V_E$ and Adam chooses the next move from vertices in $V_A$.

Given a *winning condition* and an initial vertex, we want to know whether Eve can ensure a win, no matter how Adam plays. If this is the case, we are also interested in a *winning strategy*, which specifies a winning move from each vertex in $V_E$. We now provide formal definitions of the above concepts. For a more thorough introduction, see e.g. [2, 19].

A *game graph* $G$ is a tuple $(V_E, V_A, E, c)$, where $E \subseteq (V_E \cup V_A) \times (V_E \cup V_A)$ is the set of edges and $c : (V_E \cup V_A) \to C$ is a function, which colors every vertex with a color from the set $C$. We require that $V_E \cap V_A = \emptyset$ and we let $V = V_E \cup V_A$ denote the set of all vertices. Also, for every $v \in V$, we define $vE = \{v' \mid (v, v') \in E\}$, which is the set of successors to $v$. We assume that $vE \neq \emptyset$ for all $v \in V$.

A *play* in $G$ is an infinite sequence $\alpha = v_0 v_1 v_2 \ldots$ of vertices such that $(v_i, v_{i+1}) \in E$ for all $i \in \mathbb{N}$. Let $c(\alpha)$ denote the corresponding sequence $c(v_0)c(v_1)c(v_2)\ldots$ of colors.

A *game* is a pair $(G, \mathit{Win})$, where $G$ is a game graph and $\mathit{Win} \subseteq C^\omega$ is a *winning condition*. Eve wins a play $\alpha$, if $c(\alpha) \in \mathit{Win}$. Otherwise, Adam wins.

A *strategy* for Eve is a function $\sigma : V^* V_E \to V$ such that $\sigma(xv) = v'$ implies $(v, v') \in E$. A play $v_0 v_1 v_2 \ldots$ is *played according to* the strategy $\sigma$, if

$$\forall i \in \mathbb{N} : v_i \in V_E \Rightarrow \sigma(v_0 v_1 \ldots v_i) = v_{i+1}.$$

Given an initial vertex $v_0$ and a strategy $\sigma$ for Eve, let $\mathit{Out}(\sigma, v_0)$ denote all plays from $v_0$ played according to $\sigma$. We say that $\sigma$ is a *winning strategy* from $v_0$, if $c(\alpha) \in \mathit{Win}$ for all $\alpha \in \mathit{Out}(\sigma, v_0)$.

### 2.1 Reachability games

Suppose a designated target color $c' \in C$ is given. A game with a winning condition of the form

$$\{c_0 c_1 \ldots \mid \exists i \in \mathbb{N} : c_i = c'\}$$

is called a *reachability game* [2]. In such a game, Eve wins a play, if the token eventually reaches a vertex colored $c'$. Vertices having color $c'$ are called *goal* vertices. For this type of games, we have [2]:

Theorem 2.1. *Let $\mathcal{G} = ((V_E, V_A, E, c), \mathit{Win})$ be a reachability game. Then, there exists a partition $\{\mathit{Win}_E, \mathit{Win}_A\}$ of $V$, such that Eve has a winning strategy from any initial vertex $v \in \mathit{Win}_E$ and Adam has a winning strategy from any initial vertex $v \in \mathit{Win}_A$.*

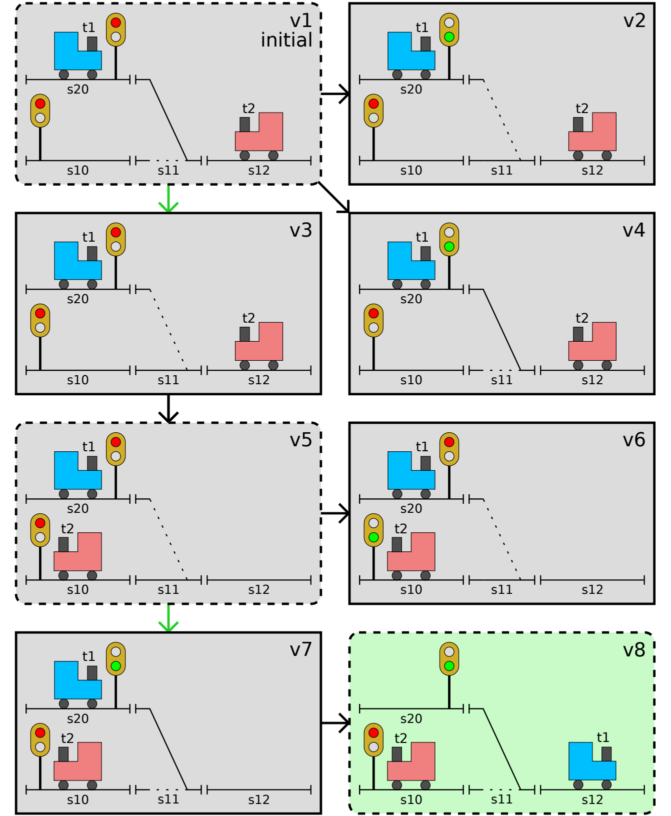



**Figure 2: A subset of the game graph induced by the railway network shown in Figure 1. The green edges (*v1*,*v3*) and (*v5*,*v7*) indicate Eve's winning moves.**

In essence, this theorem states that if both players are perfect, then the winner is determined before the first move. We use the term *winning areas* to describe $\mathit{Win}_E$ and $\mathit{Win}_A$.

In addition, it can be shown that reachability games are *memoryless determined* [2]. This means that in order to ensure a win, the winning player only needs to consider the current vertex when deciding on his next move.

### 2.2 A Game for a Railway Network

A subset of the game graph for the railway network in Figure 1 is shown in Figure 2. Recall that Eve changes the configuration of signals and points and Adam moves the trains. That is, Eve controls the vertices with dashed edges and rounded corners and Adam controls the vertices with solid edges and sharp corners. In order to cause progression, we force Adam to move a train, if these are not all blocked by closed signals. The turn alternates between the two players.

The token starts at *v1*, where the trains are at their initial positions and the points and signals are in default states. The vertex *v8* is a goal vertex, as we assume that *t1* and *t2* have sections *s12* and *s10* as respective destinations.

In *v1*, Eve has the turn and she must move the token to one of the vertices *v2*, *v3*, *v4*, and some others not shown in Figure 2. From *v2*, Adam can derail *t1* by moving the train onto the point *s11*, which connects *s10* and *s12* and not *s20* and *s12*. From *v4*, Adam can collide the trains, because *s11* connects *s20* and *s12*. Clearly, Eve wants to avoid these two vertices.

In *v3*, *t1* is blocked by a closed signal, so Eve does not risk derailment. The point *s11* connects *s10* and *s12*, so she does not risk collision either. In addition, from *v3*, Adam is forced to move *t2* to *s10*, which is the destination of *t2*. Hence, Eve moves from *v1* to *v3* and Adam moves from *v3* to *v5*.

In *v5*, Eve has the turn and she can choose between *v6*, *v7*, and other vertices not shown in Figure 2. From *v6*, Adam can derail *t2*, because the signal along *s10* is open. Therefore, Eve moves to *v7*.

In *v7*, the point *s11* connects *s20* and *s12* and the train *t1* is allowed to move, as the signal along *s20* is open. Because Adam is forced to move a train if he can, he must move *t1* to *s12* by moving the game token to *v8*. This vertex is a goal vertex, so we terminate the game with Eve as the winner. Furthermore, Adam never had a chance of winning, so Eve has a winning strategy by making the moves just described. Notice that these moves also describe a correct signaling plan for the network.

## 2.3 Solving Reachability Games

The proof of Theorem 2.1 presented in [2] is interesting, because it reveals a way to compute an optimal strategy for Eve. The corresponding algorithm is shown in Algorithm 1. This algorithm computes Eve's winning area based on a breadth-first search: Starting from the goal vertices, it iteratively expands the winning area by exploring the reverse game graph. When the algorithm discovers an Eve vertex, it is included in the winning area, as Eve can move to the current winning area in one move. When the algorithm discovers an Adam vertex where every successor has been explored, it is also included in the winning area, as Adam has no choice but to move to the current winning area. In order to achieve linear time complexity, we keep track of the number of unexplored successors to each vertex.

The optimal strategy for Eve is to move to a vertex explored earlier than the current one whenever possible. This strategy is winning from every vertex in her winning area.

We use the types shown in Figure 3 to model game graphs and the concepts from Algorithm 1. With these types, the pseudocode of Algorithm 1 translates straightforwardly into an F$^{\#}$ function `solveReachability` with the following type:

```
GameGraph<'misc,'color> ->
 'color ->
 Winners<'misc,'color> * Strategy<'misc,'color>
```

Here, the second parameter specifies the target color of the game. The function definition is in Appendix A.

We represent a game graph $G = (V_E, V_A, E, c)$ using adjacency lists of the reverse graph and we augment each vertex with its number of successors: An entry in the dictionary for a given vertex $v$ is a pair $(k, [v_1; \ldots; v_n])$, where $k$ is the number of unexplored successors to $v$ in $G$ and where $(v_i, v) \in E$ for $1 \leq i \leq n$. This representation is the basis of an efficient game-solving algorithm that is linear in both the number of vertices and the number edges.

**Algorithm 1** Algorithm for solving a reachability game $\mathcal{G}$ with finite game graph $(V_E, V_A, E, c)$ and target color $c'$. The winning areas of $\mathcal{G}$ can be obtained by scanning the output *win* and an optimal strategy for Eve is given by the output *strategy*. This algorithm is inspired by [2].

```
1:  function SolveReachability((V_E, V_A, E, c), c')
2:      for all v ∈ V_E ∪ V_A do
3:          win[v] ← Adam
4:          P[v] ← ∅
5:          n[v] ← 0
6:          strategy[v] ← ⊥
7:      for all (u, v) ∈ E do
8:          P[v] ← P[v] ∪ {u}
9:          n[u] ← n[u] + 1
10:     Q ← ∅
11:     for all v ∈ V_E ∪ V_A such that c(v) = c' do
12:         win[v] ← Eve
13:         Enqueue(Q, v)
14:     while Q ≠ ∅ do
15:         v ← Dequeue(Q)
16:         for all p ∈ P[v] do
17:             n[p] ← n[p] − 1
18:             if win[p] = Adam ∧ (p ∈ V_E ∨ n[p] = 0) then
19:                 win[p] ← Eve
20:                 strategy[p] ← v
21:                 Enqueue(Q, p)
22:     return (win, strategy)
```

```
type Player = Adam | Eve
type Vertex<'misc, 'color>
    when 'misc:equality and 'color:equality =
    'misc * 'color * Player
type GameGraph<'misc, 'color>
    when 'misc:equality and 'color:equality =
    Dictionary<Vertex<'misc, 'color>,
               int * List<Vertex<'misc, 'color>>>
type Winners<'misc, 'color>
    when 'misc:equality and 'color:equality =
    Dictionary<Vertex<'misc, 'color>, Player>
type Strategy<'misc, 'color>
    when 'misc:equality and 'color:equality =
    seq<Vertex<'misc, 'color> * Vertex<'misc, 'color>>
```

**Figure 3: Representations of game graphs, winning areas, and strategies. The `int` component of `GameGraph` is used to store the number of successors to a vertex. The types `List` and `Dictionary` are from the standard library of the Common Language Infrastructure [6, 8] and they implement dynamic arrays and hash tables.**

```
type Label = string
type Linear = Linear of Label
type LinearExtended = RealLinear of Linear | Derail
type Point = Point of Label
type LinearPort = Down of Linear | Up of Linear
type PointPort = | Stem of Point
                 | Plus of Point
                 | Minus of Point
type PointPortExtended = | RealPointPort of PointPort
                         | PointPortNull
type Port = | LinearPort of LinearPort
            | PointPort of PointPort
type PortExtended = RealPort of Port | Null
type Signal = LinearPort
```

**Figure 4: The types representing basic railway network concepts.**

# 3 RAILWAY NETWORK MODEL

The program design presented below is based on the mathematical model of railway networks developed in [17].

## 3.1 Basic Concepts

The basic concepts from the mathematical model are expressed by F# types in Figure 4. Central concepts are *linear section* (type `Linear`) and *point section* (type `Point`). Each section has a unique label.

A *port* (type `Port`) is the end of a section: A linear section has two ports called *up* and *down* and a point section has three ports called *stem*, *plus*, and *minus*. The track between *stem* and *plus* forms the main path through the point. These concepts originate from the Danish railway terminology, where *up* (*down*) denotes the direction, in which the distance from some reference location is increasing (decreasing) [24]. A *signal* (type `Signal`) is placed at a port of a linear section.

Some types have been extended in order to cover special cases: For example, `LinearExtended` contains the artificial section `Derail`, which indicates a derailment. Likewise, `PortExtended` contains the artificial port `Null`, which marks the end of a track.

## 3.2 Network Specifications and States

In Figure 5, we present the types representing railway networks and their states. A *network* (type `Network`) is a tuple $(L, P, con, Signals, \underline{start}, \underline{dest})$, where

- $L$ is the set of linear sections,
- $P$ is the set of points,
- *con* is a function describing the connections of the ports,
- *Signals* is the set containing the linear ports, at which there are signals, and
- the vectors $\underline{start}$ and $\underline{dest}$ describe the initial positions and the destinations of the trains.

A *state* (type `State`) represents an instantaneous description of the network. A state is a tuple $(\underline{pos}, \Pi, \Sigma)$, where

- $\underline{pos}$ is a vector describing the train positions,

```
type PositionVector = Linear[]
type PositionVectorExtended = LinearExtended[]

type Network = (HashSet<Linear> * HashSet<Point>
                * (Port -> PortExtended)
                * HashSet<Signal>
                * PositionVector * PositionVector)

type Direction = DirectionDown | DirectionUp
type PointState = PointStatePlus | PointStateMinus
[<CustomEquality; NoComparison>]
type State = State of PositionVectorExtended
                   * HashSet<Point>
                   * HashSet<Signal>
```

**Figure 5: The types representing railway networks and their states. The type `HashSet` is from the standard library of the Common Language Infrastructure [7] and it implements mathematical sets.**

- $\Pi$ is a set containing every point in the *plus-state* connecting the stem-port and the plus-port, and
- $\Sigma$ is a set containing the open signals.

We have implemented custom definitions for equality and comparisons of `State`. This is because the standard library uses pointer equality to determine, whether two `HashSet`s are equal. Hence, without custom definitions, we may distinguish states, which are equal in a mathematical sense.

Note that it is possible to optimize the types of Figure 4 and Figure 5 for greater performance: By incorporating `Signal` into `Linear` and `PointState` into `Point`, we can determine the states of signals and points without lookups in $\Pi$ and $\Sigma$. In fact, $\Pi$ and $\Sigma$ would be unnecessary. However, in order to increase development speed and minimize the risk of bugs, we have kept the type definitions simple and close to the mathematical model.

## 3.3 State Transitions

The railway model from [17] distinguishes between two types of *state transitions*: Movements of trains and updates to points and signals. The railway control system has no direct control of the former, whereas it has full control of the latter (leading to the roles of Adam and Eve).

At most one train moves during a single state transition. It moves directly to the next linear section, possibly traversing a point section in the process. It may also derail, if it leaves a section through an unconnected port or if it enters a point which is in the wrong state.

There is no restriction on the number of points and signals that may change state during a single transition.

## 3.4 Translation to a Reachability Game

The states of a railway network act as the vertices of the corresponding game graph and the state transitions act as the edges. Given a railway network, we use a simple search to generate the game graph explicitly: Starting from the vertex representing the initial state, we iteratively generate successors to discovered vertices. This continues until we have explored the entire game graph.

| Function | Type |
|---|---|
| `color` | `Network -> PositionVectorExtended -> Color` |
| `next` | `Network -> State -> Linear -> Direction -> LinearExtended` |
| `succAdam` | `GameVariant -> Network -> Direction[] -> Vertex -> List<Vertex>` |
| `relPointsSigs2` | `Network -> Direction[] -> Linear[] -> (List<LinearPort> * HashSet<Point> * List<Signal>)` |
| `relPointsSigs3` | `Network -> Direction[] -> Linear[] -> (bool * HashSet<Point> * List<Signal * Point option>)` |
| `succEve` | `GameVariant -> Network -> Direction[] -> Vertex -> List<Vertex>` |
| `nextLinears` | `Network -> Direction -> Linear -> Linear list` |
| `directions` | `Network -> Directions[]` |
| `graph` | `GameVariant -> Network -> (GameGraph<State,Color> * Vertex<State,Color>)` |

**Table 1: An overview of the central functions concerning the translation from a network specification to a reachability game.**

We do not generate successors to crash vertices or goal vertices, as the outcome of the game is known, when the game token reaches such a vertex. We use the type `Color` to distinguish crash vertices and goal (success) vertices from the remaining vertices:

```
type Color = Crash | Nothing | Success
```

The type of the generated graph is `Graph<State,Color>`. That is, the type of the vertices is `Vertex<State,Color>`.

The program design facilitating game-graph generation is shown in Table 1. This design follows naturally from the railway model presented in [17] and each function declaration is formulated in a manner that closely resembles the corresponding mathematical definition.

The main function is `graph`: Given a well-formed network, it returns the corresponding game graph and the initial vertex. To be more precise, `graph` generates the game graph of one of four variants $\mathcal{G}^0$, $\mathcal{G}^1$, $\mathcal{G}^2$, and $\mathcal{G}^3$ of the reachability game. We discuss these variants in Section 4. The desired variant is specified by the first parameter, which is of type `GameVariant`:

```
type GameVariant = Zero | One | Two | Three
```

The function `directions` assigns directions to the trains: Each train is assigned the direction, which allows the train to reach its destination. This direction is found using a search on the linear sections. During this search, we use `nextLinears`, which computes the next linear sections in a given direction.

The functions `succAdam` and `succEve` generate successors to a vertex. The function `succAdam` uses `next` to compute the next linear section of a train based on the current state of the network. Since at most one train can move during a single state transition, one successor is generated for each train not blocked by a closed signal.

The function `succEve` generates successors resulting from updating points and signals. Since any number of points and signals can be updated during a single state transition, this function generates a number of successors exponential in the number of points and signals. However, this only applies to the game variants $\mathcal{G}^0$ and $\mathcal{G}^1$. When generating the graphs of $\mathcal{G}^2$ and $\mathcal{G}^3$, `succEve` uses `relPointsSigs2` and `relPointsSigs3` to determine the points and signals, which influence train movements in the current state. By ignoring the remaining points and signals, we drastically reduce the number of successors. We go into further details in Section 4.

Finally, the function `color` colors a vertex based on the positions of the trains in the corresponding state.

### 3.5 From Strategies to Signaling Plans

The function `solveReachability` computes Eve's winning strategy for a given game graph. From this strategy, we must extract a *signaling plan*. A signaling plan contains exactly one entry for each vector of train positions, from which we can get the trains safely to their destinations. Thus, we partition the winning vertices based on their train positions and select one vertex from each partition. For each of these vertices, the strategy describes Eve's winning move. Since an Eve move corresponds to an update of the points and signals, each winning move describes a signaling configuration that is a safe response to the given train positions. Hence, the winning moves from the selected vertices yield the entries in the signaling plan:

```
type SignalingPlan =
    seq<PositionVector
        * (HashSet<Point> * HashSet<Signal>)>
```

## 4 REDUCING GAME SIZE

Let $\mathcal{G}^0$ be the original reachability game, in which every state is a vertex and every state transition is an edge. It follows from the declaration of `State` that a network $(L, P, con, Signals, \underline{start}, \underline{dest})$ can be in $(|L|+1)^n 2^{|P|+|Signals|}$ different states, where $n$ denotes the number of trains. Since both Eve and Adam can have the turn from any of these states, an immediate upper bound on the number of vertices in $\mathcal{G}^0$ is $(|L|+1)^n 2^{1+|P|+|Signals|}$. With some simple combinatorics, it is possible to obtain the following tighter bound [17]:

$$V^0_{\max} = pos_{\max} 2^{1+|P|+|Signals|} \quad (1)$$

Here, $pos_{\max}$ is an upper bound on the number of ways to distribute the trains, keeping in mind that we stop the game after a single derailment or a single collision. It is defined as

$$pos_{\max} = \left(\frac{1}{(|L|-n)!} + \frac{n}{(|L|-(n-1))!} + \frac{(n-1)n}{2(|L|-(n-1))!}\right)|L|!.$$

The first term is the number of distinct position vectors of non-crash states, the second term is the number of distinct

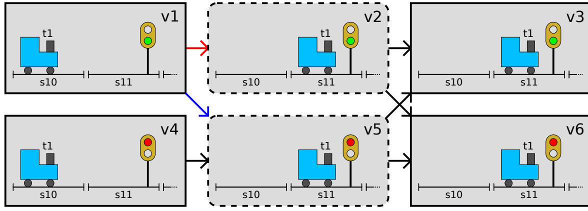



**Figure 6: An abstraction that ignores irrelevant Eve vertices. Adam resets every signal to *closed* before passing the turn to Eve. Hence, the red edge ($v1$,$v2$) is replaced by the blue edge ($v1$,$v5$). This renders *v2* unreachable. Because *v2* and *v5* have the same successors, *v2* is in Eve's winning area if and only if *v5* is in Eve's winning area. Hence, we have not changed the outcome from *v1*, so this abstraction does not change the winner of the game.**

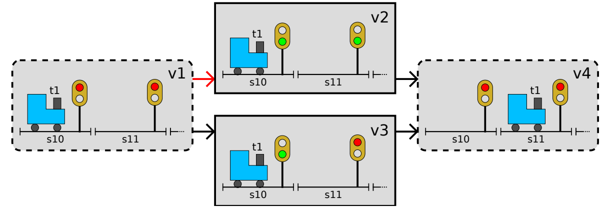



**Figure 7: An abstraction that ignores irrelevant points and signals. We assign the default value of *closed* to every signal not directly in front of a train. Hence, we remove the red edge ($v1$,$v2$). This renders *v2* unreachable. Also, because *v2* and *v3* have the same successor, *v2* is in Eve's winning area if and only if *v3* is in Eve's winning area. Therefore, we have not changed whether *v1* has a successor in the winning area. Thus, we have not changed the outcome from *v1* and consequently, we have not changed the winner of the game.**

position vectors in derail states, and the last term is the number of distinct position vectors in collision states.

The upper bound $V^0_{\max}$ is exponential in the number of points and signals and near-exponential in the number of trains. This renders synthesis intractable for all but the smallest networks. Therefore, we have devised three abstractions, each of which significantly reduces the size of the game graph. When all abstractions are applied, the upper bound on the size of the game graph is exponential only in the number of trains.

These abstractions do not change the winner of the game, so the computed signaling plans remain correct. For each abstraction, we provide an informal argument of this claim. For formalizations and rigorous proofs, we refer to [17].

Starting from the original game $\mathcal{G}^0$, we obtain the game variants $\mathcal{G}^1$, $\mathcal{G}^2$, and $\mathcal{G}^3$ by repeatedly applying an additional abstraction to the previous variant.

## 4.1 Ignoring Irrelevant Eve Vertices

The first abstraction is based on the following observation:

> When Eve receives the turn, the configuration of points and signals is irrelevant, as Eve can change it to whatever she likes before passing the turn to Adam. Hence, without changing the outcome of the game, we can force Adam to reset every point to the *minus-state* and every signal to *closed* before passing the turn to Eve.

This implies that every reachable Eve vertex has every point in the minus-state and every signal closed. The effect of this abstraction is illustrated in Figure 6. The game variant $\mathcal{G}^1$ is the result of applying this abstraction to $\mathcal{G}^0$.

In $\mathcal{G}^1$, we can have at most one reachable Eve vertex per obtainable vector of train positions. The number of Adam vertices stays the same, so an upper bound on the number vertices in $\mathcal{G}^1$ is

$$V^1_{\max} = pos_{\max}(2^{|P|+|Signals|} + 1). \tag{2}$$

## 4.2 Ignoring Irrelevant Points and Signals

The next abstraction is based on the following observation:

> When Adam has the turn, the state of a point or a signal not immediately in front of a train is irrelevant, as Eve can change that point or signal before a train comes close enough to be influenced by it. Also, if we consider a point $s$ and every train about to enter $s$ is blocked by a closed signal, then the state of $s$ is irrelevant.

If we assign the default values of *minus-state* and *closed* to irrelevant points and signals, then the size of the game graph is reduced significantly. The effect of this abstraction is illustrated in Figure 7. The game variant $\mathcal{G}^2$ is the result of applying this abstraction to $\mathcal{G}^1$.

What is the upper bound on the number of vertices in $\mathcal{G}^2$? The number of Eve vertices stays the same but the maximum number of successors to each Eve vertex is $3^n$. Why? In the worst case, each train is in front of a distinct signal $s$ and a distinct point $p$. We claim that there are three relevant configurations of $s$ and $p$: If $s$ is closed, then the state of $p$ does not matter, so we use the default value of *minus-state*. If $s$ is open, then we need to consider both the minus-state and the plus-state.

Each train gives raise to three such configurations and we must generate all combinations. Hence, we have at most $3^n$ successors. In addition, this abstraction does not introduce additional vertices, so $V^1_{\max}$ is still an upper bound. Hence, we obtain the following:

$$V^2_{\max} = \min\{pos_{\max}(3^n + 1), V^1_{\max}\} \tag{3}$$

This upper bound is exponential only in the number of trains. In contrast, the previous bound was exponential in the number of trains, points and signals. Typically, the number of trains is much smaller than the number of points and signals, so this amounts a big difference in the size of the game graph.

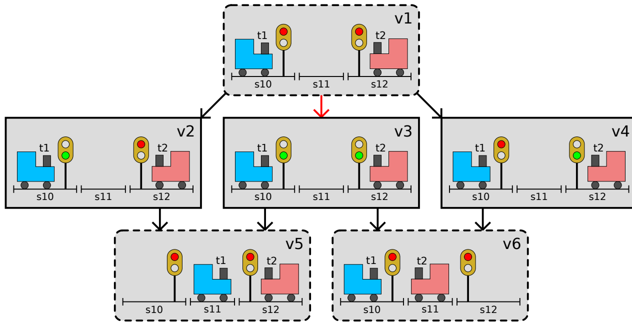



**Figure 8: An abstraction that removes redundant Eve moves. At most one signal can be open at any time. Hence, we remove the red edge (*v1*,*v3*) from the game. This renders *v3* unreachable. Also, because the successors to *v2* and *v4* are subsets of the successors to *v3*, if Eve wins from *v3* she also wins from *v2* and *v4*. Consequently, removing (*v1*,*v3*) does not influence the outcome from *v1*. Conversely, if Eve wins from *v2* or *v4*, her victory does not depend on *v3*. Hence, adding (*v1*,*v3*) does not influence the outcome from *v1* either. We can conclude that this abstraction does not change the winner of the game.**

### 4.3 Removing Redundant Eve Moves

The third abstraction is based on the following observation:

> Assume that every train has a signal in front of it. Eve can close all but a single signal and thereby determine, which train Adam is going to move. If Eve's winning strategy allows more than one signal to be open, then from Eve's point of view, it is irrelevant which of the corresponding trains Adam moves. The strategy leaves the decision up to Adam. However, Eve can make this decision herself. That is, she can select one of the signals allowed to be open and close all other signals.

We can force Eve to make this decision by only generating successor vertices, in which at most one signal is open. This significantly reduces the number of Adam vertices. The effect of this abstraction is illustrated in Figure 8. The game variant $\mathcal{G}^3$ is the result of applying this abstraction to $\mathcal{G}^2$.

As with the other abstractions, this one does not change the outcome of the game. However, in real railway networks where trains can move concurrently, it does indeed make a difference that at most one signal can be open at any time. In situations where it is safe to allow more than one train to move, this abstraction is restrictive and increases the time it takes for all trains to get to their destinations. However, in small networks with large numbers of trains, many signals have to be simultaneously closed in order to avoid collisions. In such cases, this abstraction does not appear too restrictive.

What is the upper bound on the number of vertices in $\mathcal{G}^3$? Again, the number of Eve vertices stays the same but the maximum number of successors to an Eve vertex is now $\lfloor 2^{n-\gamma}(2\gamma+1)\rfloor$, where $\gamma = (2-\ln(2))/\ln(4) \approx 0.94$. Why? First, consider only the trains directly in front of signals. Let $n'$ denote the number of these trains. In the worst case, there is a distinct point behind each signal. In each vertex, at most one signal is open, so the only relevant point is the one behind that signal. Hence, we have $2n'$ relevant configurations in addition to one where every signal is closed.

Now consider the remaining $(n-n')$ trains. In the worst case, they are all in front of points, which gives raise to $2^{n-n'}$ configurations. We must generate all combinations, so we have $2^{n-n'}(2n'+1)$ successors. Within the real numbers, this value has a maximum at $n' = \gamma \approx 0.94$, so we arrive at the following bound on the number of vertices:

$$V^3_{\max} = \min\{pos_{\max}(\lfloor 2^{n-\gamma}(2\gamma+1)\rfloor + 1), V^2_{\max}\} \quad (4)$$

If we compare this to the previous upper bound, we have reduced the base of the exponential function.

## 5 A SYNTHESIS TOOL

The presented algorithms, abstractions, and F# definitions have been incorporated into a synthesis tool. This tool accepts specifications of railway networks written in a domain specific language (DSL). The tool outputs correct signaling plans, whenever such exist.

### 5.1 Domain-Specific Language

The following piece of DSL specifies the network shown in Figure 1:

```
connections =
    s10.up -> s11.plus,
    s20.up -> s11.minus,
    s11.stem -> s12.down
signals = s20.up, s10.down
trains = s20 -> s12, s12 -> s10
```

As shown, a network is specified by a list of port connections, a list of signals, and a list of the initial positions and destinations of the trains. This immediately yields the last four components of a network specification ($L$, $P$, *con*, *Signals*, $\underline{start}$, $\underline{dest}$). The sets $L$ and $P$ are extracted from the list of connections. A formal grammar of this DSL is in [17].

The parser for this DSL was constructed using the parser-combinator library FParsec [11]. This library is an adaptation of the Haskell-library Parsec [11]. Compared to graph generation and game solving, parsing is blazingly fast (see Table 3).

### 5.2 Well-Formed Networks

The syntax of the DSL does not ensure that the network specifications are well-formed. For example, a train may have a destination, which is not a linear section. Therefore, after parsing the network description into abstract syntax, we check for ill-formed networks [17].

### 5.3 Example Output

If the given network is well-formed, the tool constructs the game graph and solves the game. Afterwards, it outputs a few performance statistics and a correct signaling plan, if one exists.

```
Railway network loaded!
Generating game graph...
Game graph generated!
Number of vertices: 37
Number of edges: 60
Computing signaling plan...
signalingplan([s20, s10]) =
- plus:
- minus: s11
- open: s20.up
- closed: s10.down
signalingplan([s20, s12]) =
- plus: s11
- minus:
- open:
- closed: s20.up, s10.down
Parsing and well-formedness: 0.0959 seconds
Game-graph generation: 0.0389 seconds
Signaling-plan computation: 0.0547 seconds
Total execution time: 0.190 seconds
```

**Figure 9: Output of the synthesis tool after solving the network from Figure 1 with $\mathcal{G}^0$.**

The output after solving the network from Figure 1 is shown in Figure 9. From this output, it is evident that the tool has computed a signaling plan: If the trains *t1* and *t2* have the respective positions *s20* and *s10*, then a safe configuration is to put *s11* in the minus-state and open only the signal at the up-port of *s20*. If *t1* and *t2* have the positions *s20* and *s12*, then a safe configuration is to put *s11* in plus-state and close all signals. This exactly corresponds to Eve's winning moves in the game from Figure 2.

## 6 EXPERIMENTS

In order to determine its performance, we conducted several experiments with the synthesis tool [17]. We used, among others, the networks Toy, Fork, Lyngby Station, and Florence Station depicted in Figure 1, Figure 10, Figure 11, and Figure 12, respectively. The complexities of these networks are summarized in Table 2.

| | Linears | Points | Signals | Trains |
|---|---|---|---|---|
| Toy | 3 | 1 | 2 | 2 |
| Fork | 7 | 2 | 6 | 3 |
| Lyngby | 11 | 6 | 14 | 5 |
| Florence | 69 | 23 | 46 | 4 |

**Table 2: Overview of the sizes of the networks used for experiments.**

During the experiments, we measured the sizes of the generated game graphs and the running times of several phases of the computations. Each of the game variants $\mathcal{G}^0$, $\mathcal{G}^1$, $\mathcal{G}^2$, and $\mathcal{G}^3$ was used to solve each of the networks. The experiments were conducted on a MacBook Pro[1] (Retina 13", primo 2015). The results are shown in Table 3.

[1]This machine has a 2.7-GHz Intel-Core-i5 processor and 8 GB of 1867-MHz DDR3 memory. The source code was compiled using The Open Edition of the F# Compiler [20] and executed using the Mono JIT compiler (version 4.2.3) under OS X El Capitan (version 10.11.5).

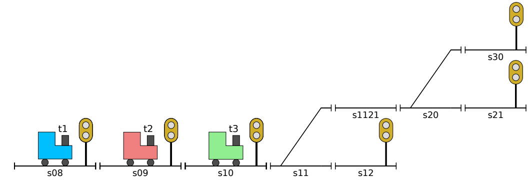



**Figure 10: The network Fork inspired by [24].**

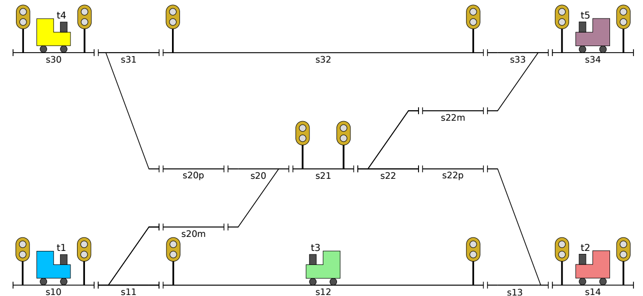



**Figure 11: A railway network resembling Lyngby Station located north of Copenhagen, Denmark. Compared to the layout of Lyngby Station shown in [24], we have added four linear sections, as our model does not allow two points directly connected to each other.**

The most interesting result is that we are able to generate correct signaling plans for Lyngby Station with five trains in 0.67 seconds and for Florence Station with four trains in 8.1 seconds. That is, we are able to generate signaling plans for real railway networks in less than 10 seconds.

These running times are achieved using $\mathcal{G}^3$. With $\mathcal{G}^2$, the running time for Lyngby Station approximately quadruples to 2.6 seconds and for Florence Station it increases by roughly 50 percent to 12 seconds. When using $\mathcal{G}^0$ and $\mathcal{G}^1$, the tool runs out of memory before the signaling plans are generated. This is a testimony to the usefulness of the abstractions presented in Section 4.

The abstraction of ignoring irrelevant points and signals seems especially powerful: Lyngby and Florence Stations are unsolvable with $\mathcal{G}^1$ but are solvable with $\mathcal{G}^2$. This is no real surprise, as $V^1_{\max}$ (see equation (2)) is exponential in the number of trains, points, and signals, whereas $V^2_{\max}$ (see equation (3)) is exponential only in the number of trains.

The difference between $\mathcal{G}^2$ and $\mathcal{G}^3$ is much smaller: The running times when using $\mathcal{G}^2$ are within a factor four of the running times obtained by $\mathcal{G}^3$. Hence, if we find the at-most-one-open-signal aspect of $\mathcal{G}^3$ too restricting, then we can use $\mathcal{G}^2$ and expect no more than a quadruplication of the running time.

It is also worth noting that the sizes of the generated game graphs are significantly lower than the upper bounds presented in Section 4. In these upper bounds, we include every possible way of distributing the trains in a network. However, many of these distributions are not obtainable, as two trains cannot pass each other on a single track. In

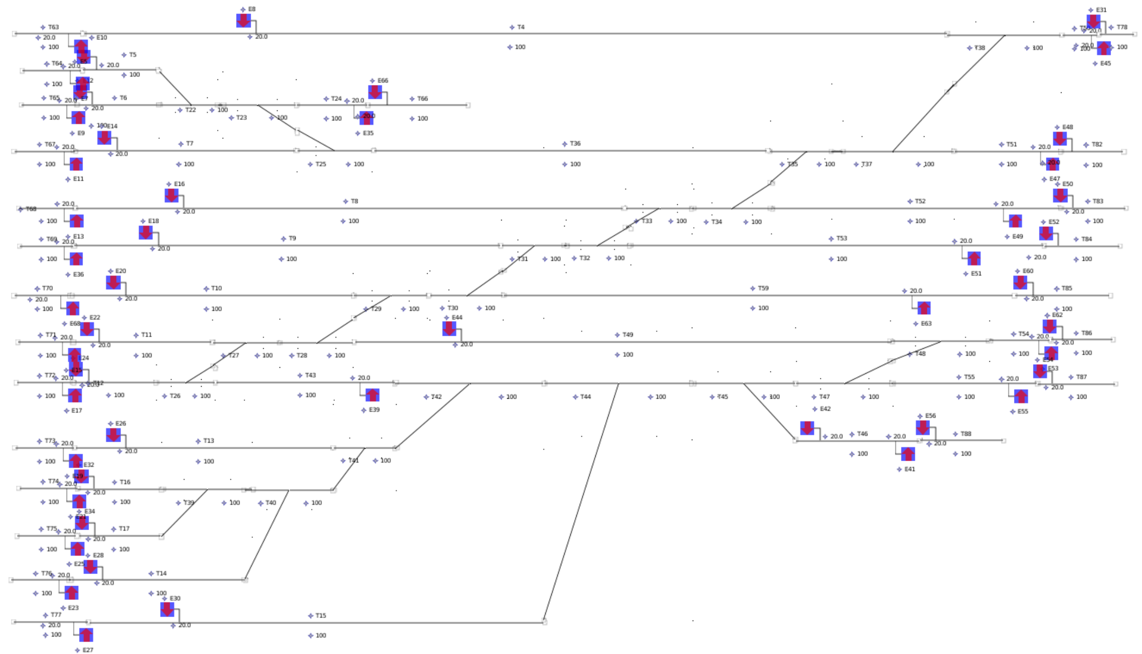


**Figure 12: Layout of roughly a quarter of Florence Station located in Florence, Italy. Each blue square represents a signal. This drawing was kindly provided by Alessandro Fantechi.**

addition, it is often the case a train cannot reach a large part of the network, as it is not allowed to change direction. Hence, in large networks with many trains, these bounds are not very tight. However, if we keep this in mind, they can still serve as indications of the complexity of a given railway network.

We now turn to the running times of the individual computation phases: The parsing and the checks for well-formedness always take around 0.1 seconds and the majority of this is probably spent on disk access. These steps are fast, because they only require an amount of work linear in the number of network entities. For example, parsing a port connection requires processing a constant amount of characters[2].

For the networks Lyngby and Florence Stations, the generation of the game graph takes significantly longer than solving the resulting game. This is expected, as Algorithm 1 spends very little time at each vertex: It makes a few reads and writes in hash tables and adds new nodes to a queue. All of these operations are fast. During graph generation, we have to maintain the hash table that represents the game graph. In addition, we need to find the next linear sections of trains, find the relevant points and signals, and compute power sets of points and signals. Therefore, game-graph generation is the performance bottleneck.

Finally, the tool can use a lot of memory. This is indicated by the fact that $\mathcal{G}$ and $\mathcal{G}^1$ cannot solve Lyngby and Florence Stations without running out of memory. Also, adding a fifth train to Florence Station causes the tool to run out of memory when we use $\mathcal{G}^2$ and $\mathcal{G}^3$.

The maximum amount of memory that the tool can allocate seems to be somewhere between 2.85 GB and 3.37 GB. This is a limit imposed by the Common Language Runtime or the operating system, as the machine used for experiments has 8 GB of memory and plenty of disk space for virtual memory.

We did not have the time to figure out how to remove this artificial limit. Hence, we cannot determine the precise amount of memory needed to solve Florence Station with five trains. However, when we add the fifth train, the upper bound on the number of vertices in $\mathcal{G}^3$ increases from $6.0 \times 10^8$ to $8.1 \times 10^{10}$, which is a factor 135 increase. When solving Florence Station with four trains, the tool uses approximately 0.14 GB of memory, so it is reasonable to assess that the tool needs roughly 19 GB of memory to solve Florence Station with five trains. By the same logic, it would take roughly 18 minutes to compute a signaling plan.

## 6.1 Purely Functional Game-Graph Generation

In order to determine the performance of purely functional game-graph generation, we left the imperative constructs of Algorithm 1 unchanged but replaced all other imperative constructs with purely functional counterparts. In this process, we replaced hash-based data structures with purely functional ones. More specifically, we replaced `HashSet`s and `Dictionaries` with `Set`s and `Map`s from the F# library [13].

[2] If the network features unusually long identifiers, then the time complexity per connection cannot be described as constant.

| | | Toy | Fork | Lyngby | Florence |
|---|---|---|---|---|---|
| $\mathcal{G}^0$ | $V^0_{\max}$ | 240 | 236544 | $3.7 \times 10^{11}$ | $2.8 \times 10^{28}$ |
| | #vertices | 37 | 34849 | – | – |
| | parsing & WF | 0.10 | 0.10 | – | – |
| | graph generation | 0.039 | 6.7 | – | – |
| | game solving | 0.055 | 0.31 | – | – |
| | total time | 0.19 | 7.1 | – | – |
| $\mathcal{G}^1$ | $V^1_{\max}$ | 135 | 118734 | $1.8 \times 10^{11}$ | $1.4 \times 10^{28}$ |
| | #vertices | 24 | 14484 | – | – |
| | parsing & WF | 0.10 | 0.10 | – | – |
| | graph generation | 0.064 | 0.31 | – | – |
| | game solving | 0.025 | 0.042 | – | – |
| | total time | 0.19 | 0.46 | – | – |
| $\mathcal{G}^2$ | $V^2_{\max}$ | 135 | 12936 | $4.3 \times 10^7$ | $2.0 \times 10^9$ |
| | #vertices | 18 | 672 | 78530 | 253063 |
| | parsing & WF | 0.10 | 0.11 | 0.11 | 0.10 |
| | graph generation | 0.069 | 0.085 | 2.3 | 12 |
| | game solving | 0.021 | 0.028 | 0.11 | 0.40 |
| | total time | 0.19 | 0.22 | 2.6 | 12 |
| $\mathcal{G}^3$ | $V^3_{\max}$ | 105 | 6006 | $8.5 \times 10^6$ | $6.0 \times 10^8$ |
| | #vertices | 15 | 410 | 25467 | 220733 |
| | parsing & WF | 0.10 | 0.11 | 0.11 | 0.10 |
| | graph generation | 0.068 | 0.074 | 0.51 | 7.5 |
| | game solving | 0.024 | 0.024 | 0.057 | 0.47 |
| | total time | 0.19 | 0.21 | 0.67 | 8.1 |

**Table 3: The results of the experiments. The number of generated vertices is denoted #vertices and the checks for well-formedness are denoted WF. Time measurements are given in seconds and measurements less than 0.5 seconds have a large relative uncertainty due to the unpredictable timing of the garbage collector. A dash indicates that the computation aborts with an out-of-memory exception after allocating between 2.85 GB and 3.37 GB of memory.**

This purely functional version performed worse than the imperative version: The running time of using $\mathcal{G}^3$ on Lyngby Station was 4.8 seconds instead of 0.67 seconds and the running time of using $\mathcal{G}^3$ on Florence Station was 63 seconds instead of 8.1 seconds. This is roughly a factor 8 increase in the running time.

The purely functional `Set`s and `Map`s are implemented using balanced binary search trees [13], so lookups, insertions, and deletions run in $O(\log n)$ time. In comparison, these operations take expected $O(1)$ time on the `HashSet`s and `Dictionaries`. This difference in the asymptotic running times is the most likely explanation of the eight-fold increase in the overall performance.

# 7 CONCLUSION

Reachability games have been successfully used to synthesize signaling plans for railway networks:

(1) A mathematical model of railway networks was developed. This model describes the state space of a network and the transitions between these states. Furthermore, a simple DSL for describing railway networks has been defined.

(2) A game semantics for the DSL was given in terms of reachability games, where the states of a network act as vertices and the state transitions act as edges. Eve, the existential player, acts as the control system being responsible for the points and signals, and Adam, the universal player, acts as an adversary controlling the trains. It was shown how to obtain a correct signaling plan from Eve's winning strategy. This signaling plan will bring trains safely from their initial positions to their destinations.

(3) A tool for synthesizing correct-by-construction signaling plans from DSL specifications was implemented in the functional programming language F#.

An upper bound on the size of the original railway game was given. This bound is exponential in the number of trains, points, and signals. This renders computations on large networks impossible, so three abstractions were introduced, each leading to a game variant equivalent to the original. Each abstraction significantly reduces the upper bound on the game size.

The current implementation is able to generate correct signaling plans for real railway networks of substantial size. Specifically, in roughly 8 seconds, we are able to generate a signaling plan guiding four trains through (a fourth of) Florence Station. With five trains, the tool runs out of memory before the computation is complete. The size of the problems, for which synthesis is feasible, is approaching the size for which state-of-the-art verification methods can verify correctness of interlocking tables [24].

Due to the static and explicit generation of the full game graphs, the bottleneck of the current implementation is the memory needed during the graph generation process. The game-solving part uses an efficient implementation of a linear-time algorithm and for the networks considered that part alone never took more than 0.5 seconds.

To improve on the space limitations during game graph generation, the next step is to exploit on-the-fly methods, which will only generate the part of the game needed to find the winning area of Eve. Such techniques have been successfully used in, for example, [5]. An even further step would be to use symbolic methods to represent and solve games. This approach is used in [16], where very large parity games are solved symbolically. Because reachability games are simpler than parity games, this approach seems promising in the context of our railway game.

A functional programming approach proved to be most adequate during the development of the synthesis tool. The program design, expressed using F#'s type system, is very close to the mathematical definition of the railway-network model [17]. This has obvious benefits concerning the validation of correctness and the development time. The same applies to the game design, implementation, and construction parts.

Concerning efficiency, the experimental results show that it is the exponential growth of the size of the game graph that is the bottleneck – not the chosen data types nor the algorithms.

Imperative data structure were used to represent the game. In that way a more succinct representation could be obtained that also resulted also in more efficient implementations of the key algorithms.

## ACKNOWLEDGMENTS

Work of the second author was partially supported by the Danish Research Foundation for Basic Research within the IDEA4CPS project. We greatly appreciate suggestions from and discussions with Alessandro Fantechi and Anne Haxthausen. Furthermore, Alessandro provided the Florence-Station example.

## A GAME SOLVING

Below, we list the source code that implements Algorithm 1. In contrast to Algorithm 1, the function **solveReachability** takes as input a *reverse* game graph with information on the number of successors to each vertex:

```
1 let color (_, c, _) = c
2 let owner (_, _, plr) = plr
3
4 //val solveReachability :
5 //    GameGraph<'misc,'color> -> 'color ->
6 //    Winners<'misc,'color> * Strategy<'misc,'color>
7 let solveReachability
8         (graph : GameGraph<'misc,'color>) target =
9
10     let numPos = graph.Count
11     let win : Dictionary<Vertex<'misc,'color>, Player> =
12         Dictionary (numPos)
13     let strategy : Dictionary<Vertex<'misc,'color>,
14                               Vertex<'misc,'color>> =
15         Dictionary (numPos)
16
17     // Initialization
18     let queue = Queue ()
19     for KeyValue (v, _) in graph do
20         if color v = target then
21             win.[v] <- Eve
22             queue.Enqueue v
23         else
24             win.[v] <- Adam
25
26     // Breadth first search
27     while queue.Count > 0 do
28         let v = queue.Dequeue ()
29         for p in snd graph.[v] do
30             let (n, pred) = graph.[p]
31             graph.[p] <- (n - 1, pred)
32             if win.[p] = Adam && (owner p = Eve
33                                   || fst graph.[p] = 0)
34             then
35                 win.[p] <- Eve
36                 strategy.[p] <- v
37                 queue.Enqueue p
38
39     // Post-processing of the computed strategy
40     let strategy' =
41         Seq.filter (fun (KeyValue (u, v)) ->
42                         owner u = Eve)
43                    strategy
44     let strategy'' =
45         Seq.map (fun (KeyValue (u, v)) -> (u, v))
46                 strategy'
47     (win, strategy'')
```

## B GAME GRAPH GENERATION

Below, we list the source code of the function **graph**, which explicitly generates a *reverse* game graph. This graph carries information on the number of successors to each vertex:

```
1 //val graph :
2 //    GameVariant -> Network ->
3 //    (Games.GameGraph<State,Color>
4 //     * Games.Vertex<State,Color>)
5 let graph gameVariant (network : Network) =
6
7     let graph : Games.GameGraph<State,Color> =
8         Dictionary ()
9     let frontier :
10            HashSet<Games.Vertex<State,Color>> =
11         HashSet ()
12
13     // Graph-generation helper functions
14     let isNew v = not (graph.ContainsKey v)
15     let addVertex v = graph.[v] <- (0, List ())
16     let addEdge from' to' =
17         (snd graph.[to']).Add from' |> ignore
18     let setNumSuccessors v num plr =
19         match plr with
20         | Games.Player.Adam ->
21             countAdamMoves (int64 num)
22         | Games.Player.Eve  ->
23             countEveMoves (int64 num)
24         let (_, pred) = graph.[v]
25         graph.[v] <- (num, pred)
26
27     // Compute the directions of the trains
28     let dir = directions network
29
30     // Graph-generation loop
31     let rec graphAux () =
32         if frontier.Count = 0 then ()
33         else
34             let v = takeOne frontier
35             let mutable succs = null
36             match v with
37             | (_, _, Games.Player.Eve)  ->
38                 succs <- succEve gameVariant
39                                  network dir v
40                 setNumSuccessors v succs.Count
41                                  Games.Player.Eve
42             | (_, _, Games.Player.Adam) ->
43                 succs <- succAdam gameVariant
44                                   network dir v
45                 setNumSuccessors v succs.Count
46                                  Games.Player.Adam
47             for v' in succs do
48                 if isNew v' then
49                     frontier.Add v' |> ignore
50                     addVertex v'
51                 addEdge v v'
52             graphAux ()
53
54     // Initialize the search
55     let initialVertex = initialVertex network
56     match initialVertex with
57     | (_, Success, _) -> addVertex initialVertex
58     | (_, Crash, _)   ->
59         failwith ("the starting vertex "
60                   + "contains a failure state")
61     | (_, Nothing, _) ->
62         addVertex initialVertex
63         frontier.Add initialVertex |> ignore
64         graphAux ()
65
66     (graph, initialVertex)
```

# REFERENCES


[1] Banedanmark. 2009. The Signalling Programme, A total renewal of the Danish signalling infrastructure. http://uk.bane.dk/db/filarkiv/5636/Updated%20UK%20Brochure%20090317__BDK__Signalprogrammet__UKbrochure__%20www.pdf ISBN: 978-87-90682-04-0.

[2] Dietmar Berwanger. 2013. Graph Games with Perfect Information. (2013). Course notes, Master Parisien de Recherche en Informatique.

[3] Roderick Bloem, Krishnendu Chatterjee, Karin Greimel, Thomas A. Henzinger, Georg Hofferek, Barbara Jobstmann, Bettina Könighofer, and Robert Könighofer. 2014. Synthesizing robust systems. *Acta Inf.* 51, 3-4 (2014), 193–220. DOI: https://doi.org/10.1007/s00236-013-0191-5

[4] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. 2012. Synthesis of Reactive(1) designs. *J. Comput. Syst. Sci.* 78, 3 (2012), 911–938.

[5] F Cassez, A David, E Fleury, KG Larsen, and D Lime. 2005. Efficient on-the-fly algorithms for the analysis of timed games. *Lecture Notes in Computer Science* 3653 (2005), 66–80. DOI: https://doi.org/10.1007/11539452__9

[6] CDictionary 2016. Dictionary(TKey, TValue) Class (System.Collections.Generic). (2016). Retrieved May 28, 2016 from https://msdn.microsoft.com/en-us/library/xfhwa508(v=vs.110).aspx

[7] CHashSet 2016. HashSet(T) Class (System.Collections.Generic). (2016). Retrieved May 28, 2016 from https://msdn.microsoft.com/en-us/library/bb359438(v=vs.110).aspx

[8] CList 2016. List(T) Class (System.Collections.Generic). (2016). Retrieved June 3, 2016 from https://msdn.microsoft.com/en-us/library/6sh2ey19(v=vs.110).aspx

[9] Alexandre David, Huixing Fang, Kim Guldstrand Larsen, and Zhengkui Zhang. 2014. Verification and performance evaluation of timed game strategies. *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8711 (2014), 100–114. DOI: https://doi.org/10.1007/978-3-319-10512-3__8

[10] ERTMS 2016. The European Rail Traffic Management System. (2016). Retrieved May 18, 2016 from http://www.ertms.net/

[11] FParsec 2016. FParsec Documentation. (2016). Retrieved May 27, 2016 from http://www.quanttec.com/fparsec/

[12] Oliver Friedmann and Martin Lange. 2009. Solving Parity Games in Practice. *Lecture Notes in Computer Science* 5799 (2009), 182–196. DOI:https://doi.org/10.1007/978-3-642-04761-9__15

[13] FSet 2016. Collections.Set<'T> Class (F#). (2016). Retrieved June 4, 2016 from https://msdn.microsoft.com/visualfsharpdocs/conceptual/collections.set%5b%27t%5d-class-%5bfsharp%5d

[14] Erich Grädel, Wolfgang Thomas, and Thomas Wilke (Eds.). 2002. *Automata, Logics, and Infinite Games: A Guide to Current Research.* Lecture Notes in Computer Science, Vol. 2500. Springer.

[15] Michael R. Hansen and Hans Rischel. 2013. *Functional Programming using F#.* Cambridge University Press, Shaftesbury Rd, Cambridge CB2 8RU, UK.

[16] Gijs Kant and Jaco Van De Pol. 2014. Generating and solving symbolic parity games. *Electronic Proceedings in Theoretical Computer Science* 159 (2014), 2–14. DOI:https://doi.org/10.4204/EPTCS.159.2

[17] Patrick Kasting. 2016. *Synthesis of Railway Signaling Plans using Reachability Games.* BSc thesis. Technical University of Denmark.

[18] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. 2009. Temporal-Logic-Based Reactive Mission and Motion Planning. *IEEE Trans. Robotics* 25, 6 (2009), 1370–1381. DOI: https://doi.org/10.1109/TRO.2009.2030225

[19] Christof Löding. 2011. Infinite games and automata theory. In *Lectures in Game Theory for Computer Scientists.* Cambridge University Press, Shaftesbury Rd, Cambridge CB2 8RU, UK, 38–73. DOI:https://doi.org/10.1017/CBO9780511973468.003

[20] OFC 2016. The Open Edition of the F# compiler, core library and tools. (2016). Retrieved May 31, 2016 from https://github.com/fsharp/fsharp

[21] Amir Pnueli and Roni Rosner. 1989. On the Synthesis of a Reactive Module. In *Conference Record of the Sixteenth Annual Acm Symposium on Principles of Programming Languages.* Association for Computing Machinery, 271 Chemin de Saint-Julien, 06140 Biot, France, 179–190. DOI:https://doi.org/10.1145/75277.75293

[22] Amir Pnueli and Roni Rosner. 1989. On the Synthesis of an Asynchronous Reactive Module. *Lecture Notes in Computer Science* 372 (1989), 652–671. DOI:https://doi.org/10.1007/BFb0035790

[23] Don Syme, Adam Granicz, and Antonio Cisternino. 2015. *Expert F# 4.0.* Apress Media, 233 Spring Street, 6th Floor, New York, NY 1001, US.

[24] Linh Hong Vu. 2015. *Formal Development and Verification of Railway Control Systems - In the context of ERTMS/ETCS Level 2.* Ph.D. Dissertation. Technical University of Denmark.