

# A Simple Graph Theory and Its Application in Railway Signalling

Wai Wong  
Department of Engineering,  
University of Warwick,  
Coventry, England

## Abstract

*This paper describes a simple graph theory expressed in higher order logic and the applications of it in railway signalling. A theory of network based on the graph theory is developed to model the static aspects of railway track networks. Tools are being developed to produce formal specification of track networks and to generate control tables for specifying the interlocking required between routes.*

## 1 Introduction

Formal methods are important and indispensable tools for analysis and development of safety-critical systems. They help the designers to gain more insight into the systems, to give more confidence of producing a correct design.

In general, the first step is to analyze the system requirements and to produce a formal specification of the system. This specification captures the functional requirements, and it is usually written in a formal language, e.g., Z[8], VDM[5], or HOL[4]. From this specification, one can then derive some important properties using theorem provers, such as safety and liveness. Then, an implementation is developed. One can verify or prove that the implementation have the same functionality as the specification. This process is likely applied to several levels in any sizable practical systems.

During this process, it is inevitable to model the system by some mathematical structures. Graphs are often used, for example, to model communication networks. In the research in applying formal methods to railway signalling, I decided to model the railway track network by directed graphs. A simple graph theory developed in Higher-Order Logic(HOL) is described in Section 2. This is followed by the description of the theories modelling the railway track network. Then,

in Section 4, how these theories are used in signalling systems is explained and the paper concludes with a brief discussion on the advantage of this approach and makes proposals for further research.

## 2 The graph theory

Graph theory is a large branch of mathematics, and it has found applications in many diversified fields. The theory described in this section is only a first attempt of expressing a small portion of the conventional graph theory in Higher-Order Logic. I hope this will provide a starting point for modelling real-world systems using graphs.

### 2.1 The representation of graphs

First of all, the graph have to be represented by some structure in HOL. This representation should reflect the abstract properties of graphs, and should be general and flexible so as to be suitable for use in different applications. Based on these considerations, a type and a predicate have been chosen to represent graphs. The type is a pair of sets. For convenience, it is abbreviated in Meta Language (ML) as<sup>1</sup>:

```
let Vertex = ":"* and
    Edge = ":(* # * # **)" and
    Graph = ":(*)set # (* # * # **)set";;
```

where \* and \*\* are distinct polymorphic types. The first of the pair is a set of vertices and the second is the set of edges. This follows most conventional definitions of graphs. However, not every object of the type `~Graph` is a graph. A predicate is required to distinguish those which are graphs from others in the type. The definition of this predicate reads:

<sup>1</sup>`new.type_abbrev` cannot be used since these are polymorphic types.

**HOL Definition 1 (GRAPH\_DEF)**

```
"GRAPH ((V:~Vertex)set),(E:~Edge)set) =
  !e. e IN E ==>
    (((es e) IN V) /\ ((ed e) IN V))";;
```

where *es* and *ed* return the first and second field of the triple representing the edges, respectively. They are the source and destination of the edge *e*. This specifies that, to be a graph, all elements of the edge set must start from a vertex and end at a vertex. This is the dominant abstract property of a graph.

The type `:~Vertex` can be instantiated by any properly defined type which is required to label the vertices of a particular graph. Similarly, edges can be labelled by instantiating the last field of the triple of type `:~Edge` with a required type. A projection operator *elb* is defined to return this field, for instance, *elb e* is the label of edge *e*.

Having this definition of a graph, we need to assert that there exists at least one graph, i.e., the theorem

**HOL Theorem 1 (GRAPH\_EXISTS)**

$$\vdash \exists G. \text{GRAPH } G$$

A trivial example of graph is the null graph  $(\{\}, \{\})$ , and a more interesting graph shown in Figure 1(a) can be written in HOL as

```
{1, 2, 3, 4, 5, 6, 7, 8},
{(1, 2, a), (2, 3, b), (3, 4, c),
 (4, 1, d), (5, 6, e), (6, 7, f),
 (7, 8, g), (8, 5, h), (1, 5, i),
 (2, 6, j), (3, 7, k), (4, 8, l)}
```

As a consequence of this representation, all graphs are *directed*. This is because  $(v_1, v_2, x) \neq (v_2, v_1, x)$  for all  $v_1, v_2$  and  $x$ . To represent an undirected graph, each edge can be replaced by a pair of anti-parallel edges. Also, all graphs are *labelled*. To represent an unlabelled graph, the label field of the edges can be instantiated by the type `:one`.

For convenience, two constants, *VS* and *ES*, are defined to access the vertex set and the edge set. They are characterised by the following two theorems

**HOL Theorem 2 (VERTICES)**

$$\vdash \forall V E. \text{VS}(V, E) = V$$
**HOL Theorem 3 (EDGES)**

$$\vdash \forall V E. \text{ES}(V, E) = E$$

A *loop* is defined as an edge having identical ends:

**HOL Definition 2 (LOOP\_DEF)**

```
"LOOP (e:~Edge) = (es e = ed e)"
```

and *HAS\_LOOP* *G* is true if and only if the graph *G* contains a loop:

**HOL Definition 3 (HAS\_LOOP)**

```
"HAS_LOOP G = ?(e:~Edge).
  (e IN (ES G)) /\ (LOOP e)"
```

The graph shown in Figure 1(b) has a loop, the edge labelled *l*. A graph is said to have *multiple edges* if and only if *MULTI\_EDGE* *G* is true, where the definition of *MULTI\_EDGE* is:

**HOL Definition 4 (MULTI\_EDGE\_DEF)**

```
"MULTI_EDGE G = ?(e1:~Edge) e2.
  (e1 IN (ES G)) /\
  (e2 IN (ES G)) /\
  ~(e1 = e2) /\
  (es e1 = es e2) /\
  (ed e1 = ed e2)"
```

The graph shown in Figure 1(c) has multiple edges,  $(1, 2, a)$  and  $(1, 2, b)$ . Then, a *simple* graph is defined to be a graph containing neither loops nor multiple edges:

**HOL Definition 5 (SIMPLE\_GRAPH)**

```
"SIMPLE_GRAPH (G:~Graph) =
  (GRAPH G) /\
  ~(HAS_LOOP G) /\ ~(MULTI_EDGE G)"
```

and a *finite* graph is a graph whose vertex set and edge set are both finite.

**HOL Definition 6 (FINITE\_GRAPH)**

```
"FINITE_GRAPH (G:~Graph) =
  (GRAPH G) /\
  FINITE (VS G) /\ FINITE (ES G)"
```

Other abstract properties of graphs can be defined in similar way.

**2.2 Relationship between vertices and edges**

An edge is said to *incident with* the vertices which are the source or destination of the edge. It is said to *incident from* the source vertex and to *incident to* the

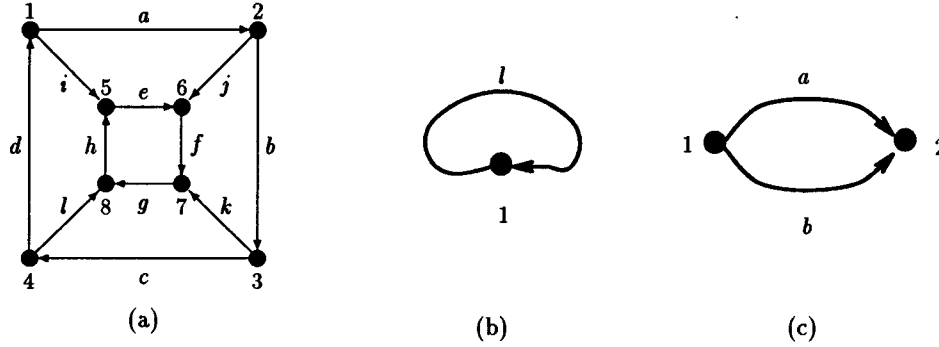


Figure 1: Example of simple graphs

destination vertex. The function `INCIDENT_WITH`, applying to a graph  $G$  and a vertex  $v$ , returns a set of edges which incident with the vertex  $v$ .

**HOL Definition 7 (`INCIDENT_WITH_DEF`)**

```
"INCIDENT_WITH (G:~Graph) v =
  {e | e IN (ES G) /\
    ((es e = v) /\ (ed e = v))}"
```

Let us name the graph in Figure 1(a) as  $G'$ , then

`INCIDENT_WITH  $G'$  1`

is equal to  $\{(1, 2, a), (4, 1, d), (1, 5, i)\}$ . Similarly, we can defined `INCIDENT_FROM` and `INCIDENT_TO` in HOL as below:

**HOL Definition 8 (`INCIDENT_FROM_DEF`)**

```
"INCIDENT_FROM (G:~Graph) v =
  {e | e IN (ES G) /\ (es e = v) }"
```

**HOL Definition 9 (`INCIDENT_TO_DEF`)**

```
"INCIDENT_TO (G:~Graph) v =
  {e | e IN (ES G) /\ (ed e = v) }"
```

The *degree* of a vertex is the total number of edges incident with it. The HOL definition of degree make use of the definitions of incidence and the cardinal number of sets:

**HOL Definition 10 (`DEGREE_DEF`)**

```
"DEGREE (G:~Graph) v =
  (IN_DEGREE G v) + (OUT_DEGREE G v)"
```

Similarly, *out-degree* is the number of edges incident from a vertex and *in-degree* is the number of edges incident to a vertex:

**HOL Definition 11 (`OUT_DEGREE_DEF`)**

```
"OUT_DEGREE (G:~Graph) v =
  CARD (INCIDENT_FROM G v)"
```

**HOL Definition 12 (`IN_DEGREE_DEF`)**

```
"IN_DEGREE (G:~Graph) v =
  CARD (INCIDENT_TO G v)"
```

Thus, `DEGREE  $G'$  1` is 3, `OUT_DEGREE  $G'$  1` is 2 and `IN_DEGREE  $G'$  1` is 1.

Two vertices are said to be *adjacent* if and only if there exists an edge between them. The predicate `VER_ADJA  $G$   $v_1$   $v_2$`  is true if there is an edge  $(v_1, v_2, x)$  or  $(v_2, v_1, y)$  for some  $x$  and  $y$ . The HOL definition of `VER_ADJA` is:

**HOL Definition 13 (`VER_ADJA_DEF`)**

```
"VER_ADJA G v1 (v2:*) = (GRAPH G) /\
  (v1 IS_VERTEX G) /\
  (v2 IS_VERTEX G) /\
  (? (e:~Edge). (e IS_EDGE G) /\
    (((es e = v1) /\ (ed e = v2)) /\
    ((es e = v2) /\ (ed e = v1))))"
```

In Figure 1(a), `VER_ADJA  $G'$  1 2` is T while `VER_ADJA  $G'$  1 3` is F. Similarly, two edges are adjacent if there is a vertex which is the destination of one and the source of the other.

**HOL Definition 14 (`E_ADJA_DEF`)**

```
"E_ADJA G e1 (e2:~Edge) = (GRAPH G) /\
  (e1 IS_EDGE G) /\ (e2 IS_EDGE G) /\
  ((ed e1 = es e2) /\ (ed e2 = es e1))"
```

A vertex  $v_2$  is a *successor* of another vertex  $v_1$  if and only if there exists an edge from  $v_1$  to  $v_2$ . The predicate `IS_SUC_VER` defined below indicates this relationship.

**HOL Definition 15 (IS\_SUC\_VER\_DEF)**

```
"IS_SUC_VER (G: ^Graph) v1 v2 =
  ?e. (e IS_EDGE G) /\
    (es e = v1) /\ (ed e = v2)"
```

The vertex 2 in  $G'$  is the successor of vertex 1. The converse of successor is the *predecessor*. The corresponding predicate is IS\_PRE\_VER:

**HOL Definition 16 (IS\_PRE\_VER\_DEF)**

```
"IS_PRE_VER (G: ^Graph) v1 v2 =
  ?e. (e IS_EDGE G) /\
    (ed e = v1) /\ (es e = v2)"
```

The functions SUC\_VERS and PRE\_VERS return the set of vertices which are successors and predecessors, respectively.

**HOL Definition 17 (SUC\_VERS)**

```
"SUC_VERS (G: ^Graph) v =
  {v' | (v' IS_VERTEX G) /\
    (IS_SUC_VER G v v'))"
```

**HOL Definition 18 (PRE\_VERS)**

```
"PRE_VERS (G: ^Graph) v =
  {v' | (v' IS_VERTEX G) /\
    (IS_PRE_VER G v v'))"
```

Referring to Figure 1(a),  $SUC\_VERS\ G'\ 1 = \{2, 5\}$  and  $PRE\_VERS\ G'\ 1 = \{4\}$ .

**2.3 Operations on graphs**

The primitive operations on graphs are insertion and deletion of a vertex or an edge. The definition of inserting a vertex is:

**HOL Definition 19 (INSERT\_VERTEX\_DEF)**

```
"INSERT_VERTEX v (G: ^Graph) =
  (v INSERT (VS G), (ES G))"
```

and the definition of inserting an edge is:

**HOL Definition 20 (INSERT\_EDGE\_DEF)**

```
"INSERT_EDGE e (G: ^Graph) =
  ((VS G),
  (( (es e) IS_VERTEX G) /\
    ((ed e) IS_VERTEX G)) =>
    (e INSERT (ES G)) | (ES G)))"
```

Note that to maintain the integrity of a graph, the only edges which can be inserted into it are those incident with vertices already in the graph. The reverse operations of insertion is DELETE\_VERTEX and DELETE\_EDGE. Their definitions are listed below:

**HOL Definition 21 (DELETE\_VERTEX)**

```
"DELETE_VERTEX (G: ^Graph) v =
  (((VS G) DELETE v),
  ((ES G) DIFF (INCIDENT_WITH G v)))"
```

**HOL Definition 22 (DELETE\_EDGE)**

```
"DELETE_EDGE (G: ^Graph) e =
  ((VS G), ((ES G) DELETE e))"
```

Note also that deleting a vertex must also delete all the edges incident with it. The following four theorems assert that the abstract property of a graph is maintained over these operations.

**HOL Theorem 4 (GRAPH\_INSERT\_VERTEX)**

$$\vdash \forall G v. \text{GRAPH } G \supset \text{GRAPH } (v \text{ INSERT\_VERTEX } G)$$
**HOL Theorem 5 (GRAPH\_INSERT\_EDGE)**

$$\vdash \forall G e. \text{GRAPH } G \supset \text{GRAPH } (e \text{ INSERT\_EDGE } G)$$
**HOL Theorem 6 (GRAPH\_DELETE\_VERTEX)**

$$\vdash \forall G v. \text{GRAPH } G \supset \text{GRAPH } (G \text{ DELETE\_VERTEX } v)$$
**HOL Theorem 7 (GRAPH\_DELETE\_EDGE)**

$$\vdash \forall G e. \text{GRAPH } G \supset \text{GRAPH } (G \text{ DELETE\_EDGE } e)$$

All of these operations are commutative. These facts are asserted by the following theorems:

**HOL Theorem 8 (INSERT\_VERTEX\_COMM)**

$$\begin{aligned} \vdash \forall G v_1 v_2. \\ v_1 \text{ INSERT\_VERTEX } (v_2 \text{ INSERT\_VERTEX } G) = \\ v_2 \text{ INSERT\_VERTEX } (v_1 \text{ INSERT\_VERTEX } G) \end{aligned}$$
**HOL Theorem 9 (INSERT\_EDGE\_COMM)**

$$\begin{aligned} \vdash \forall G e_1 e_2. \\ e_1 \text{ INSERT\_EDGE } (e_2 \text{ INSERT\_EDGE } G) = \\ e_2 \text{ INSERT\_EDGE } (e_1 \text{ INSERT\_EDGE } G) \end{aligned}$$
**HOL Theorem 10 (DELETE\_VERTEX\_COMM)**

$$\begin{aligned} \vdash \forall G v_1 v_2. \\ (G \text{ DELETE\_VERTEX } v_1) \text{ DELETE\_VERTEX } v_2 = \\ (G \text{ DELETE\_VERTEX } v_2) \text{ DELETE\_VERTEX } v_1 \end{aligned}$$

### HOL Theorem 11 (DELETE\_EDGE\_COMM)

$$\vdash \forall G e_1 e_2. \\ (G \text{ DELETE\_EDGE } e_1) \text{ DELETE\_EDGE } e_2 = \\ (G \text{ DELETE\_EDGE } e_2) \text{ DELETE\_EDGE } e_1$$

Another important operation on graphs is the *union* of two graphs. It is used later in building up networks. The union of two graphs  $G_1$  and  $G_2$  is defined to be the unions of their vertex set and edge set. The HOL definition reads:

### HOL Definition 23 (G\_UNION)

```
"G_UNION (G1: ^Graph) G2 =
  ((VS G1) UNION (VS G2), (ES G1) UNION (ES G2))"
```

The operation of G\_UNION is close within the set of all graphs, i.e., the union of any two graphs is a graph.

### HOL Theorem 12 (GRAPH\_UNION)

$$\vdash \forall G_1 G_2. \\ \text{GRAPH } G_1 \wedge \text{GRAPH } G_2 \supset \\ \text{GRAPH } (G_1 \text{ G\_UNION } G_2)$$

This operation is symmetric, associative and the union of a graph with itself results in itself. These properties are asserted by the following three theorems.

### HOL Theorem 13 (G\_UNION\_SYM)

$$\vdash \forall G_1 G_2. G_1 \text{ G\_UNION } G_2 = G_2 \text{ G\_UNION } G_1$$

### HOL Theorem 14 (G\_UNION\_ASSOC)

$$\vdash \forall G_1 G_2 G_3. \\ (G_1 \text{ G\_UNION } G_2) \text{ G\_UNION } G_3 = \\ G_1 \text{ G\_UNION } (G_2 \text{ G\_UNION } G_3)$$

### HOL Theorem 15 (G\_UNION\_IDENT)

$$\vdash \forall G. G \text{ G\_UNION } G = G$$

It is obvious that if  $v$  is a vertex of the union of  $G_1$  and  $G_2$ , then it is either a vertex of  $G_1$  or a vertex of  $G_2$ . Hence the following theorem:

### HOL Theorem 16 (VERTEX\_IN\_UNION)

$$\vdash \forall G_1 G_2 v. \\ v \text{ IS\_VERTEX } (G_1 \text{ G\_UNION } G_2) = \\ v \text{ IS\_VERTEX } G_1 \vee v \text{ IS\_VERTEX } G_2$$

## 2.4 Walks, trails and paths

One of the most important uses of graphs with respect to applications in railway signalling systems is the derivation of paths. Consider any two vertices  $v_1$  and  $v_2$  in a graph,  $v_2$  is *reachable* from  $v_1$  if there is a sequence of edges through which one can arrive at  $v_2$  from  $v_1$ . There are usually many different ways one can arrive at  $v_2$ . According to whether all the edges in the sequence are distinct, the sequences can be classified into several types. Different terms are used for each type of them. A *walk* is a sequence of edges  $[e_1, e_2, \dots, e_n]$ , where  $n > 0$ , such that the destination of  $e_i$  is equal to the source of  $e_{i+1}$  for  $1 \leq i < n$ . This implies that the edges are not necessarily distinct in a walk, i.e., it may pass through the same edge more than once. In HOL, a sequence of edges is represented by a list of edges, of type  $(\text{^Edge})\text{list}$ . A list of edges satisfies the predicate WALK if and only if it is a walk.

### HOL Definition 24 (WALK\_DEF)

```
"WALK G (w: (^Edge)list) =
  ~(NULL w) /\ (WALK_TAIL w G)"
```

### HOL Definition 25 (WALK\_TAIL\_DEF)

```
"(WALK_TAIL NIL (G: ^Graph) = T) /\
  (! (hd: ^Edge) t1. WALK_TAIL (CONS hd t1) G =
    (GRAPH G) /\ (hd IN (ES G)) /\
    ((NULL t1) /\ (WALK_TAIL t1 G) /\
     (ed hd = es (HD t1))))"
```

Here, the recursive predicate WALK\_TAIL guarantees the list of edges forms a walk by checking whether the source of the next edge is equal to the destination of the current edge. Also, the degenerated case, the null list, is not to be considered as a walk.

A *trail* is a walk which contains no repeated edges, i.e., all edges in the sequence are distinct. However, it may pass through the same vertex more than once, so contains a cycle. The definition of TRAIL is:

### HOL Definition 26 (TRAIL\_DEF)

```
"TRAIL (G: ^Graph) (l: (^Edge)list) =
  (WALK G l) /\ (UNIQUE_EL l)"
```

The clause UNIQUE\_EL  $l$  makes sure that all elements in the list  $l$  are distinct, i.e., no edges in a trail are repeated.

A *path* is a trail which passes through any vertex at most once, i.e., there is no cycle in a path. PATH  $G l$  if and only if  $l$  is a path in the graph  $G$ .

### HOL Definition 27 (PATH\_DEF)

```
"PATH (G: ^Graph) (l: (^Edge)list) =
  (TRAIL G l) /\ (UNIQUE_EL (VER_LIST l))"
```

where `VER_LIST` returns a list of vertices that  $l$  passes through. The clause `UNIQUE_EL (VER_LIST l)` guarantees that all vertices passed through by  $l$  are distinct. In the application in railway signalling, paths are the most important type of lists, therefore, some theorems about paths are described below.

The entry of a path is defined to be the source vertex of the first edge in the sequence:

**HOL Definition 28 (PATH\_ENTRY)**

"PATH\_ENTRY (l:(^Edge)list) = es (HD l)"

and the exit of a path is the destination vertex of the last edge in the sequence. It is defined in terms of the exit of a walk (`WALK_EXIT`):

**HOL Definition 29 (PATH\_EXIT\_DEF)**

"PATH\_EXIT (p:(^Edge)list) = WALK\_EXIT p"

**HOL Definition 30 (WALK\_EXIT\_DEF)**

"WALK\_EXIT (CONS (hd:(^Edge) t1) =  
(NULL t1) => (ed hd) | (WALK\_EXIT t1))"

An existing path can be extended by `CONS`ing an edge to the front of it. The following theorem expresses what kind of edge can be put in front of a path to make a new path:

**HOL Theorem 17 (PATH\_CONS)**

$$\begin{aligned} \vdash \forall p h G. \text{PATH } G p \wedge h \text{ IS\_EDGE } G \wedge \\ (\text{PATH\_ENTRY } p = \text{ed } h) \wedge \neg(\text{es } h = \text{ed } h) \wedge \\ \neg \text{es } h \text{ IN VER\_SET } p \wedge \neg h \text{ IN EL\_SET } p \supset \\ \text{PATH } G (\text{CONS } h p) \end{aligned}$$

where `VER_SET` returns a set of vertices  $p$  passes through and `EL_SET` returns the set containing all elements of the list  $p$ . Two existing paths can also be concatenated to form a new path providing certain conditions hold. This is expressed in the following theorem:

**HOL Theorem 18 (PATH\_CAT)**

$$\begin{aligned} \vdash \forall G p_1 p_2. \text{GRAPH } G \wedge \text{DISJ\_PATH } G p_1 p_2 \wedge \\ (\text{PATH\_EXIT } p_1 = \text{PATH\_ENTRY } p_2) \wedge \\ \neg \text{PATH\_ENTRY } p_1 \text{ IN VER\_SET } p_2 \supset \\ (\exists p_3. \text{PATH } G p_3 \wedge \\ (\text{PATH\_ENTRY } p_3 = \text{PATH\_ENTRY } p_1) \wedge \\ (\text{PATH\_EXIT } p_3 = \text{PATH\_EXIT } p_2) \wedge \\ (p_3 = \text{APPEND } p_1 p_2)) \end{aligned}$$

This says that two paths can be concatenated to form a new path if they are disjoint and the end of the first is the beginning of the second.

If  $l$  is a path in  $G_1$ , then it is still a path in the union of  $G_1$  and another graph, say  $G_2$ .

**HOL Theorem 19 (PATH\_G\_UNION)**

$$\begin{aligned} \vdash \forall l G_1 G_2. \\ \text{GRAPH } G_1 \wedge \text{GRAPH } G_2 \wedge \text{PATH } G_1 l \supset \\ \text{PATH } (G_1 \text{ G\_UNION } G_2) l \end{aligned}$$

If  $l$  is a path in  $G$ , then it is a path of the graph resulting from inserting an edge or a vertex into  $G$ .

**HOL Theorem 20 (PATH\_INS\_EDGE)**

$$\vdash \forall e G. \text{PATH } G l \supset \text{PATH } (e \text{ INSERT\_EDGE } G) l$$

**HOL Theorem 21 (PATH\_INS\_VERTEX)**

$$\vdash \forall e G. \text{PATH } G l \supset \text{PATH } (e \text{ INSERT\_VERTEX } G) l$$

Other definitions, such as subgraphs, graph isomorphism, have also been formulated in a similar manner, and some properties of them have been proved.

### 3 The railway track network

Let us now consider how to model the railway track network. We are interested in the topology of the track and signal layouts, and not their physical dimensions, i.e., how the network is formed and what are the relationships between component parts. In the first subsection below, two theories in HOL are developed to model the functions of the railway track components and signals. The second subsection will show how to combine these theories with the `graph` theory to model the track network.

#### 3.1 Railway network components

The railway track network consists of several types of components which can be divided into two groups: the track components and the signals. The functional properties of them in a network is expressed in the HOL theories `part` and `signal`.

### 3.1.1 The theory PART


This theory describes the track components of railway networks. These components are called *parts*. There are four distinct kinds of them: buffer, plain track, diamond crossing and point.




A type `:Part` is defined to represent these track components. The definition reads:



#### HOL Definition 31 (Part.Axiom)

```
'Part = BPART num |
  TPART num Tcir |
  DPART num Tcir (num#num) (num#num) |
  PPART num Tcir Point (num#num#num)'
```

Each part has an identification number, and except buffers, an associated track circuit. The type `Tcir` represents a track circuit which has its own ID number and a state function. A track circuit may be shared by more than one part.

The first two kinds of parts are simple. A `BPART` represents a buffer which is the dead end of the railway. It is drawn as  in a layout. A `TPART` represents a section of plain track.

A diamond crossing is represented by a `DPART` part. This is where two tracks cross each other. This is drawn as  or . The `num#num` fields indicate the ID numbers of the adjacent parts. A movement through the diamond crossing can only be made between the parts indicated in the same pair. There are some crossings known as slip crossing which look like . These are modelled as three parts: a diamond crossing with a point at either end of it.

A point is where the track has a branch which is shown in layouts as  or . This is represented by a `PPART`. The `Point` field represents the point including its identification and its state. A point can be in one of the following states at any given time: *NORMAL*, *REVERSE* or *MOVING*. The triple of `num` indicates the adjacent parts. The first is the part at the trailing end, and the second and the third field are the parts at the normal and reverse ends, respectively.

### 3.1.2 The theory SIGNAL

Like track components, there are several kinds of signals. On a particular signal post, a single signal or a combination of several kinds of signals can be installed. Each signal post is represented by an object of type `:Signal` which is defined as follows:

#### HOL Definition 32 (Signal.Axiom)

```
'Signal = SIGNALM num Msig |
  SIGNALMJ num Msig Jsig |
  SIGNALS num Shsig'
```

```
SIGNALMS num Msig Subsig |
SIGNALMSJ num Msig Subsig Jsig |
SIGNALS num Shsig'
```

A constructor is provided for each type or combination of types of signals. The type `Msig` represents main signals, `Jsig` is for junction indicators, `Subsig` is for subsidiary signals and `Shsig` for shunting signals. Each of these signals contains a function which return its current state.

The most important concept of the operation of signals is the ON or OFF states. A signal is said to be ON if it is showing the aspect which indicates that the train must not pass that signal. On the other hand, a signal is OFF if it is showing any other aspect. If a signal is neither ON nor OFF, it must be faulty.

### 3.2 The NETWORK theory

Having the specifications of the parts, signals and a generic graph theory, They can now be used to model a complete track layout. This is developed in the theory **NETWORK**.

Since the track components are the basic building block of the track network, they are taken as the vertices of a graph. The type of vertices is instantiated by the type `:Part`. The edges of the graph then represent the connection between the parts. Physically, this is the point where two adjacent parts meet, which is known as the *join*. Two edges are placed between adjacent parts because the edges are directed and a train can move between two parts in both direction. A signal is attached to the edge which is incident to the part whose entry is controlled by the signal. For example, the signal in Figure 2 is attached to the edge from *T1* to *T2*. It controls the train movement from *T1* to *T2* along this edge. However, a reverse movement from *T2* to *T1* will follow the anti-parallel edge and is not controlled by this signal.

A type `:Edge` is defined for the edges. This becomes an instance of the polymorphic type `**` in the abbreviation (in ML) by `:~Graph`. Objects of this type consists of either a *join* or a combination of a join and a signal. Thus, we can now define an abbreviated type **Network** for track networks:

```
":(Part)set#(Part#Part#Edge)set"
```

This abbreviated type is an instance of the generic type `:~Graph`.

However, not all objects of this type are a proper network. A predicate **NETWORK** is required to distinguish the real railway track network from others. It defines a subset of objects of type `:Network` to be

proper railway track networks. This is to say that if any object of type `:Network` satisfies the predicate `NETWORK`, it is a representation of a physical track layout which can be constructed following the rules of a railway authority. The rules used in this study are taken from British Rail's current practice[7]. These rules are embedded in the definition of `NETWORK` which is defined inductively, i.e., a network can be built up by adding component parts to an existing network.

### HOL Definition 33 (`NETWORK_DEF`)

```
"NETWORK (N:Network) =
  !P. ((!n. P({(BPART n)}, { }))) /\
    (!n t. P({(TPART n t)}, { }))) /\
    (!n t p n3. P({(PPART n t p n3)}, { }))) /\
    (!n t n1 n2. P({(DPART n t n1 n2)}, { }))) /\
    (!N p1 p2. (P N /\
      (p1 IS_VERTEX N) /\ (p2 IS_VERTEX N) /\
      (NFC p1 N) /\ (NFC p2 N))
    ==>
    (!e1 e2. P((p2,p1,e2) INSERT_EDGE
      ((p1,p2,e1) INSERT_EDGE N))))
  (!N1 N2 p1 p2. (P N1 /\ P N2 /\
    (p1 IS_VERTEX N1) /\ (p2 IS_VERTEX N2) /\
    (((VS N1) INTER (VS N2)) = EMPTY) /\
    (NFC p1 N1) /\ (NFC p2 N2))
    ==>
    (!s1 s2. P(NJOIN N1 p1 s1 N2 p2 s2))))
==> P N"
```

This specifies that a single track component (a part) on its own is a legal network, and there are two ways of building up larger networks:

1. by adding edges to connect up some parts in an existing network;
2. by connecting two existing networks at some parts using the operator `NJOIN`.

The two networks to be combined together must not already overlapped. The parts to be joined must satisfy the predicate `NFC` which stands for Not-Fully Connected. Its definition is:

### HOL Definition 34 (`(NFC_DEF)`)

```
"(NFC (BPART n) (N:Network) =
  (IN_DEGREE N (BPART n) < 1)) /\
  (NFC (TPART n t) (N:Network) =
  (IN_DEGREE N (TPART n t) < 2)) /\
  (NFC (PPART n t p n3) (N:Network) =
  (IN_DEGREE N (PPART n t p n3) < 3)) /\
  (NFC (DPART n t n1 n2) (N:Network) =
  (IN_DEGREE N (DPART n t n1 n2) < 4))"
```

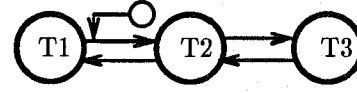


Figure 2: A simple network.

Suppose that all vertices in the network in Figure 2 are of `TPART`, then the middle one, namely `T2` is fully connected, i.e.,  $NFC\ T2\ N = F$  while the other two parts are not fully connected.

The `NJOIN` operation connects two networks by adding edges between two not-fully-connected vertices. The labels of these edges are given as arguments. Its definition is:

### HOL Definition 35 (`NJOIN_DEF`)

```
"NJOIN (N1:Network) (n1:Part)
  (s1:Edge) N2 n2 s2 =
  ((n1,n2,s1) INSERT_EDGE ((n2,n1,s2)
    INSERT_EDGE (N1 G_UNION N2)))"
```

In Figure 3, the network `N1` has a not-fully-connected `PPART P1` and network `N2` has a not-fully-connected `TPART T2`. These two networks can be joined together to form a new network. The operation can be written in HOL as:

`NJOIN N1 P1 J1 N2 T2 J1`

The result of this is the network shown on the left hand side of Figure 4, and the corresponding track layout is shown on the right hand side.

Since the type `:Network` is an instance of `:Graph` and all networks are built according to the specification in `NETWORK`, they are also graphs, hence the following theorem:

### HOL Theorem 22 (`NETWORK_GRAPH`)

$\forall N. NETWORK\ N \supset GRAPH\ N$

Since the theory `graph` is an ancestor of `NETWORK`, all functions defined in it are available and all theorems proved are applicable in networks. For example, the theorem `PATH_CAT` described above is still true in a network. Therefore, the functions defined in the `graph` theory can be used for reasoning in networks, for example, about finding routes and proving them.



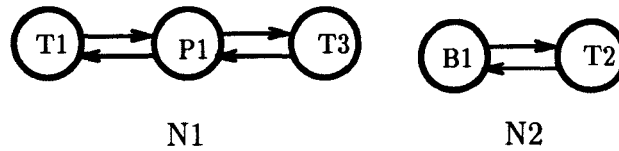


Figure 3: Two simple networks.

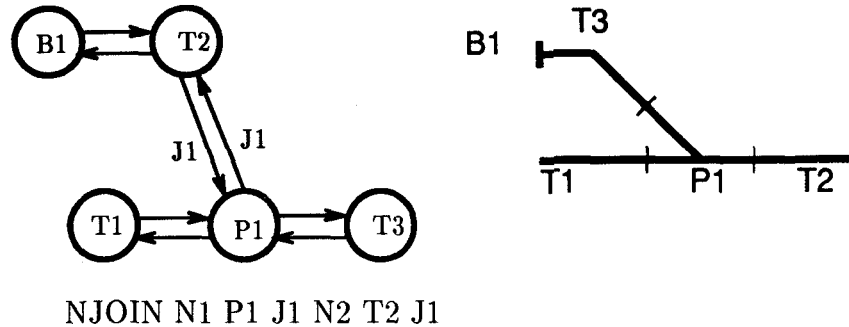


Figure 4: A network formed by joining two simple ones.

## 4 Applications of the theories

### 4.1 Procedures of designing signalling scheme

In British Rail, a system of electronic interlocking known as *Solid-State Interlocking* (SSI)[2] is replacing the conventional electromechanical interlockings. This new technology has been successfully installed in large, busy mainline stations, such as York and Liverpool Street in London. The SSI system utilizes microprocessor technology. To ensure the safety and reliability, it employs a mixture of triplex and duplex redundancy in both hardware and software. The SSI systems are data-driven. For a particular signalling scheme, a set of geographic data, which encodes all the required functions within it, is used by the SSI to control the signalling equipments, including points and signals.

Briefly, the procedures of designing and implementing new signalling schemes using SSI in British Rail[6] is as follows:

1. specify the required track and signalling layout;
2. produce *control tables* which are the conventional means of specifying the interlocking and control requirements of the layout;
3. generate *geographic data* which forms the database used by the SSI system;

4. test the geographic data on simulator;

5. install the SSI with EPROMs containing the geographic data.

One of the major task of the railway signalling engineer is to produce the control tables required for a particular track and signalling layout. The control tables have a well-defined syntax and semantics for the interlocking functions, and are very well understood by signalling engineers. The geographic data are extracted from this table and written in a special-purpose high-level programming language. This is then compiled into data object codes and installed in the SSI.

The theories described in previous sections present a top level abstract model of the static, topological properties of railway track networks. The applications of these theories are mainly in the design stage of new signalling schemes, i.e., the first two steps in above procedures.

### 4.2 Track layout compiler

The first application is to produce a formal representation of the track layout in the network model. This is the first step if any formal reasoning about the network is to be carried out. To facilitate the creation of track layout and the generation of its formal description, a prototype of CAD tool has been developed. It is called railway track compiler. It consists of

a graphical user interface, an input checker, a parser and code generator.

The graphical user interface allows the user to input track components and signals, and to build up a network on a graphics screen. This graphical presentation of track layout provides a clear, direct view of the layout to the engineer. It can also send the layout to a printer to produce high quality output (see Figure 5). The whole layout is divided into a grid of cells. Each cell can contain at most one track components plus an optional signal and annotations. The interface is menu-driven.

The most important feature of this compiler is that the internal representation of the layout is based on the abstract model defined in the **NETWORK** theory. The type of parts which can be put into a cell is dependent on the parts in the adjacent cells. The input checker validates the user inputs according to the rules defined in the abstract model. The menu will show only those parts which is 'legal' in the current cell. This helps the user to select the right component to be placed into the cell.

The parser and code generator takes the internal representation of the layout and converts it to a formal specification in HOL. It uses the definition of **NETWORK** described in Section 3.2 to check the validity of the layout. It ensures that the resulting layout is always a proper network. For example, the layout shown in Figure 5 can be written in HOL as:

```
{ T100, T101, T102, T103, T104, T105,
  T106, T107, T108, T109, T110, T111,
  T112, P200, P201, D300 },
{ (T100,T101,S10), (T101,T100,j1),
  (T101,P200,j2), (P200,T101,j2),
  (P200,T104,j3), (T104,P200,j3),
  (P200,D300,j4), (D300,P200,j4),
  (T104,T105,j5), (T105,T104,j5),
  (T105,T106,S14), (T106,T105,j6),
  (D300,T102,j7), (T102,D300,j7),
  (T102,T103,S12), (T103,T102,j8),
  (T107,T108,S11), (T108,T107,j9),
  (T108,D300,j10), (D300,T108,j10),
  (D300,P201,j11), (P201,D300,j11),
  (T109,T110,S13), (T110,T109,j12),
  (T110,P201,j13), (P201,T110,j13),
  (P201,T111,j14), (T111,P201,j14),
  (T111,T112,S15), (T112,T111,j15) }
```

where  $T_n$  has been defined as  $(TPART\ n\ ncir)$ , and similarly, the points  $(P201, P202)$  and the diamond crossing  $(D300)$  have been defined as the appropriate kind of parts.  $ncir$  is the track circuit number associated with the part. The graph corresponding to this network is shown in Figure 6. The usefulness of this

specification of the layout will be shown in the next subsection.

### 4.3 Generation of control table

The second application is the generation of control tables for the track layout. To do this, there are two problems we have to solve: the first is to find the possible routes; the second is to determine the conditions which have to be satisfied for each route. These conditions form the entries in the control table.

A *route* is a path which starts at a signal, terminates at another signal and satisfies the following extra restrictions:

1. when passing a point, it cannot enter from a normal edge and continue to a reverse edge or vice versa;
2. when passing a diamond crossing, it cannot move from one leg to another.

For example, a train cannot move from D300 to T104 through P200 in a single route. Neither can a train move from T108 to P200 through D300.

A route from S10 to S12 is the following list of edges:

```
S10S12 = [
  (T100,T101,S10); (T101,P200,j2);
  (P200,D300,j4); (D300,T102,j7) ]
```

Notice that the entry of the route is the node before the entry signal, and the exit of the route is the node before the exit signal. The definition of a route in HOL is:

#### HOL Definition 36 (ROUTE\_TAIL\_DEF)

```
"(ROUTE_TAIL [] = T) /\
 (ROUTE_TAIL (CONS (h:Part#Part#Edge) t) =
  (t = []) \/\
  ((IS_PPART (ed h) =>
    (TRAILING_EDGE h =>
      (NORMAL_EDGE (HD t) \/\
        REVERSE_EDGE (HD t)) |
      TRAILING_EDGE (HD t)) |
    (IS_DPART(ed h) => SAME_LEG h (HD t) | T)) /\
  ROUTE_TAIL t))"
```

#### HOL Definition 37 (ROUTE\_DEF)

```
"ROUTE (N:Network) r =
  (PATH N r) /\ (ROUTE_TAIL r) /\
  (IS_SIG (elb (HD r)))"
```

After defining what a route is, attempt can be made to find all the possible routes in a layout. There are

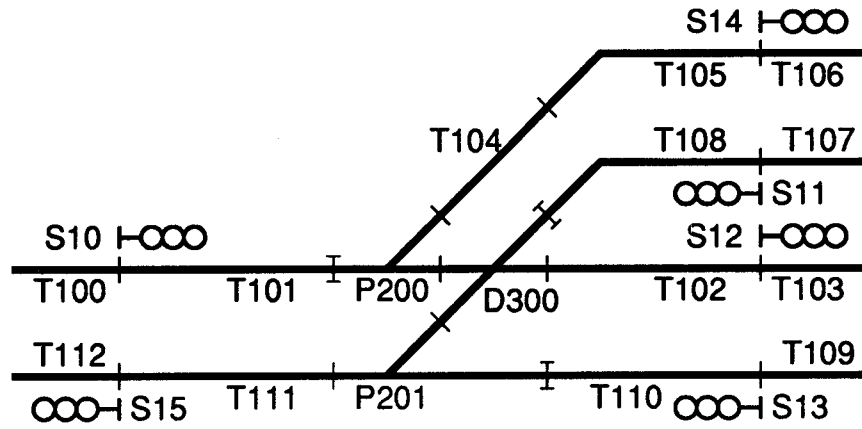


Figure 5: An example of track layout

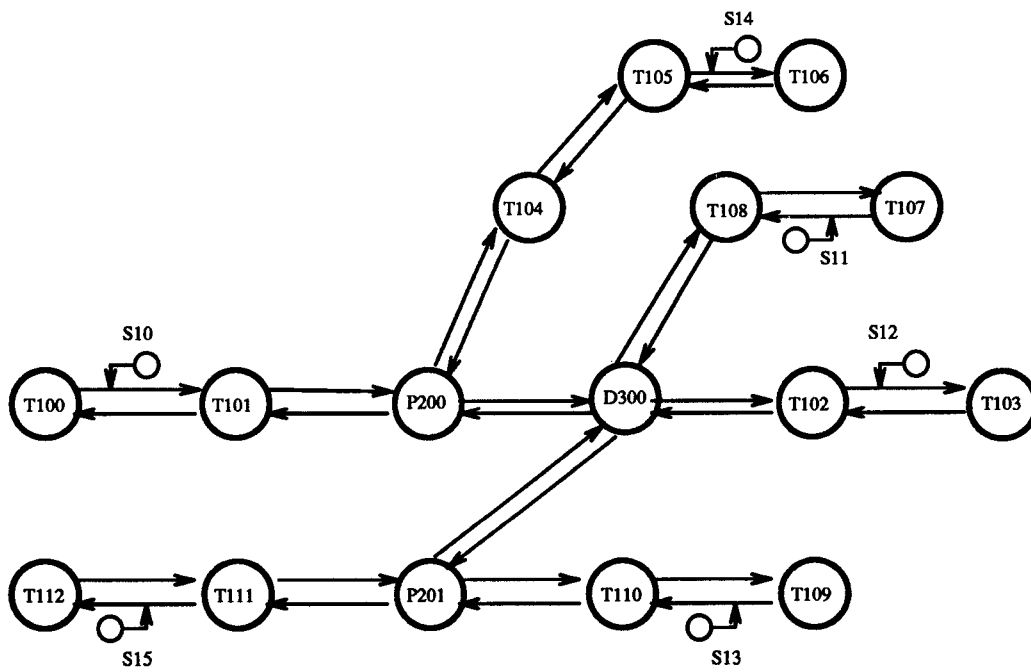


Figure 6: A graph representing the track layout in Figure 5

many well known algorithms for finding paths in a graph, for example, the Dijkstra's shortest path algorithm [1] and the Murchland's all paths algorithm [9]. The Murchland's all paths algorithm find all the paths between two given vertices. It is particular useful to our problem. Since a network is an instance of graph, these can be used directly to find a path in a network.

However, not all paths are 'legal' routes. This can easily be distinguished using the predicate **ROUTE** after all the paths is found. The algorithm may be augmented to search for paths which meet our conditions for routes. A program implementing this algorithm can be developed to work out all possible paths in a network. It will take as its input the formal specification of a network, and return a list of routes.

We can now consider the problem of generating control tables. A control table for the route from **S10** to **S12** is shown in Table 1. The first task of generating control tables is to codify the safety rules. These rules specify the interlocking requirements for setting up a route, or in the railway jargon, *proving* the route.

Considering only the simplest situations, the safety requirements are listed below with the indication of corresponding column in the example table<sup>2</sup>:

1. all track circuits on the route must be clear (column 2);
2. all points on the route must be set, locked and detected at the correct position according to the travelling direction of the route (column 4 and 5);
3. the exit signal must be in working order, i.e., showing either **ON** or **OFF** aspect (column 6);
4. the entry signal to the conflicting route must be proved **ON**(column 7);
5. the track circuits from the entry signal of the conflicting route to the point of conflict must be clear(column 2).

These requirements can be specified in HOL using functions of type  $:(\text{Edge})\text{list} \rightarrow (*)\text{list}$  where the  $:(\text{Edge})\text{list}$  is the route and  $:(*)\text{list}$  is the list of required track circuits or, points or signals. For example, the function which returns a list of points required normal is **NORM\_POINTS**:

#### HOL Definition 38 (NORM\_POINTS\_DEF)

"**NORM\_POINTS** 1 = **FLAT** (**MAP** **NORM** 1)"

<sup>2</sup>column 3 is used for delayed yellow approach which is not considered in this simple example.

It is defined in terms of **NORM** which takes an edge and returns a list of points required normal if a movement is made from the source node to the destination node.

#### HOL Definition 39 (NORM\_DEF)

"**NORM** (p1,p2,(e:Edge)) =  
 ((**IS\_PPART** p1) /\n  
 (**PART\_PNT\_NORMAL** p1 = **PART\_ID** p2)) =>  
 [**PART\_POINT** p1] | []"

The contents of column 4 in the control table is exactly the result of **NORM\_POINTS** **S10S12**. Functions for other columns are defined in a similar way.

One complication is the handling of the rules 4 and 5 which requires the search of the conflicting routes. A conflicting route is one that crosses the current route at a diamond crossing or converge to the current route at a point. In our example, the route from **S11** to **S15** is in conflict with route **S10S12**.

The task of searching for conflicting routes is make easy by the fact that all possible routes in a network have be found by the program mentioned above. The conflicting routes can be extracted from them by the following function:

#### HOL Definition 40 (CONFLICT\_ROUTES\_DEF)

"(**CONFLICT\_ROUTES** [] n = []) /\n  
 (**CONFLICT\_ROUTES** (**CONS** h t) n =  
**APPEND** ((n **IN** (**VER\_SET** h)) => h | [])  
 (**CONFLICT\_ROUTES** t))"

This function takes as its first argument a list of routes and returns a list of routes which are in conflict at the vertex n. Suppose that *ll* is the list of all routes in the network, then

**CONFLICT\_ROUTES** *ll* **D300**

will return the list containing only a single route **S11S15**. The entry signal is the signal on the first edge of this list. The required track circuits can be determined by

**MAP** **PART\_CIRCUIT** (**FST**(**CUT**(**V\_LIST** *l*) **D300**))

where **PART\_CIRCUIT** returns the circuit in a part, and **V\_LIST** returns a list of vertices of a path except the entry vertex. **CUT** *l* *n* splits the list *l* into a pair of sublists just before the element *n*. The only track circuit on this route up to the diamond crossing is **T108**.

For many networks, there are usually more than one possibly route between two given nodes. This will simply generate a row for each route in the control table. Which route is to be chosen for a particular

ROUTE	TRACK CIRCUITS		REQUIRE POINTS		SIGNALS	
	CLEAR	OCCUPIED	NORMAL	REVERSE	ALIGHT	ON
S10S12	t101,t200		P200		S12	
	t300,t102					
	Protect from conflicting traffic					
	t108					S11

Table 1: An example of control table

train depends on many other conditions. This may well require human intervention, and it is not within the scope of this paper.

In summary, the above description has shown how can a formal specification of a track network be generated from graphical user inputs, and how control tables can be generated for the interlocking requirements automatically. The conversion from the control tables to the geographic data can also be automated. Since the control tables are very familiar to the signalling engineers, it is very important that any automation attempts should not bypass the production of such tables.

#### 4.4 Proving routes

Further applications based on the theories described above is also being considered. For example, the reasoning of route proving in the control of interlocking. This requires the notion of time to be represented in a dynamic network.

The state of a network at any given time is determined by the states of its constituent parts. These include the states of the track circuits, of the points and of the signals. The possible states of track circuits are: *CLEAR*, *OCCUPIED* and *LOCKED*. The track circuits are designed as fail safe, i.e., the output from a faulty circuit is always *OCCUPIED*. The *LOCKED* state indicates that the section of track the circuit is on has been reserved for a route and/or a train is about to enter this section. The states of points and signals have been described in the section 3.1.1 and 3.1.2.

Since the dynamic states of the parts have been specified in their definitions, a dynamic network can be based on the current state of its parts. At any given time, a dynamic network is a subgraph of the underlying static network. The argument of the safe operation of the network can be carried out on this framework.

Suppose that at a given time  $t$ , the points P200 and P201 are both at *NORMAL* position. The network will be reduced to that shown in Figure 7. In this state, there are only two possible routes, namely S10S12 and

S13S15. If the track along these routes are clear, they can be proved using the function described below.

Based on the requirements listed in the previous subsection, the specification of proving a route can be written in HOL as:

#### HOL Definition 41 (ROUTE\_PROVED)

```
"ROUTE_PROVED (N:Network) r t =
  let p2 = PATH_EXIT r in
  let o1 = OVERLAP N r p2 in
  (ROUTE N r) =>
    ((EVERY (\x. TC_CLEAR x t) (TCIRCUITS r)) /\
     (EVERY (\x. TC_CLEAR x t) (TCIRCUITS o1)) /\
     (EVERY (\p. PNT_NORMAL p t) (NORM_POINTS r)) /\
     (EVERY (\p. PNT_REVERSE p t) (REV_POINTS r)) /\
     (EVERY (\p. PNT_NORMAL p t) (NORM_POINTS o1)) /\
     (EVERY (\p. PNT_REVERSE p t) (REV_POINTS o1)) /\
     (EVERY (\s. OFF s t) (MAP (EDGE_SIGNAL o elb) o1))) /\
    (~ (SIGNAL_FAULT (EDGE_SIGNAL p2) t)) /\
    (PROVE_CONFLICT_ROUTE N r t)) | F"
```

In this definition,  $r$  is the route to be proved,  $p2$  is the exit parts of the route, and  $o1$ , known as *overlap*, is a list of edges representing the short section of track ahead of the exit of the route. The function of  $o1$  is to protect the train from overrunning the exit signal. So, if at time  $t$ , all the required track circuits, points and signals are at the required states, *ROUTE\_PROVED* is true. This definition actually encodes within it all the safety rules specifying the conditions of setting up a route. The function *PROVE\_CONFLICT\_ROUTE* checks the states of the track circuits, points and signals in the conflicting routes to see whether they satisfy the required conditions. The methods of working out these parts and signals has been described in the previous subsection.

A point is said to be locked if the condition under which it can be moved from *NORMAL* to *REVERSE*, or vice versa, is not satisfied. For example, it requires the signal S10 to be ON and the track circuit t101 and t200 clear to move P200. Some points also depend on the position of other points or have to be moved in synchronization with other point(s). These

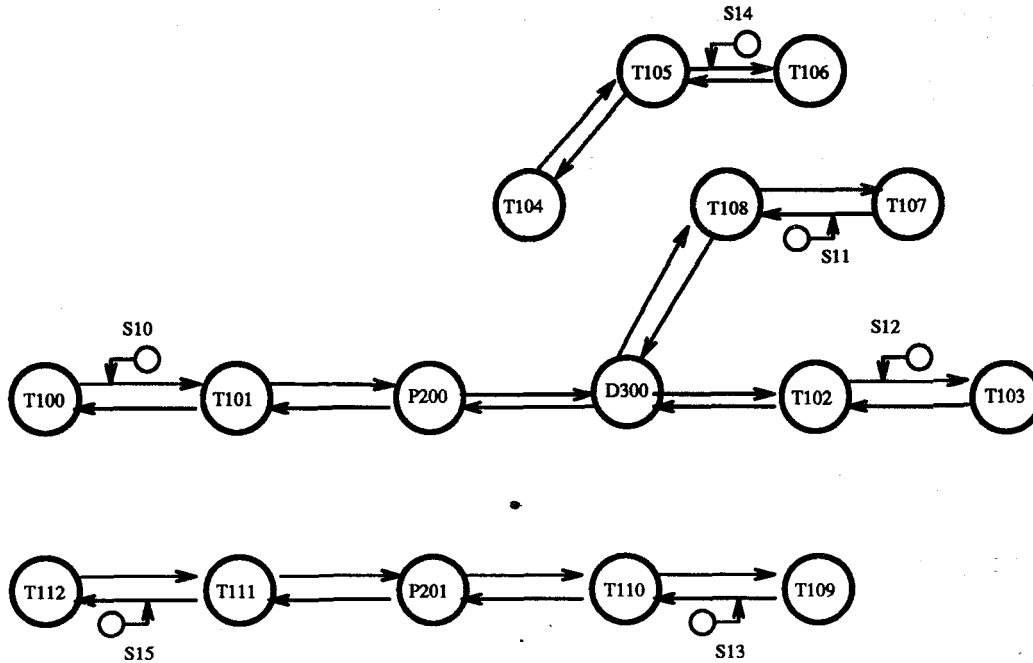


Figure 7: A graph representing the layout of Figure 5 at a given state

rules can be embedded in the functions for controlling the points. Similarly, the control of signals can be specified. Thus, the points can be moved to the required position to form a route and signals to be set up as required.

After a route is proved, the entry signal can be turned off to allow a train to enter the route. When a train is approaching or entering a route, all parts of the route must be locked, i.e., points are not allowed to move and track can not be used for another route.

The control of the complete interlocking system is formed by combining these specifications in a consistent way. The next step will be the derivation of safety properties of such interlocking. One of the important properties is that no conflicting routes can be proved simultaneously if the control functions obey the safety rules. This can be expressed in HOL as

$$\begin{aligned} & \forall N r_1 r_2 t. \\ & \text{CONFLICTING } N r_1 r_2 \supset \\ & \neg(\text{ROUTE\_PROVED } N r_1 t \wedge \\ & \text{ROUTE\_PROVED } N r_2 t) \end{aligned}$$

Since the rules and specifications are for generic networks, the properties derived should be true for all networks built according to the specification of

NETWORK. This is an analogy of the SSI. The control system is generic. To control a particular signalling scheme, the SSI requires only its geography data.

## 5 Discussions and Conclusions

The approach of this research is to develop some general theory and then apply it to the specific problem. The advantage of this approach is that the general theory, namely the graph theory, provides a sound mathematical structure as a base on which more specific arguments can be depended. Railway track networks are fairly complicated, an abstract model of them must base on some structure. They are intrinsically well suited to be represented by graphs. Adopting graphs as the structure of track networks allows us to use many well known algorithms on graph, particularly important are the algorithms for finding paths. Without a properly defined data structure, it would be very difficult to specify and modelling track networks and to derive their useful properties.

Graph theory has been applied in numerous practical problems in very diversified scientific and engineering fields. In addition to be used in railway signalling, the graph theory developed in high order logic can provide a starting point for other applications. For

example in transport industry, the problem of finding the most economical route of delivering goods and the problem of maximizing the network capacity can be solved using graphs.

The model of network presented in this paper concerns mainly the static aspect of the railway track networks. The aim of this work is to develop formal specifications for the design stage of signal schemes. Obviously, the dynamic aspect of the networks has to be considered as well. Research on this topic is being carried out, and some work has been shown in section 4.4. Work on other related areas, for instance the interlocking of level crossing has also been carried out[3]. However, a large amount of work still need to be done in the safe control of interlocking systems.

The safety record of the railway industry has been very good. This has been achieved by the rigorous regulations developed through decades of working experience. As more and more new technology is being adopted by the industry, especially the use of micro-processor in controlling vital safety functions, it is very important to ensure that the high integrity and reliability of the interlocking systems are maintained. Due to the complexity of the new technology, formal methods should be employed in the analysis, development and implementation of signalling system.

The work presented in this paper is a small step towards this aim. The author hopes that this will spark off more research effort in applying formal methods in the signalling industry.

## References

- [1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [2] Alan Cribbens. A solid state interlocking (ssi): an integrated electronic signalling system for mainline railways. In *IEE Proceedings, Part B*, volume 134, pages 148 – 158, MAY 1987.
- [3] W. J. Cullyer and W. Wong. A mathematical approach to the protection of grade crossing. In *Proceedings of international symposium on railing-highway grade crossing research and safety*, Knoxville, Tennessee, USA, 31st Oct – 3rd Nov 1990.
- [4] Michael J. Gordon. *HOL: A Proof Generating System for Higher-Order Logic*, pages 73–128. Kluwer Academic Publishers, 1987.
- [5] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, London, 1986.
- [6] I. H. Mitchell. The design and testing of application database for a railway signalling system. In *Proceedings of International Conference on Software Engineering for Real-time Systems*, Cirencester, U.K., 28 – 30 Sept. 1987, pages 159 – 164.
- [7] O. S. Nock. *Railway Signalling — a treatise on the recent practice of British Railways*. A and O Black, London, 1980.
- [8] J. M. Spivey. *The Z Notation: a Reference Manual*. Prentice-Hall, 1989.
- [9] W.L.Price. *Graphs and Networks: An Introduction*. Butterworth Co (Publishers) Ltd., London, 1971.