GROUP PROJECT 2015-16
A RAILWAY NETWORK INTERLOCKING SUPPORT TOOL

CSC8206

Group 8	
NAME	**ID**
Ali Alssaiari	109265362
Jack Chandler	120232420
Ryan Crosby	120248937
Lee Robinson	120286250
Maciej Sokolowski	120357332

MARCH 10, 2016

NEWCASTLE UNIVERSITY

Contents

1. Introduction 2
2. Requirements 4
2.1 Functional requirements 4
2.2 Non-functional requirements 5
3. System Design 6
3.1 Rules for the network 7
3.2 Interlocking setting 8
3.3 Software used during design/development 9
3.4 Overall Design 10
4. Implementation 13
4.1 Package Diagrams 14
4.2 FileReader 17
4.3 Interlocking Table Display 18
4.4 Validation 18
4.5 Sample output 19
5. Testing 22
5.1 Test plan 22
5.2 Extended Rail Network Design 24
5.3 Testing results 27
5.4 Screenshots of validation 31
6. Evaluation 33
6.1 Requirements Not Met 33
6.2 Future work / Possible extensions 33
6.3 Group Personnel Assessment 34
6.4 Resources and references 34
Appendix 35

1. Introduction

The task assigned to us was to design and create a program with the capability of reading in a railway network system from a file and constructing a representation of the network. In addition the program should calculate the correct interlocking settings for the network, showing all the conflicting routes in order to prevent an accident (be it head on collisions, following collision or derailment of trains) from occurring.

After the task was assigned to us, we organised a meeting to discuss the project requirements and to begin putting together an initial plan. We decided to split the group into two sub teams - a team for programming (Jack and Ryan) and a team for documentation/testing (Lee, Ali, Maciej). For certain parts of the project, the whole team had to work together to meet the deadlines - such as for the progress report.

For the duration of this project we used the Scrum software engineering approach which involves daily meetings to ensure the productivity of the group was being maximised. Thanks to these daily meetings the team can resolve any pressing issues that are identified and allows clear and regular updates of the project's development. One of the major advantages that convinced us to implement Scrum was not only its flexible nature but it allowed the team to begin the development of the project almost immediately.

To conform to the Scrum methodology we adopted, the two separate teams would meet frequently to discuss issues and progress related to their tasks. In addition to this, the team would meet as a whole at least twice a week to ensure that our internal deadlines were being adhered to and also so group members can share any new issues/concerns that have arisen. We used the Scrum methodology due to the short time frame of the project. It had several advantages, mainly the constant updates with progress and an easy way to point out any new developments, such as change in development styles or implementation details.

To ensure that the team could work together as effectively as possible, it was decided in the original meeting that using google drive would suit our needs. Google drive allows each member to update key documents in a shared folder from any location provided they have an internet connection. The programming team decided upon using github for the purpose of sharing code as it allows new updates to code to be pulled from or pushed to the shared repository. This repository was also used by the testing team for purpose of testing each individual component of the system as it was completed. For the purpose of communication and arranging meetings, the team decided upon using Whatsapp as it is a free instant messaging service. Whatsapp was also used for any minor questions that team members needed answered in a timely fashion.

As well as working independently in the aforementioned teams, the two teams also worked in parallel to complete deadlines due to the time sensitive nature of the project. To ensure that the system was reliable and less prone to errors arising - it was imperative that both teams worked together to find and fix any errors as well as discuss what scenarios that could

potentially lead to a train crash. This involved designing several different train networks and running the tests against all possible journeys.

2. Requirements

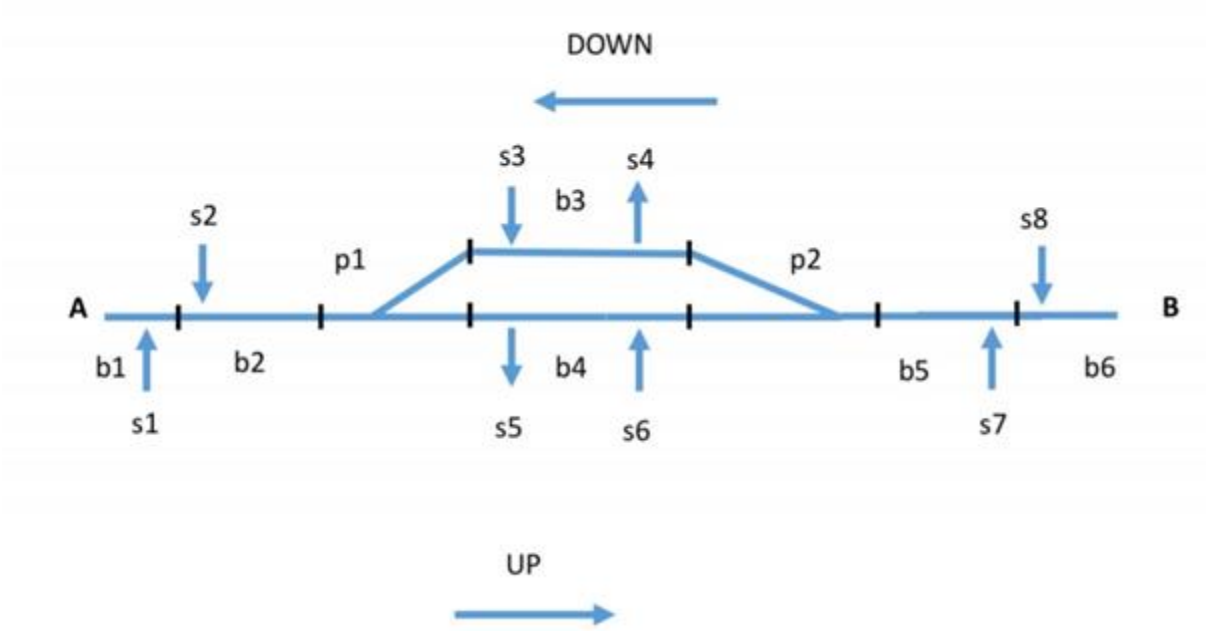
2.1 Functional requirements

No	Requirement	Priority (H, M L)	Comments	Used by
1	Read in a network specification from a file	H	This allows user to provide network data to program from a text file format (.txt)	System
2	Produce a graph from the file entered by the user	L	Convert the network data from a .graphml into a graph using JGraphT libraries	System
3	Produce visual representation of the current state of the rail network	M	Program provides a visual representation of the network through the use of terminal	System
4	Calculate the settings for interlocking	H	Program needs to calculate all of the data necessary to produce the interlocking table	System
5	Display blocks within the network that are occupied	L	Shows the current blocks that are being occupied by the train and the points which will be unsafe to enter	System
6	Display a table showing the interlocking settings	H	Program is able to display previously calculated settings in form of table on the screen	System
7	Export a table as a file	L	The program will allow the user to download and save the table as a file if they desire	System
8	Allow the user to define journey from one part of the network to another from the command line	H	The user can define any specific journey and program will produce the safe interlock setting for the duration of journey	User
9	Perform checks against head-on collisions	H	The prevention of two trains moving in opposite directions on the same track from colliding	System
10	Perform checks against following collisions	H	The prevention of a train running into the back of another train travelling in the same direction on the track	System
11	Perform checks against derailment collisions	H	The prevention of a point being set against a train travelling along a track	System
12	Data should always be displayed in sensible order	M	This should allow the user to quickly find the information that they require	System
13	Data entered by user should be restored in the event of application crash	L	This requirement will prevent users of the application from getting frustrated if their data is lost	System

2.2 Non-functional requirements

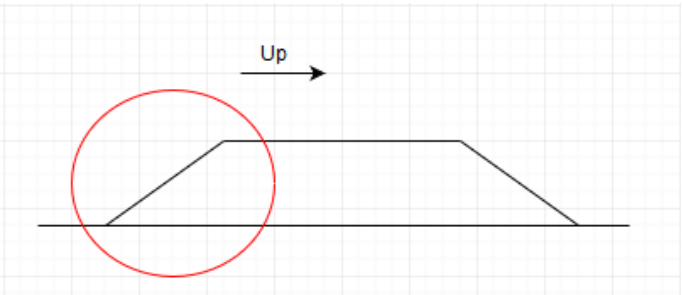
No	Requirement	Priority (H, M L)	Comments	Used By
1	Ensure the security of application from adversaries	H	To prevent hazards and accidents from occurring (potentially leading to loss of life and damage), adequate security measures should be implemented	System
2	Clear and visually appealing GUI/HCI design	L	Reduce user errors and increase the usability of the application. Encourages users/operators to choose this application over other competitors	System
3	Software documentation/guide	L	Documentation assists user how to use the applications correctly and effectively	User
4	If the specification requirements are met, the application should run smoothly and efficiently	M	Users would be more likely to use an application that provides information quickly	System
Testing requirements:				
5	Independent from programmers	H	Tests should be designed independently from programmers in order to discover any errors that the programming team overlooked	Testers
6	Prioritise the safety aspects of the system to be tested	H	Tests should focus on safety-critical aspects of the system	Testers
7	Exhaustive test coverage	H	Every aspect of the system should be tested thoroughly	Testers

3. System Design



To ensure the validity of the network certain rules and constraints have been designed for each aspect of the system. If the network itself is invalid the application will not allow any routes to be defined.

Note:
We have also come up with some definitions for specific parts of the network.



When a point has two neighbours in the up direction and one in the down, we have called this a point that **faces up** as shown above. The other point is not facing up.
Also we have defined a pair of points to be like the set of points above, if two points are set up like this then they are considered a pair.

3.1 Rules for the network

Blocks:

- The maximum number of neighbours for a block is two and each neighbour can either be a Block or a Point.
- The block in the (PLUS) track cannot have a block from the (MINUS) track as a neighbour and vice versa. E.g. b3 cannot be a neighbour for b2.
- A block can have minimum of one neighbour and this is used for the special cases of the beginning and end of the railway track - either no neighbour to the left or no neighbour to the right. E.g. b1 and b6 in the figure above - each one of them only has one neighbour.
- Each block will have a maximum 1 (UP) signal and 1 (DOWN) signal, if the block is the first block or the last block in the track it can only has a one signal.

Points:

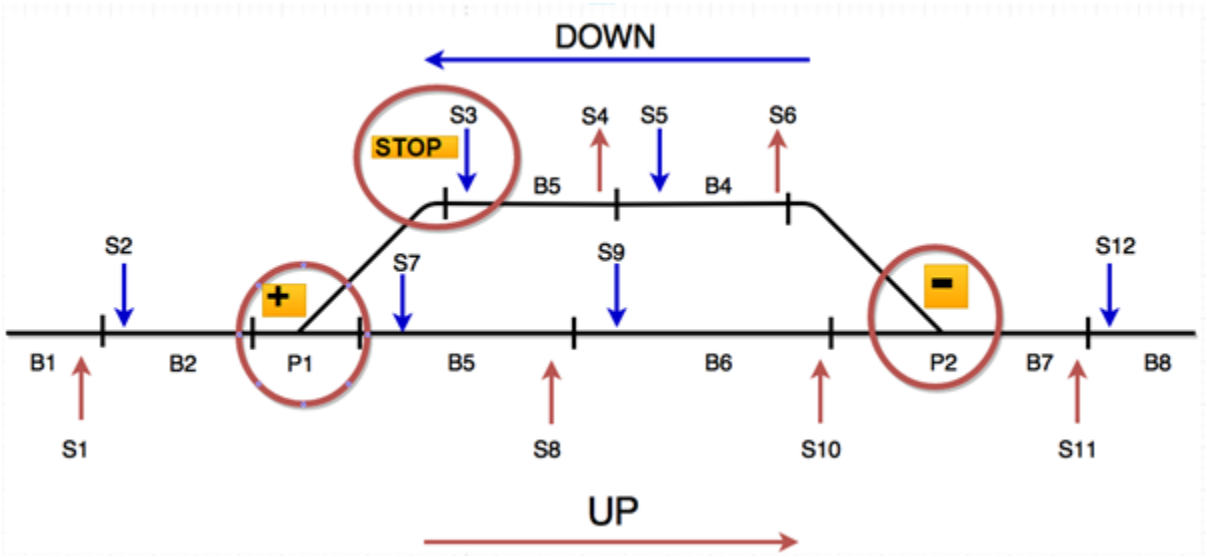
- A point will always have 3 neighbours which can only be blocks. A point cannot have another point as a neighbour.
- A point cannot be at the start or the end of the network.

In addition to the previous constraints that are related to forming a valid network structure, there are rules to form valid routes, and also the **safety** rules for interlocking:

Routes

- A route can only pass through a maximum of one point, it cannot pass through two points.
- The source signal and the destination signal for the route cannot be equal. E.g. (S1 to S) is an invalid route.
- The route cannot go from an (UP) signal to a (DOWN) signal, which means the source signal and destination signal must be in the same track direction (PLUS) or (MINUS). E.g. s1 to s3 is an invalid route and should not be accepted as s1 is an (UP) signal and s3 is a (DOWN) signal.
- The source and destination signals must go in the direction of the route. E.g. a route defined as s1 to s6 is valid and accepted as both signals are (UP) and the direction is PLUS and the defined route is in the (UP) direction. But if the route defined as (s6 to s1) this is invalid route as the direction is not going in (PLUS) direction even the 2 signals are (UP) signals.

3.2 Interlocking setting



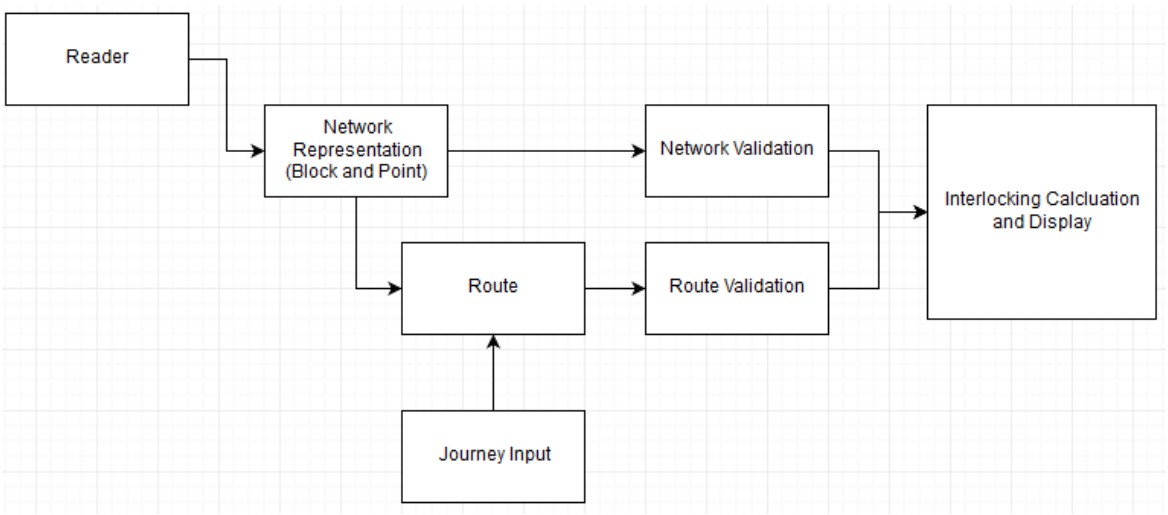
- For the point setting(s) - if the route is from a signal outside a pair of points to inside a pair of points, then the point is set to the correct setting depending on the destination block's track, in addition to this, the other point in the pair will be set to the opposite setting. This is to prevent a head on collision. E.g. the route (S1 to S8) then (P1) set to (PLUS) and (P2) set to (MINUS). If the route comes from the inside of a pair of points to the outside, then only one point needs to be set. E.g. the route (S10 to S11) the point (P2) will set to (PLUS).
- For signals - if a route has a point in it, a signal is needed to (STOP) a train from the other track because otherwise it will derail when passing through the point (preventing derailment) e.g. the route (S1 to S8) the signal (S3) set to (STOP). Furthermore, the rest of the signals on the route that are in the opposite direction of the route, are set to (STOP), this is to once again prevent a head on collision. E.g. the route (S1 to S8) the signal (S2, S7) set to (STOP).
- A conflict occurs when two routes have the same block in their path. When these conflicts are listed, they will prevent following collision from occurring.

3.3 Software used during design/development

In the initial phase of planning our program, group responsible for designing program and implementing all the solutions agreed on using following:

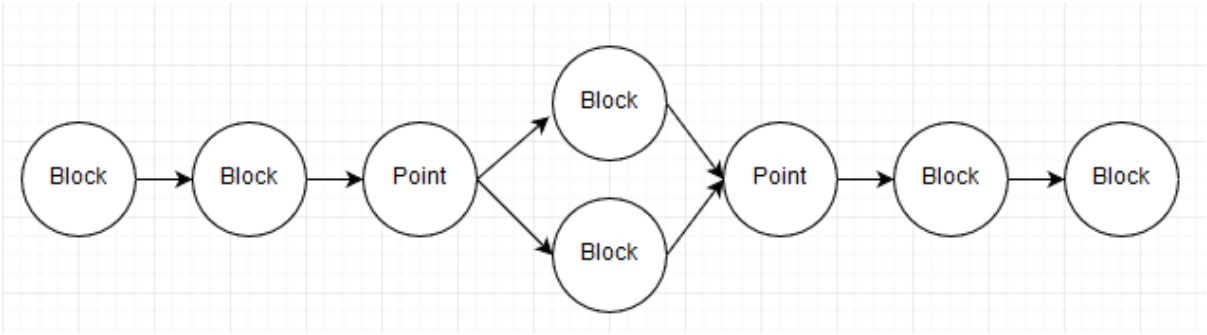
- JGraphT - it is a free, open-source Java graph library which supports lots of different types of graphs (directed, undirected, weighted, unmodifiable and many others). Even though JGraphT is a really powerful tool it is designed in a simple and type-safe way, by the use of Java generics. Graphs can be created with the use of XML documents, URLs, etc. It is focused mainly on data structures and algorithms.
- Java - one of the most well-known object-oriented programming languages, with huge community support. Compiled Java code can run on any platform that supports Java, without the extra need for recompiling. We have chosen Java as everyone has history in using this language at some point in their academic life, therefore everyone has experience using it – which makes coding, testing and additional help from other team members much easier.
- Eclipse - We decided upon using Eclipse as our workspace as it is an open source IDE that allows users to easily import new libraries or plugins if required. A key feature which let us to choosing Eclipse was the ease of which the team can push or pull new versions of the code to github.
- Maven - build automation tool used mostly for Java projects. Each particular Maven features are provided by plugins, which are downloaded when they are used for the first time. The file specifying how the project is built, is called POM-file.
- GraphML - easy to use file format which consists of a language core and extension mechanism - former is used to describe properties of graphs and the latter to add application specific data. Instead of using custom syntax, it is based on XML, which makes it a great tool to use with all types of different services.
- JGraphX - Java Swing diagramming library. In comparison to JGraphT which focuses on data structures and algorithms, JGraphX is focused on the GUI, it provides visualisation and interaction for node-edge graphs, supports different imports and exports and provides automatic positioning of edges and nodes.
- OVal - OVal is an open source framework that can be used for object validation in Java. Oval allows us to use Java annotations to declare the constraints on an object.
- J-text-utils - Is an open source Java library that allows ASCII table to be constructed

3.4 Overall Design

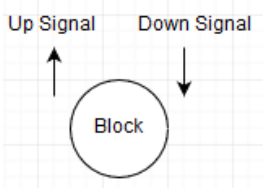


This diagram shows the initial overall design of our system. It shows that before the interlocking calculations are performed, the different parts of the network which have to be validated in their respective component and the routes are validated in another. Only when the network and the routes in all the input journeys are validated will the interlocking calculations take place.

Network Representation



We decided to use a graph to represent our network, the nodes can be either blocks or points, and we can use validation on the network to check our rules for the system as stated earlier.



Blocks also have the option of having one up signal, and one down signal. Then the network can be created via an input file, we can specify the signals which each block needs and later validate this network. One of the advantages of using a graph is that once connected via edges, each section will know about its direct neighbours and this way paths / sub-networks of the total network are able to be defined.

Routes

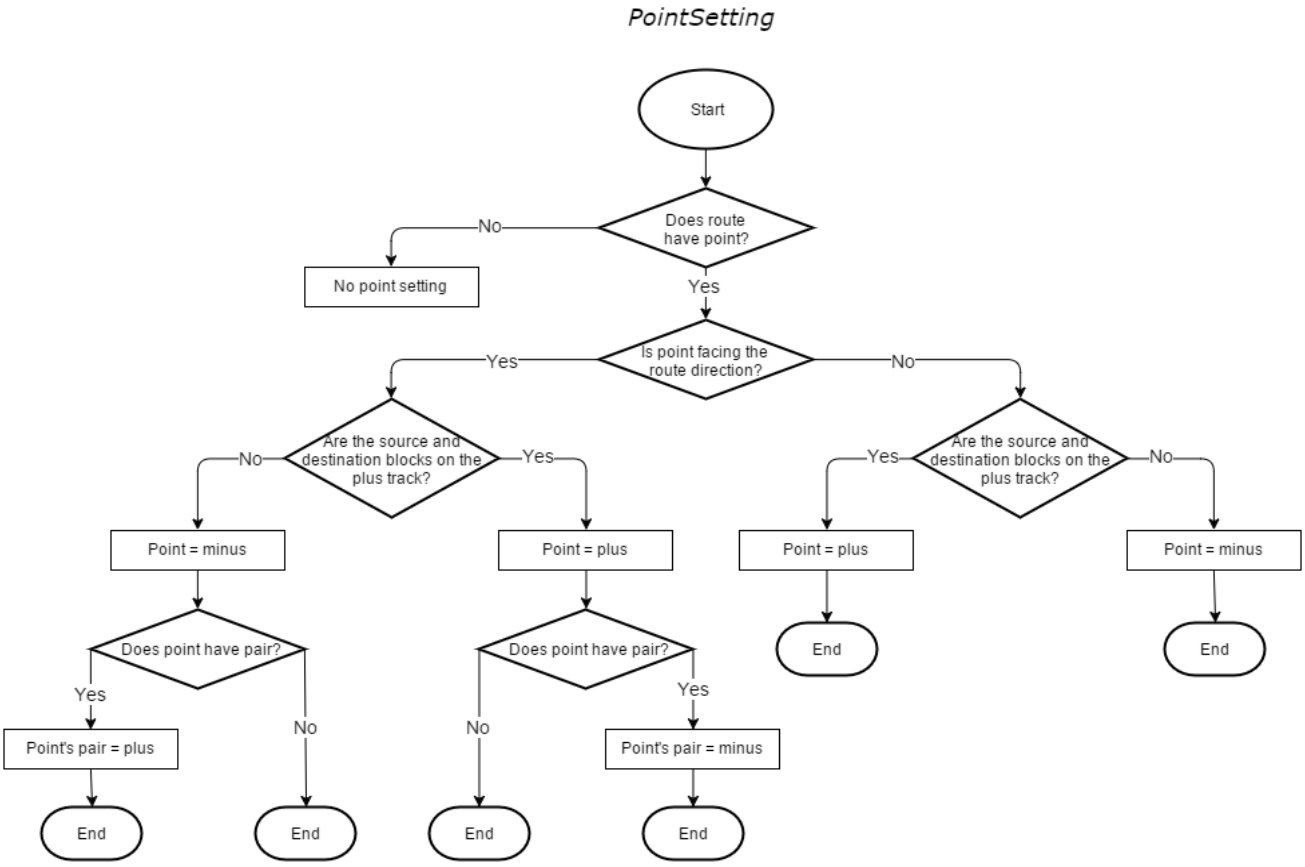
Routes are defined as signal to signal and we have said that a route will know all blocks that will be in a route. This is called the path and is needed since the interlocking settings depend on knowing all the sections in a route so they can be calculated. To form the path we can traverse our network graph and add sections depending on the source and destination signals of the route.

Journey Input

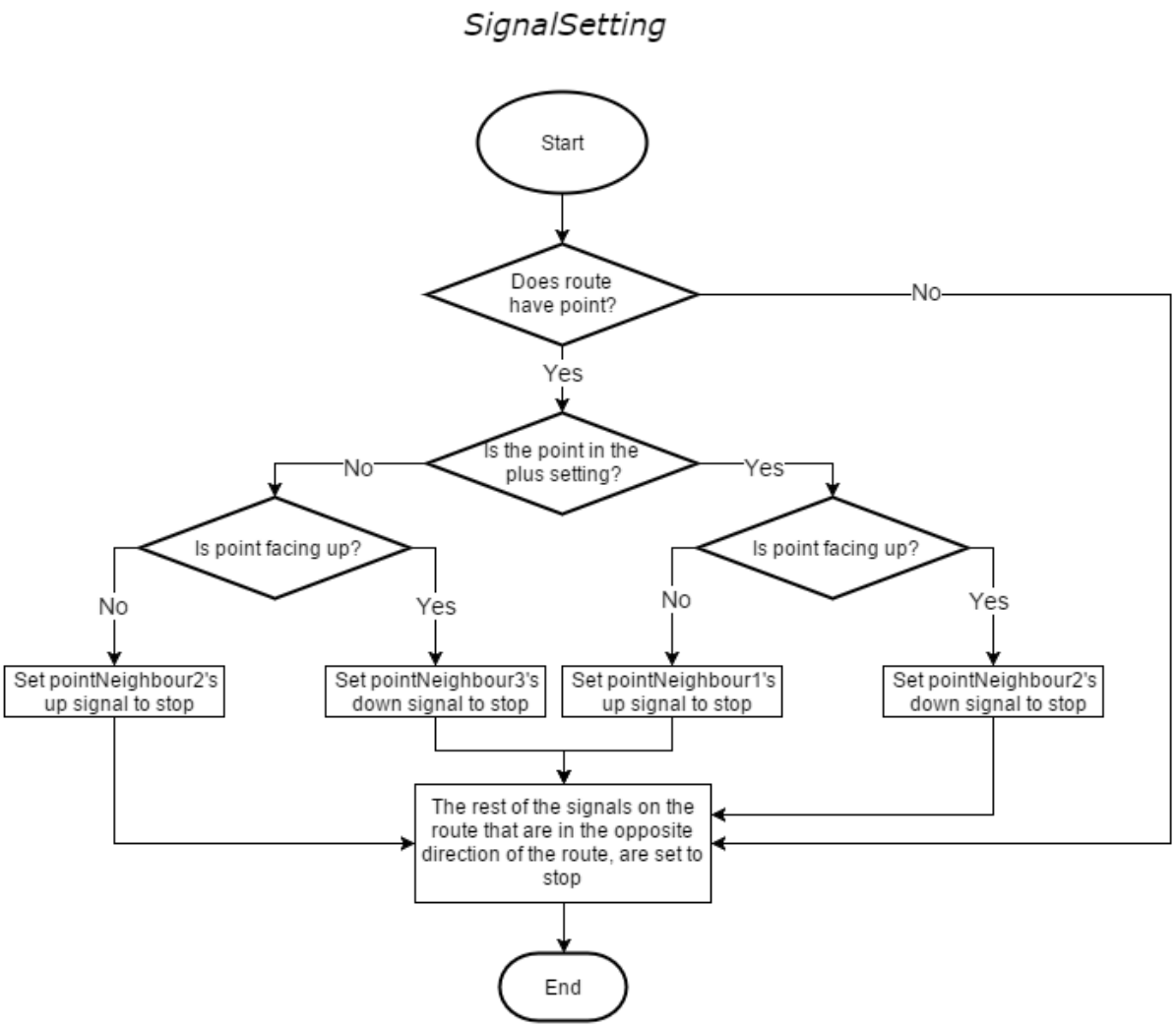
We have decided that using a simple command line interface for a user to enter journeys will make the program simpler to implement, as well as also being simpler for the user.

Interlocking Settings

Since there are three interlocking settings, here, we will show the two different designs for working out the settings for points and signals.



This diagram shows our logic to decide what settings the points will be depending on the route given.



This diagram shows our logic to decide which signals should be set to STOP.

4. Implementation

Our program was built in a modular way, mainly because since we used the scrum methodology we would build different parts of the system, at separate times, and then make them compatible together. Building it this way also means if any extensions were going to be built, they could easily be implemented without having to change the other parts of the system.

The basis for the NetworkRepresentation package are the three classes, Section, Block and Point. In our implementation a Section is the general part of our network graph, and Block / Point inherit the Section class so that any Section can have a neighbour of both Blocks and Points. Blocks have two Signal fields to represent the possible up and down signals. The class CreateNetwork is then mainly used for the static method *addEdges* to add the Blocks and Points to the list of Sections *neighList* for every section in the network. Now we can build a virtual model of a train network.

For the Reader class a method was added to Block and Point called getInstance. This is to allow Reader the ability to create a list of Blocks and Points depending on the input file taken in. A method called readFile, takes a text file defined earlier line by line and called the appropriate getInstance (object factory) methods. If the first line read in is a block, the system would then call the Block object factory, if the system reads in a Point the program would then call a Point object factory. This is also where Blocks/Points are validated with our ValidateNetwork class, so as to keep our Network well formed. ValidateNetwork throws InvalidNetworkException if anything is invalid.

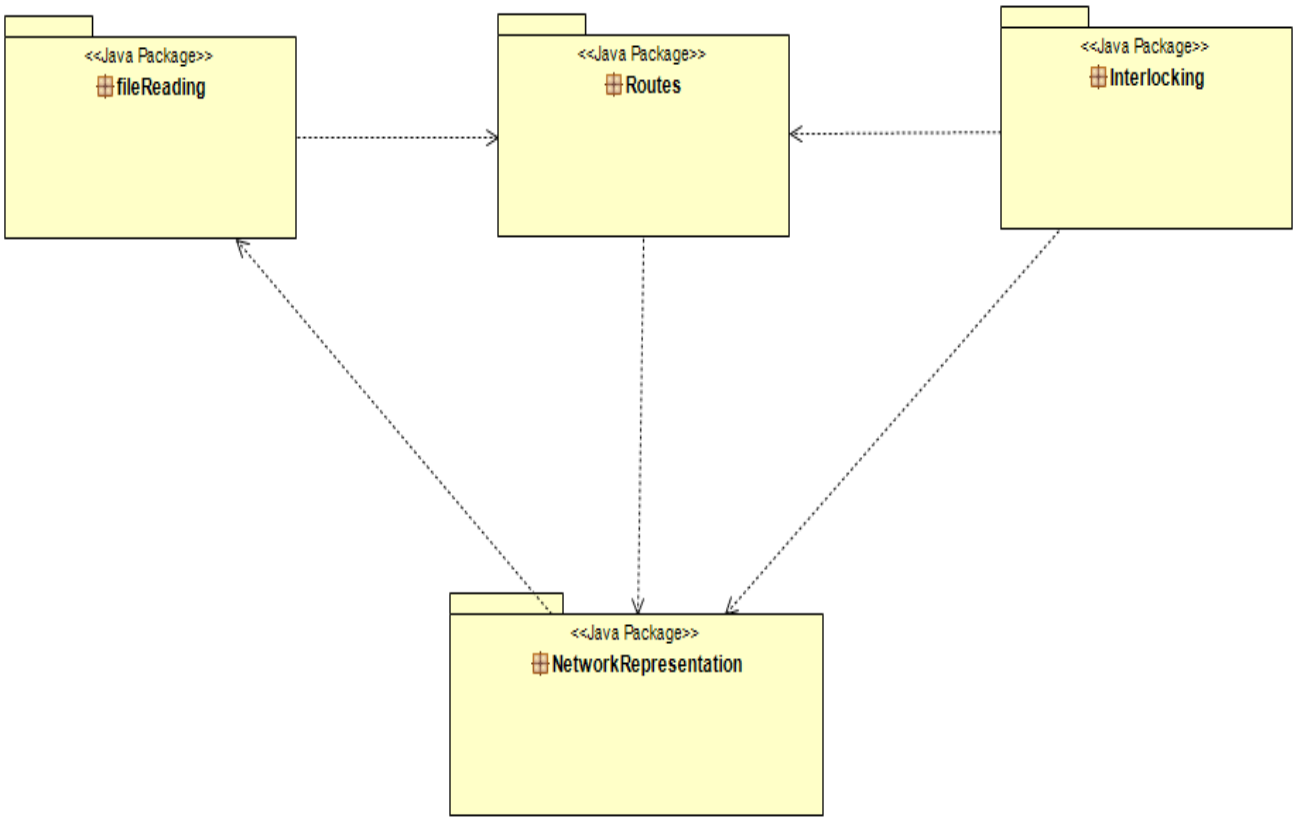
The Route package contains the class Route used to define a route in the system, as well as the InputJourney class which takes in a simple command line input on how many Journeys you want. Because the sections in our network have access to their neighbours, this allows us to easily populate the path of the route by traversing through the network and adding it to the list of Sections called route. This is done every time an instance of route is created.

The idea for InputJourney was to ask a user for a number of journeys, and how many routes this journey was made up from, this was done with scanners. For each journey the system will then ask how many routes are in said journey. The user then needs to add the signals that the route will take place from. Once the signals were entered, a Route object factory would be called and a method called validateRoute would be called on every new route, this would throw InvalidRouteException if any route was invalid.

Finally, the InterlockingTableGenerator class generated the interlocking settings to apply our safety rules conceived in the system design, as well as produce a table to show this to the user. The algorithms shown earlier by flow diagrams for signal, point and conflict settings, were implemented into the system by using the routes created by the user. Each route given in the list of journeys was iterated through, and pointSettings, signalSettings, and conflictSettings were applied on each route. This was done by accessing the path of the route, made from a sub-network, and then just changing boolean values for the signals and points, for their respective algorithms.

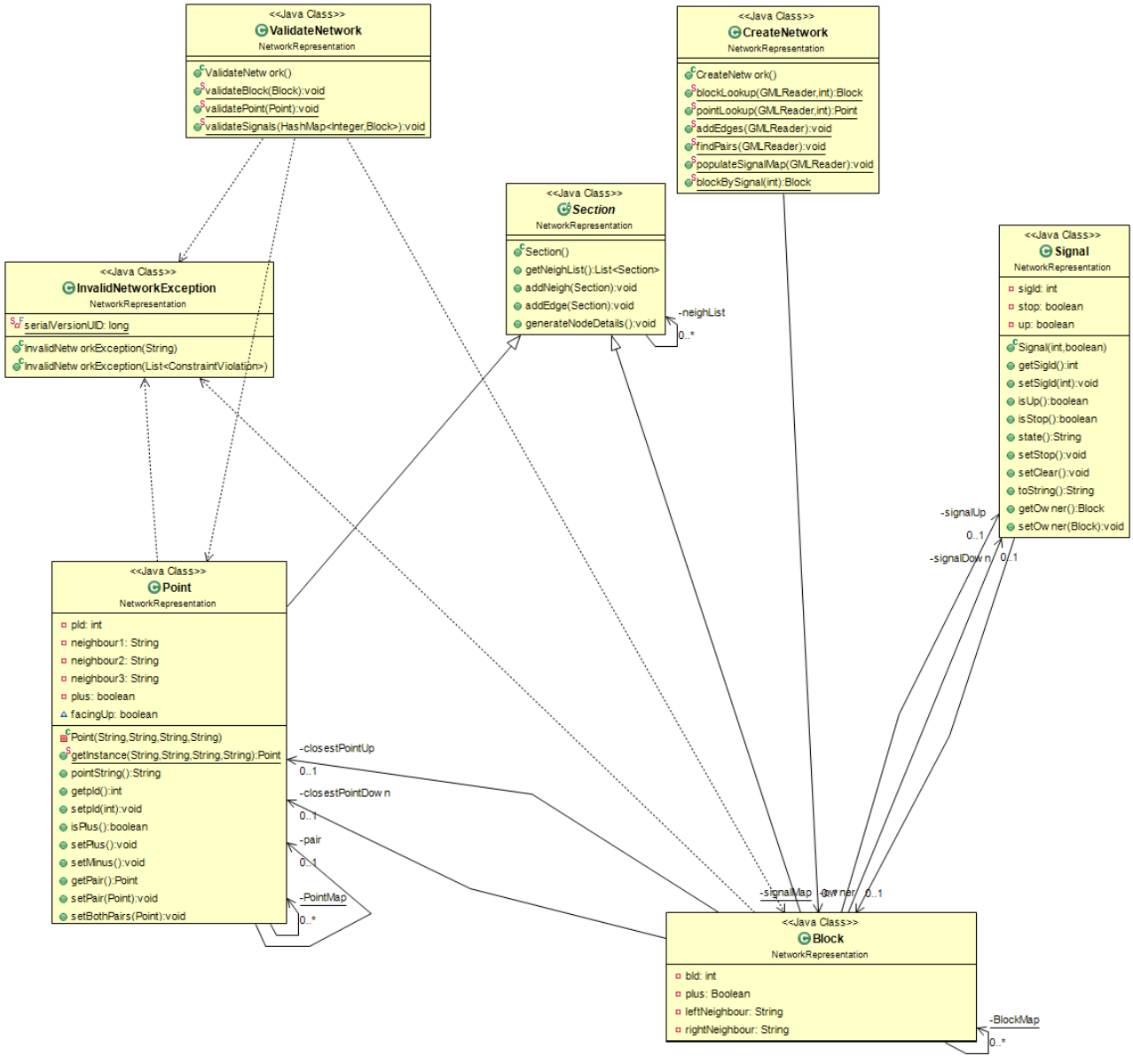
4.1 Package Diagrams

Abstract representation:

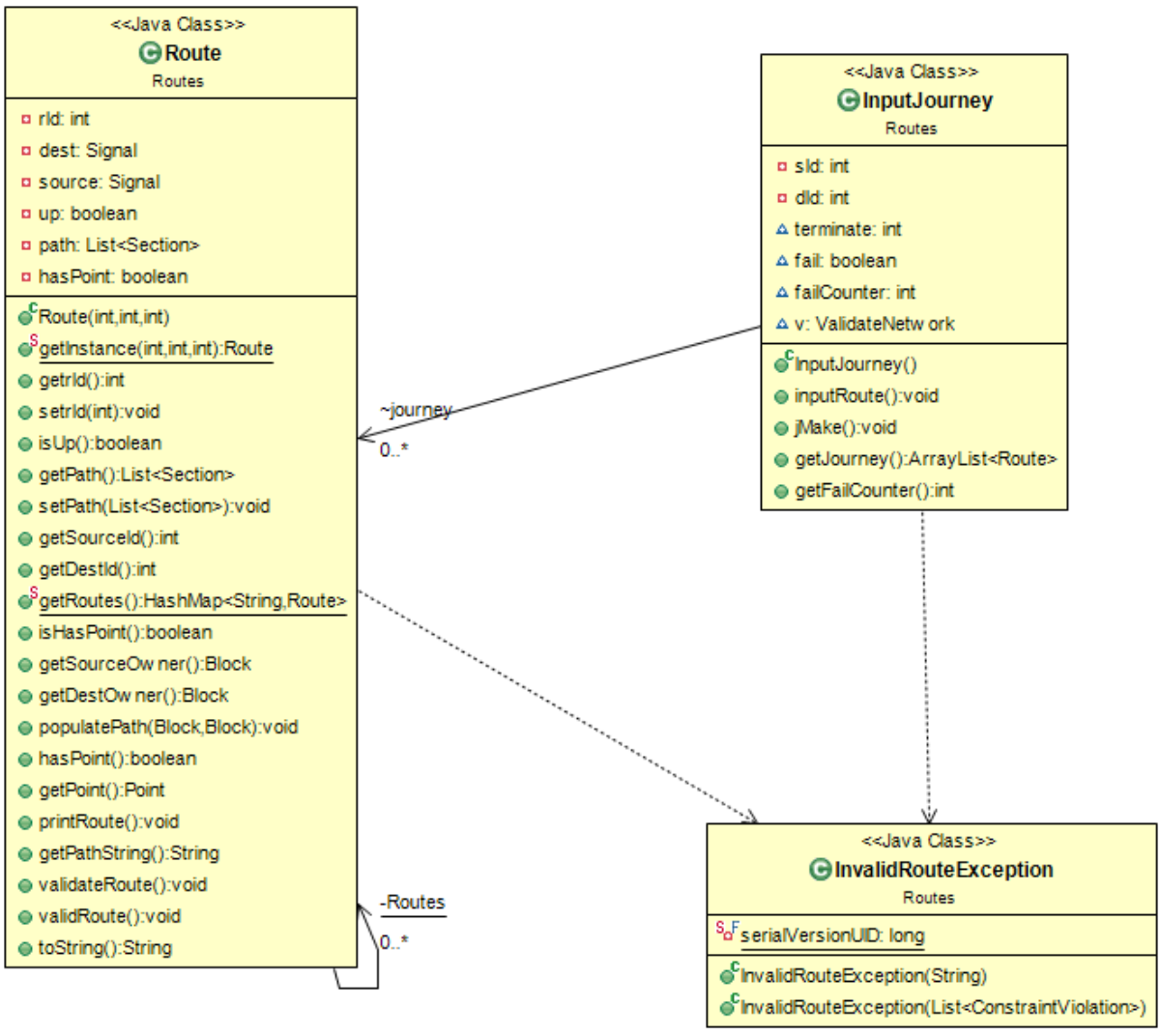


This package diagram is the implementation of our general design.

NetworkRepresentation Package:



Route Package:

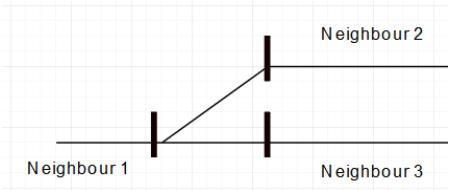


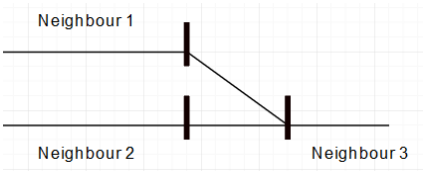
4.2 FileReader

After completing research into different ways to use GML in the program it was decided that instead we would define our own, there were two main reasons for this; the main reason was the limited development time, rather that waste time learning how to use a library we thought it would be more efficient to create and use our own code. Another reason is if the user does have to use the actual file for any reason, a simple text file with an easy to read structure, would make understanding it a lot easier than a more complex language like GML. When reading in the file for the network the blocks and points associated with the network should be in the following structure:

Block	Point
Block (type) b1 (blockID) NA (Left neighbour) b2 (Right neighbour) plus (Rail) s1 (Signal Down) NA (Signal Up)	Point (type) p1 (pointID) b2 (Neighbour 1)* b3 (Neighbour 2)* b5 (Neighbour 3)*

*A Point's neighbours are numbered differently depending on if the point is facing up or down, as shown below. We feel like this is the order that someone would naturally name these neighbours in a diagram.





4.3 Interlocking Table Display

```
_____________________________________________________________________________
| ID| Source| Destination| Points      | Signals   | Path      | Conflict     |
|===========================================================================|
| r1| s1    | s4         | p1:m  p2:p  |  s5 s2 s3 | b2, p1, b3|  r2 r5 r7 r8 |
```

We used a java library called j-text-utils which made it possible to print a formatted table to the user. The table is made from a 2D String array, so each row is just a string array of object ids that are added to the array every time each interlocking algorithm is ran (signalSettings, pointSetting etc.).

4.4 Validation

The validation of a well formed networ, was done mainly using OVal. This allowed us to apply the constraints of our rules in the system design to the Block and Point class. When the File Reader created the different instances of Blocks and Points, a class called ValidateNetwork would use one of OVal's classes, Validator, to check all the constraints of the newly created pieces of the network, and throw a new InvalidNetworkException, if any of the rules were broken.

```
@MatchPattern(pattern = { "[bBpP]\\d|^NA" }, message = "Please input a valid section ie. b1, p3 OR input NA")
private String leftNeighbour;
@MatchPattern(pattern = { "[bBpP]\\d|^NA" }, message = "Please input a valid section ie. b1, p3 OR input NA")
private String rightNeighbour;
```

More validation was used to check that when a user input a journey, the routes created from this journey were valid ones, once again using the rules formulated in the system design. This time InvalidRouteExceptions are thrown if a rule is broken. In either case, every time an exception is thrown, it will output a custom message specifying which rule has been broken, as well as the Block/Point/Route id so that it can be identified.

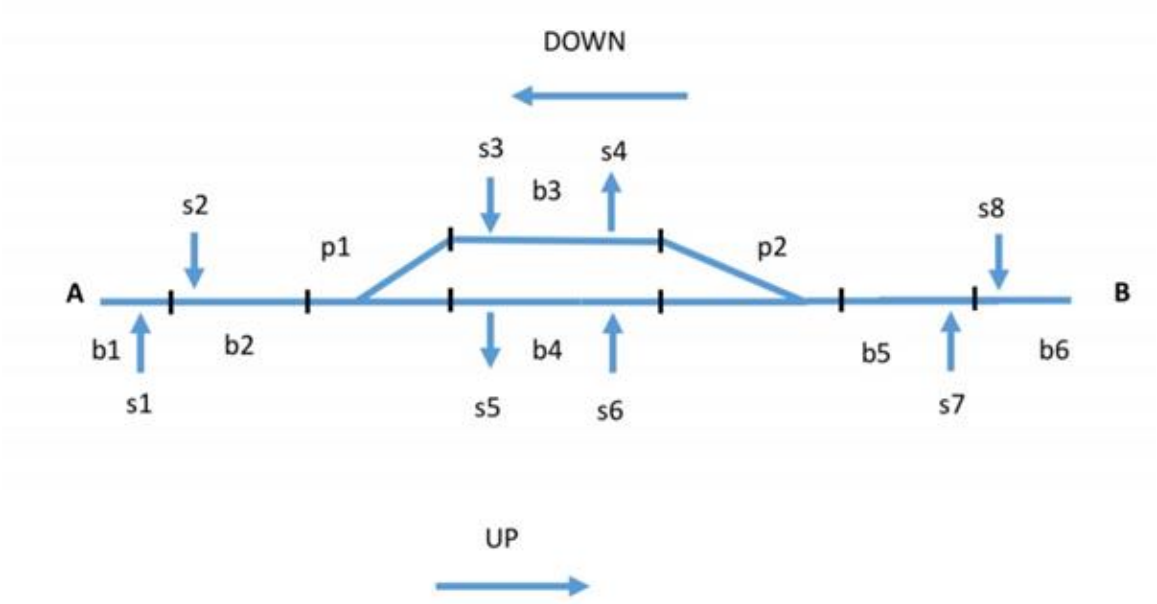
```
// If both parts of the network are Blocks (and
// neighbours) and they aren't on the same track (plus
// and minus) then throw an error
if (temp.getNeighList().get(1) instanceof Block)
{
    if (((Block) temp).isPlus() == ((Block) temp.getNeighList().get(1)).isPlus())
    {
        path.add(temp.getNeighList().get(1));
        temp = temp.getNeighList().get(1);
        blockCounter++;
    }

    else
        throw new InvalidRouteException(
                "Cannot create a route with two consecutive blocks where each block is on a different part of the track.");

}
```

4.5 Sample output

To demonstrate the output from the system, we will take the following network into consideration and assume that all data entered is correct with respect to the rules.



The file is firstly read and all sections are displayed along with their associated signals, neighbours and which track they belong to:

```
BLOCK:1
 leftNeighbour: NA
 rightNeightbour: b2
 Plus: true
 signalUp: s1- clear
 signalDown: NA- NA

BLOCK:2
 leftNeighbour: b1
 rightNeightbour: p1
 Plus: true
 signalUp: NA- NA
 signalDown: s2- clear

BLOCK:3
 leftNeighbour: p1
 rightNeightbour: p2
 Plus: false
 signalUp: s4- clear
 signalDown: s3- clear

BLOCK:4
 leftNeighbour: p1
 rightNeightbour: p2
 Plus: true
 signalUp: s6- clear
 signalDown: s5- clear

BLOCK:5
 leftNeighbour: p2
 rightNeightbour: b6
 Plus: true
 signalUp: s7- clear
 signalDown: NA- NA

BLOCK:6
 leftNeighbour: b5
 rightNeightbour: NA
 Plus: true
 signalUp: NA- NA
 signalDown: s8- clear

POINT:1
 neighbour1: b2
 neighbour2: b3
 neighbour3: b4

POINT:2
 neighbour1: b3
 neighbour2: b4
 neighbour3: b5
```

The output below is for two journeys from A to B via the main/plus track and B to A via the adjacent/minus to track. These journeys are defined from the command line interface like so:

```
How many journeys do you wish to add?
2
You are on journey 1
How many routes do you wish to add?
2
Please Enter a beginning destination ------ Such as S1
s1
Please Enter a final destination ------ Such as S6
s6
Please Enter a beginning destination ------ Such as S1
s6
Please Enter a final destination ------ Such as S6
s7
You are on journey 2
How many routes do you wish to add?
2
Please Enter a beginning destination ------ Such as S1
s8
Please Enter a final destination ------ Such as S6
s3
Please Enter a beginning destination ------ Such as S1
s3
Please Enter a final destination ------ Such as S6
s2
```

Finally the interlocking settings for these routes are calculated and displayed in the following format:

```
_____________________________________________________________________________
| ID| Source| Destination| Points      | Signals    | Path       | Conflict|
|===========================================================================|
| r1| s1    | s6         | p1:p  p2:m  |  s3 s2 s5  | b2, p1, b4|  r4     |
| r2| s6    | s7         | p2:p        |  s4 s8     | p2, b5     |  r3     |
| r3| s8    | s3         | p2:m  p1:p  |  s6 s7 s4  | b5, p2, b3|  r2     |
| r4| s3    | s2         | p1:m        |  s5 s1     | p1, b2     |  r1     |
```

The signals for the blocks and the setting for the points (plus/minus) are then updated and reserved so the latest journey can successfully pass through the railway track without any collisions or derailments from occurring.

```
BLOCK:1
 leftNeighbour: NA
 rightNeightbour: b2
 Plus: true
 signalUp: s1- stop
 signalDown: NA- NA

BLOCK:2
 leftNeighbour: b1
 rightNeightbour: p1
 Plus: true
 signalUp: NA- NA
 signalDown: s2- clear

BLOCK:3
 leftNeighbour: p1
 rightNeightbour: p2
 Plus: false
 signalUp: s4- stop
 signalDown: s3- clear

BLOCK:4
 leftNeighbour: p1
 rightNeightbour: p2
 Plus: true
 signalUp: s6- stop
 signalDown: s5- stop

BLOCK:5
 leftNeighbour: p2
 rightNeightbour: b6
 Plus: true
 signalUp: s7- stop
 signalDown: NA- NA

BLOCK:6
 leftNeighbour: b5
 rightNeightbour: NA
 Plus: true
 signalUp: NA- NA
 signalDown: s8- stop

POINT:1
 neighbour1: b2
 neighbour2: b3
 neighbour3: b4
 Setting: minus

POINT:2
 neighbour1: b3
 neighbour2: b4
 neighbour3: b5
 Setting: minus
```

5. Testing

5.1 Test plan

As our main concern when designing this application was safety and reliability, it is imperative that the application is tested as thoroughly as possible. To accomplish this we decided to run multiple different tests to catch and correct the majority of errors/problems. The testing phase ran in parallel to the development/implementation with each fundamental component of the system being tested as soon as it was completed (unit testing). In addition to this, the whole system has been tested as a whole to ensure that no additional bugs were introduced (system testing).

To ensure the maximum amount of test coverage, we decided that the testing team would design all of the tests independently from the programmers. This allowed the testing team to consider additional test cases that perhaps the programming team may have overlooked.

Once the testing of the system has been fully completed, we prioritised the correction of errors based upon their severity in terms of each individual errors potential safety risk to the train/passengers. After the more serious errors have been fixed, the testing team will focus on less safety critical errors that have been uncovered - such as user interface faults, etc.

Our approach for the test plan is to use black box testing, which we are comfortable with using (from previous projects). This involves inputting data (both expected and erroneous) and comparing it with our expected output without looking at the code within the application. Black box testing is testing the code without knowing any aspect of implementation. This was a suitable form of testing in the case of this project because end users will face a similar use of the situation.

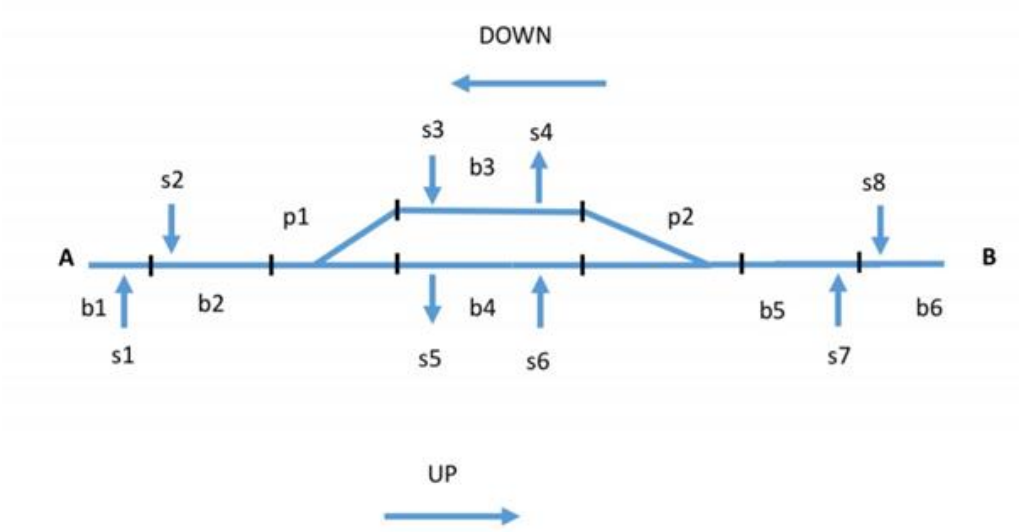
We decided the best approach for testing our application was to run it against numerous networks to try and discover any new errors that may not have arisen in the simple network that was provided. For ease of reading, the network shown in the design section has been used in the test plan unless stated otherwise.

Unfortunately due to the time frame that was allocated to the project, some of the lower priority functional requirements did not make it into the final version of the program. These functional requirements were not critical to making the system safe and error-free and were mainly based around the graphical representation of the network. For this reason, they have not been included in the test plan.

Testing Scenarios:

The original railway design in the specification document

The test first should be applied on the original railway network that was defined in the specification.



ID	Source	Destination	Points	Signals	Path	Conflict
r1	s1	s4	p1:m p2:p	s5 s2 s3	b2, p1, b3	r2 r5 r7 r8
r2	s1	s6	p1:p p2:m	s3 s2 s5	b2, p1, b4	r1 r6 r7 r8
r3	s4	s7	p2:m	s6 s8	p2, b5	r4 r5 r6
r4	s6	s7	p2:p	s4 s8	p2, b5	r3 r5 r6
r5	s8	s3	p2:m p1:p	s6 s7 s4	b5, p2, b3	r1 r3 r4 r6
r6	s8	s5	p2:p p1:m	s4 s7 s6	b5, p2, b4	r2 r3 r4 r5
r7	s5	s2	p1:p	s3 s1	p1, b2	r1 r2 r8
r8	s3	s2	p1:m	s5 s1	p1, b2	r1 r2 r7

5.2 Extended Rail Network Design

Before the programming team began their implementation it was decided in addition to the sample railway network in the specification, we would consider additional railway networks, such as in figure [1]. This extended network has 2 more blocks along with the appropriate signals for each one. The main reason for designing these additional networks was to discover any potential errors that may not have arisen from the simple sample network that was provided.

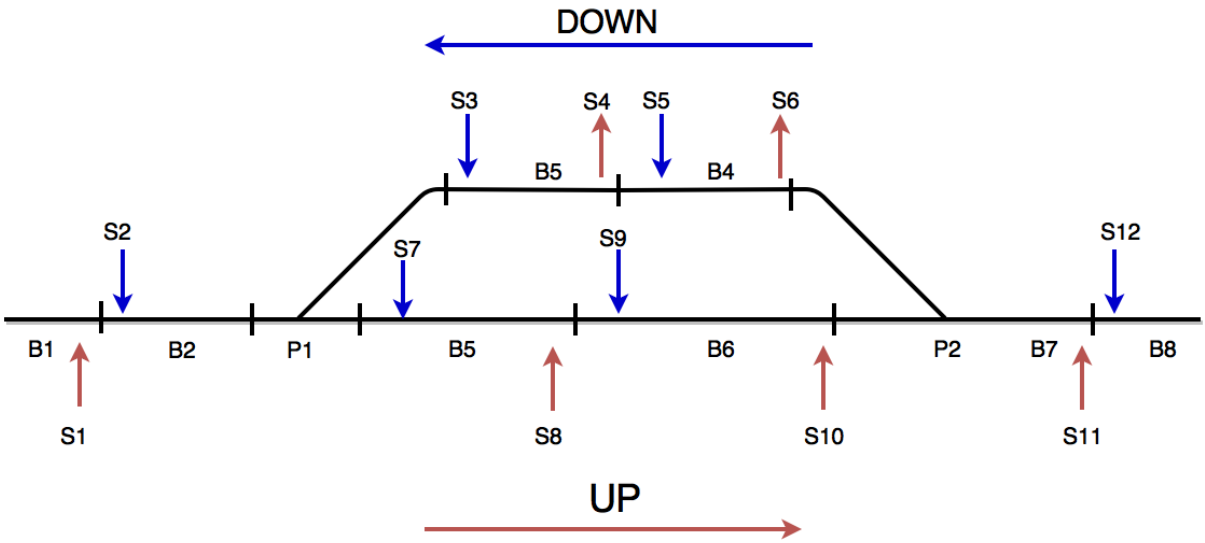


Fig. 1 Extended network model

We calculated the interlocking table for this network that will be used to compare against the interlocking table from the program to ensure that the settings for points, the conflicts/paths for each route and the signals for each route are all correct. This interlocking settings table will allow us to easily find any invalid outputs from the program which can be reported back to the programming team to be corrected.

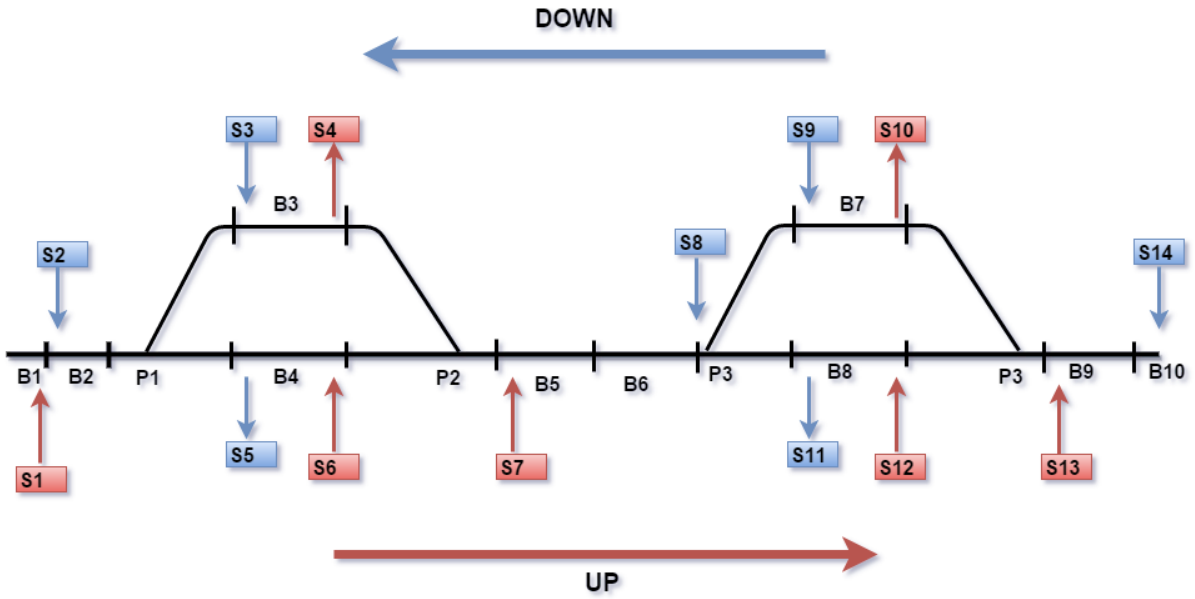
ID	Source	Dest	Points	Signals	Path	Conflicts
r1	s1	s8	p1:p; p2:m	s2; s3; s7	b2; b5; p1	r2; r5; r6; r10; r11; r12; r15; r16; r19; r20
r2	s1	s10	p1:p; p2:m	s2; s3; s7; s9	b2; b5; b6; p1;	r1; r3; r5; r6; r9; r10; r11; r12; r15; r16; r19; r20
r3	s8	s11	p2:p	s6; s9; s12	b6; b7 p2	r2; r4; r7; r8; r9; r10; r13; r14; r19; r20
r4	s10	s11	p2:p	s6; s12	b7; p2	r3;r7;r8;r9;r10;r13; r14

r5	s1	s4	p1:m; p2:p	s2; s3; s7	b2; b3; p1	r1; r2; r6; r11; r12; r14; r15; r16; r17; r18
r6	s1	s6	p1:m; p2:p	s2; s3; s5; s7	b2; b3; b4; p1	r1; r2; r5; r7; r11; r12; r13; r14; r15; r16; r17; r18
r7	s4	s11	p2:m	s5; s10; s12	b4; b7; p2	r3; r4; r6; r8; r9; r10; r13 r14; r17; r18;
r8	s6	s11	p2:m	s10; s12	b7; p2	r3; r4; r7; r9; r10; r13; r14;
r9	s12	s9	p2:m; p1:p	s6; s10; s11	b6; b7; p2	r2; r3; r4; r7; r8; r10; r13; r14; r19; r20
r10	s12	s7	p2:m; p1:p	s6; s8; s10; s11	b5; b6; b7; p2	r1; r2; r3; r4; r7; r8; r9; r11; r13; r14; r19; r20
r11	s9	s2	p1:p	s1; s3; s8	b2; b5; p1	r1; r2; r5; r6; r10; r12; r15; r16; r19; r20
r12	s7	s2	p1:p	s1; s3	b2; p1	r1; r2; r5; r6; r11; r15; r16
r13	s12	s5	p2:m; p1:p	s6; s10; s11	b4; b7; p2	r3; r4; r6; r7;r8;r9;r10; r13; r14; r17; r18
r14	s12	s3	p2:m; p1:p	s4; s6; s10; s11;	b3; b4; b7; p2	r3; r4; r5; r6; r7; r8; r9; r10; r13; r15; r17; r18;
r15	s5	s2	p1:m	s1; s4; s7	b2; b3; p1	r1; r2; r5; r6; r11; r12; r14; r16; r17; r18
r16	s3	s2	p1:m	s1; s7	b2; p1	r1; r2; r5; r6; r11; r12; r15;
r17	s4	s6		s5	b3, b4	r5; r6; r7; r13; r14; r15; r18
r18	s5	s3		s4	b3, b4	r5; r6; r7; r13; r14; r15; r17
r19	s8	s10		s9	b5; b6	r1; r2; r3; r9; r10; r11; r20
r20	s9	s7		s8	b5; b6	r1; r2; r3; r9; r10; r11; r19

From the program:

ID	Source	Destination	Points	Signals	Path	Conflict
r1	s1	s8	p1:p p2:m	s3 s2 s7	b2, p1, b5	r2 r5 r6 r10 r11 r12 r15 r16 r19 r20
r2	s1	s10	p1:p p2:m	s3 s2 s7 s9	b2, p1, b5, b6	r1 r3 r5 r6 r9 r10 r11 r12 r15 r16 r19 r20
r3	s8	s11	p2:p	s6 s9 s12	b6, p2, b7	r2 r4 r7 r8 r9 r10 r13 r14 r19 r20
r4	s10	s11	p2:p	s6 s12	p2, b7	r3 r7 r8 r9 r10 r13 r14
r5	s1	s4	p1:m p2:p	s7 s2 s3	b2, p1, b3	r1 r2 r6 r11 r12 r14 r15 r16 r17 r18
r6	s1	s6	p1:m p2:p	s7 s2 s3 s5	b2, p1, b3, b4	r1 r2 r5 r7 r11 r12 r13 r14 r15 r16 r17 r18
r7	s4	s11	p2:m	s10 s5 s12	b4, p2, b7	r3 r4 r6 r8 r9 r10 r13 r14 r17 r18
r8	s6	s11	p2:m	s10 s12	p2, b7	r3 r4 r7 r9 r10 r13 r14
r9	s12	s9	p2:p p1:m	s6 s11 s10	b7, p2, b6	r2 r3 r4 r7 r8 r10 r13 r14 r19 r20
r10	s12	s7	p2:p p1:m	s6 s11 s10 s8	b7, p2, b6, b5	r1 r2 r3 r4 r7 r8 r9 r11 r13 r14 r19 r20
r11	s9	s2	p1:p	s3 s8 s1	b5, p1, b2	r1 r2 r5 r6 r10 r12 r15 r16 r19 r20
r12	s7	s2	p1:p	s3 s1	p1, b2	r1 r2 r5 r6 r11 r15 r16
r13	s12	s5	p2:m p1:p	s10 s11 s6	b7, p2, b4	r3 r4 r6 r7 r8 r9 r10 r14 r17 r18
r14	s12	s3	p2:m p1:p	s10 s11 s6 s4	b7, p2, b4, b3	r3 r4 r5 r6 r7 r8 r9 r10 r13 r15 r17 r18
r15	s5	s2	p1:m	s7 s4 s1	b3, p1, b2	r1 r2 r5 r6 r11 r12 r14 r16 r17 r18
r16	s3	s2	p1:m	s7 s1	p1, b2	r1 r2 r5 r6 r11 r12 r15
r17	s4	s6		s5 s3	b4, b3	r5 r6 r7 r13 r14 r15 r18
r18	s5	s3		s4 s6	b3, b4	r5 r6 r7 r13 r14 r15 r17
r19	s8	s10		s9 s7	b6, b5	r1 r2 r3 r9 r10 r11 r20
r20	s9	s7		s8 s10	b5, b6	r1 r2 r3 r9 r10 r11 r19

A second extended railway design:



ID	Source	Destination	Points	Signals	Path	Conflict
r1	s1	s4	p1:m p2:p	s5 s2 s3	b2, p1, b3	r2 r13 r15 r16
r2	s1	s6	p1:p p2:m	s3 s2 s5	b2, p1, b4	r1 r14 r15 r16
r3	s6	s7	p2:p	s4 s8	p2, b5	r4 r13 r14
r4	s4	s7	p2:m	s6 s8	p2, b5	r3 r13 r14
r5	s7	s10	p3:m p4:p	s11 s8 s9	b6, p3, b7	r6 r9 r11 r12
r6	s7	s12	p3:p p4:m	s9 s8 s11	b6, p3, b8	r5 r10 r11 r12
r7	s10	s13	p4:m	s12 s14	p4, b9	r8 r9 r10
r8	s12	s13	p4:p	s10 s14	p4, b9	r7 r9 r10
r9	s14	s9	p4:m p3:p	s12 s13 s10	b9, p4, b7	r5 r7 r8 r10
r10	s14	s11	p4:p p3:m	s10 s13 s12	b9, p4, b8	r6 r7 r8 r9
r11	s9	s8	p3:m	s11 s7	p3, b6	r5 r6 r12
r12	s11	s8	p3:p	s9 s7	p3, b6	r5 r6 r11
r13	s8	s3	p2:m p1:p	s6 s7 s4	b5, p2, b3	r1 r3 r4 r14
r14	s8	s5	p2:p p1:m	s4 s7 s6	b5, p2, b4	r2 r3 r4 r13
r15	s3	s2	p1:m	s5 s1	p1, b2	r1 r2 r16
r16	s5	s2	p1:p	s3 s1	p1, b2	r1 r2 r15

Output from the program:

5.3 Testing results

No	Functionality under test	Expected output	Result	Comment
1	Reading network specification from a file			
1.1	Read a block with valid structure in from a file	Block is successfully read in with the correct id, signal(s), neighbour(s), track position and stored	Works as expected	
1.2	Read a block with an invalid structure in from a file	Block is not successfully read or stored with an exception throw	Program throws an exception	
1.3	Read block in with invalid details (track, type, neighbours)	Block not accepted by program	Works as expected	see Fig 1.3
1.4	Read in block with more than two neighbours	Block is not successfully read or stored with an exception throw	Program throw an exception with feedback message	see Fig 1.4
1.5	Read in block with no neighbours (NA for both left and right neighbours)	Block should be accepted but should not be possible to be used in a route	Block accepted but cannot be used	
1.6	Read a point with valid structure in from a file	Point is successfully read in with the correct id and neighbours and stored	Works as expected	
1.7	Read a point with an invalid structure in from a file	Point is not successfully read or stored with an exception throw	Program throws an exception with feedback message	see Fig 1.7
1.8	Read in point with less than 3 neighbours	Point is not successfully read or stored with an exception throw	Program throw an exception with feedback message	see Fig 1.8
1.9	Read in point with more than 3 neighbours	Point is not successfully read or stored with an exception throw	Program throw an exception with feedback message	see Fig 1.9
1.10	Reading a file with valid data structure of the rail network	Accept the file and begin processing without errors	Works as expected	
1.11	Reading a file with invalid data structure of the rail network	Reject the file and give feedback message to the user	Rejects the file but doesn't give appropriate feedback message to the user	
2	Produce visual representation of the current state of the rail network			
2.1	Process a file and produce valid representation of the network	Graphical and accurate representation of the rail network specified in loaded file	Not implemented	

3	Calculate the settings for interlocking			
3.1	Enter rail network from specification to check if the points are set to correct position	Points are set to the correct position to prevent derailment	Works as expected	
3.2	Enter rail network from specification to check if the correct signals are set to 'STOP'	Appropriate signals are set to 'STOP' to prevent following and head on collisions	Works as expected	Some of the downward signals weren't set to Stop. Solved in 2nd version
3.3	Enter rail network from specification to check if the correct path is displayed	Path that the train travels across is displayed	Works as expected	
3.4	Enter rail network from specification to check if all conflicts are displayed	All conflicting routes are correctly calculated by the program	Works as expected	
3.5	Enter more complicated rail network (With two blocks between points)	The table produced by the program will be the same as the results we produced ourselves	When exiting a point with at least one block neighbour between start and point, signal wasn't set in neighbouring block(s) to prevent oncoming collision	Solved in 2nd version
3.6	Enter an extended rail network (two sets of points)	The table produced by the program will be the same as the results we produced ourselves	Works as expected	
4	Display a table showing the interlocking settings			
4.1	Process a file and produce valid representation of the network	Display accurate representation of the rail network specified in loaded file in a form of table	Some conflicts missing (first one in first two) in extended NW	Solved in 2nd version. See Figure 4.1
5	Allow the user to define journey from one part of the network to another from the command line			
5.1	Define a journey from beginning to end	Program produces correct interlocking setting for all appropriate routes	Works as expected	
5.2	Define multiple journeys	Program produces correct interlocking setting for all appropriate routes	Works for the majority of routes with the exception of only first 2 journeys missing some conflicts	Fixed in 2nd version
5.3	Define a correct route from signal A to signal B	Correctly calculate interlocking settings	Works as expected except in previously highlighted exceptions (in 3rd test)	Fixed in 2nd version

5.4	Enter only starting signal	Route should not be accepted by program and give feedback message to the user to enter correct journey	Not accepted, program relies on starting signal being entered before continuing	
5.5	Enter starting signal in invalid format in command line (e.g. 11 instead of s11)	Starting signal should not be read in correctly - error thrown	Starting signal not read in correctly	Fixed in 2nd version
5.6	Enter only destination signal	Route should not be accepted by program and give feedback message to the user to enter correct journey	Not accepted, program relies on destination signal being entered before continuing	
5.7	Enter destination signal in invalid format in command line (e.g. 16 instead of s16)	Destination signal should not be read in correctly - error thrown	Destination signal not read in correctly	Fixed in 2nd version
5.8	Don't enter any signals for a route	Program should give feedback message to the user to specify correct journey	Not accepted, program relies on both signals being entered before continuing	
5.9	Enter invalid/non-existent signals	Program should give feedback message to the user to specify correct journey	Not accepted / Exception thrown, feedback message given	see figure 5.9
5.10	Enter two opposite signals (eg 'Up' signal to 'Down' signal and vice versa)	Program should not accept this input and provide feedback message to the user	Not accepted / Exception thrown - feedback message provided to user	See figure 5.10
5.11	Move from a signal on minus/plus track to a signal on the adjacent plus/minus track	Program should not accept this input and provide feedback message to the user	Not accepted / Exception thrown - feedback message to user	see figure 5.11
5.12	Defining a route that travels through two points	Program should not accept this input and provide feedback message to the user	Not accepted / Exception thrown - feedback message provided to user	See figure 5.12
6	Perform checks against head-on collisions			
6.1	Enter two routes of length 1 for trains going in opposite directions along a main rail	Program should give a warning message / error that the journey is unsafe	No conflict when moving one block to a neighbouring block and vice versa in the opposite direction	Fixed in 2nd version
6.2	Enter two routes for trains going in opposite directions along a main rail	Program should give a warning message / error that the journey is unsafe	Works as expected - conflict shown in table	
6.3	Train enters a point	The opposite point should be set to the opposite position to provide protection	Correctly set in the interlocking settings	

6.4	For any route specified by the user	The appropriate signals will be set to stop to keep the route clear to avoid collisions	Signals work as expected except when exiting a point with at least one block neighbour between start and point, signal wasn't set in neighbouring block(s) to prevent oncoming collision	Fixed in version 2
6.5	Two trains travelling in opposite directions	The signals never allow them to occupy the same block at the same time	Works as expected	
7	Perform checks against following collisions			
7.1	For any route specified by the user	The appropriate signals will be set to stop to keep the route clear to avoid collisions	Works as expected	
7.2	Two trains travelling in the same direction	The signals never allow them to occupy the same block at the same time	Signals don't prevent following collisions but conflicts prevent them	
8	Perform checks against derailment collisions			
8.1	Train enters a point	The point should be set to correct position (PLUS for main rail and MINUS for adjacent rail) and the opposite point should be set to the opposite sign	Works as expected	
8.2	Train exits a point	The point should be set to correct position (PLUS for main rail and MINUS for adjacent rail)	Works as expected	

5.4 Screenshots of validation

Fig 1.3

```
Block b2 is invalid.
Exception in thread "main" Please input a valid section ie. b1, p3 OR input NA
NetworkRepresentation.InvalidNetworkException
```

Fig 1.4

```
Exception in thread "main" NetworkRepresentation.InvalidNetworkException: Block b2: Please input valid signal eg. s1
```

Fig 1.7

```
Point p1 is invalid.
Exception in thread "main" NetworkRepresentation.InvalidNetworkException
Please input a valid block eg. b1 NOTE: Points must have 3 Neighbours
```

Fig 1.8

```
Point p1 is invalid.
Exception in thread "main" NetworkRepresentation.InvalidNetworkException
        at NetworkRepresentation.ValidateNetwork.validatePoint(ValidateNetwork.java:34)
        at fileReading.GMLReader.readFile(GMLReader.java:67)
        at test.test.main(test.java:24)
Please input a valid block eg. b1 NOTE: Points must have 3 Neighbours
```

Fig 1.9

```
Point p2 is invalid.
Exception in thread "main" NetworkRepresentation.InvalidNetworkException
        at NetworkRepresentation.ValidateNetwork.validatePoint(ValidateNetwork.java:34)
        at fileReading.GMLReader.readFile(GMLReader.java:67)
        at test.test.main(test.java:24)
Please input a valid block eg. b1 NOTE: Points must have 3 Neighbours
```

Fig 4.1

ID	Source	Destination	Points	Signals	Path	Conflict
r1	s1	s4	p1:m p2:p	s5 s2 s3	b2, p1, b3	r2 r5 r7 r8
r2	s1	s6	p1:p p2:m	s3 s2 s5	b2, p1, b4	r1 r6 r7 r8
r3	s4	s7	p2:m	s6 s8	p2, b5	r4 r5 r6
r4	s6	s7	p2:p	s4 s8	p2, b5	r3 r5 r6
r5	s8	s3	p2:m p1:p	s6 s7 s4	b5, p2, b3	r1 r3 r4 r6
r6	s8	s5	p2:p p1:m	s4 s7 s6	b5, p2, b4	r2 r3 r4 r5
r7	s5	s2	p1:p	s3 s1	p1, b2	r1 r2 r8
r8	s3	s2	p1:m	s5 s1	p1, b2	r1 r2 r7

Fig 5.9

```
Please Enter a beginning destination ------ Such as S1
s1
Please Enter a final destination ------ Such as S6
b
Destination must be in the following format S1 or s1, please input route again
Please Enter a beginning destination ------ Such as S1
```

fig 5.10

```
Please Enter a beginning destination ------ Such as S1
s1
Please Enter a final destination ------ Such as S6
s2
Exception in thread "main" Routes.InvalidRouteException: Route r1: Cannot go from a signal in one direction, to a signal in another direction.
```

fig 5.11

```
Please Enter a beginning destination ------ Such as S1
s4
Please Enter a final destination ------ Such as S6
s8
Exception in thread "main" Routes.InvalidRouteException: Cannot create a route which goes back on itself.
```

fig 5.12

```
Please Enter a beginning destination ------ Such as S1
s1
Please Enter a final destination ------ Such as S6
s11
Exception in thread "main" Routes.InvalidRouteException: Route r1: Routes can only pass through one point.
```

6. Evaluation

6.1 Requirements Not Met

Due to the time constraint that was imposed by the specification, we unfortunately could not allocate our time to implement certain features that were outlined within the initial requirements. We decided as group that the features that were concerned with the safety and reliability of the system were more important rather than some cosmetic features such as a clear and easy to use GUI.
The functional requirements that were not implemented are listed below:

- Produce a graph from the file entered by the user (**FR2**)
- Export a table as a file (**FR7**)
- Data entered by user should be restored in the event of application crash (**FR13**)

6.2 Future work / Possible extensions

As previously mentioned we did not have enough time to implement all of the features which we planned to include in the original functional requirements. A feature we wanted to add was a visualisation of the network in the form of a Graphical User Interface. As the underlying framework for this has been provided in the form of the network data structure, we feel like this would be a feasible extension to the system if the timeframe allowed for it. In addition to this as we already have the basis for a safe network that allows multiple trains to conduct routes throughout it, it would be possible to simulate the movement of trains on the provided network using a visual interface.

Future work could also be done to the program by allowing the user to construct their own network or edit an existing one if the train network were to change from its original format. This could be accomplished by implementing a feature would could allow the user to construct sections (either blocks or points) on the fly and then adding the connecting tracks from the new railway section(s) to the existing ones.

We also considered the possibility of exporting and saving a network and its interlocking settings to a file but again we could not fit it into our timeframe. This feature could be implemented if future work on the system was to be done.

Another extension would be to model trains moving throughout the track, currently we have the system in the form that a train reserves a full length of track when it needs to complete a route. An extension would be to have a live update for the train at each block. So a train only

occupies a small section of track and the decisions about block safety are made before entering and leaving said blocks

As trains never travel at a constant speed, it could be possible be implement a train class that includes a speed variable to perform further checks against following collisions as well as to help visualise the the movement of trains throughout the network. By allowing trains to adjust their speed, the duration of time that sections of track are reserved for routes would be reduced resulting in more journeys being completed during a certain timeframe.

6.3 Group Personnel Assessment

In our group the work was divided equally so that all the team members have put the same amount of effort into the whole project.

NAME	ID	Percentage Contribution
Ali Alssaiari	109265362	20%
Jack Chandler	120232420	20%
Ryan Crosby	120248937	20%
Lee Robinson	120286250	20%
Maciej Sokolowski	120357332	20%

6.4 Resources and references

GitHub link to project source code: https://github.com/edwardjackchandler/trainnetwork.git

OVal - the object validation framework for Java™ 5 or later – . 2016. *OVal - the object validation framework for Java™ 5 or later* – . [ONLINE] Available at: http://oval.sourceforge.net/. [Accessed 10 March 2016].

The j-text-utils Open Source Project on Open Hub. 2016. *The j-text-utils Open Source Project on Open Hub*. [ONLINE] Available at: https://www.openhub.net/p/j-text-utils. [Accessed 10 March 2016].

Appendix

Classes and methods

1. **Reader**
   - 1.1. readFile (String path) - Using the path provided in the parameter, this method reads through the network structure and calls the appropriate methods. Either Block get instance or Point get instance. Throws a file not found exception.
   - 1.2. getStoreBlock() – Returns the list of Blocks associated with the Network
   - 1.3. getStorePoint() – Returns the list of Points associated with the Network
   - 1.4. getStoreSection() – Returns the list of Points and Blocks associated with the Network

NetworkRepresentation

1. **Section**
   - 1.1. getNeighList() - Returns a List containing Points and Blocks
   - 1.2. addNeigh(Section section) - Adds another neighbour (block/point) to section

2. **Block (Inherits Section)**
   - 2.1. Block (String bID, String leftNeighbour, String rightNeighbour, Boolean plus, String signalDown, String signalUp) - Constructor method for creating each block using the parameters provided. This method uses regular expressions in an OVal @MatchPattern constraint to ensure that the parameters are in the correct format and throws the appropriate exception if not.
   - 2.2. getInstance (String bID, String leftNeighbour, String rightNeighbour, Boolean plus, String signalOne, String singalTwo) - Object factory for Blocks, stores the block in a hash map, this is to ensure multiple blocks are not entered more than once. It also is a convenient way to store the blocks for later access.
   - 2.3. getSingalFromSigID (int sigID) - Returns a signal based upon the signal id, depending on whether the signal is up or not. Returns null if no signals are associated with the block.
   - 2.4. isPlus() - Returns a boolean based on whether the block is on the main/plus rail (*true*) or the adjacent/minus rail (*false*).
   - 2.5. addSignals (Signal signalDown, Signal signalUp) - If the block has two signals associated with it, the signals are added in the correct direction.
   - 2.6. addSingalUp (Signal signalUp) - If the block was only one up signal, the signal is assigned
   - 2.7. addSingalDown (Signal signalDown) - If the block was only one down signal, the signal is assigned
   - 2.8. hasOneNeighbour() - Returns a boolean that checks if the block is at the start or end of the network
   - 2.9. getLeftNeighbour() - Returns neighbouring block to the left if applicable
   - 2.10. getRightNeighbour() - Returns neighbouring block to the right if applicable

2.11. ifNull(Signal s) - Checks if there is a signal associated with the block, returns NA if false or the signal id if true
2.12. ifNullState(Signal s) - If block contains a signal, returns the signal's state - either *stop* or *clear*

3. Point (inherits Section)

3.1. Point (String pId, String neighbour1, String neighbour2, String neighbour3) - Constructor method for creating each block using the parameters provided. This method uses regular expressions in an OVal @MatchPattern constraint to ensure that the parameters are in the correct format and throws the appropriate exception if not.
3.2. getInstance(String pId, String neighbour1, String neighbour2, String neighbour3) - Object factory for Points, stores the block in a hash map, this is to ensure multiple blocks are not entered more than once. It also is a convenient way to store the blocks for later access.
3.3. isPlus() - Checks to see if the boolean plus is true (main rail) or false (adjacent rail)
3.4. setPair() - Creates a connection from a point to its neighbouring point
3.5. addToGlobalList(List<Point> pointList) - Adds the point to a global list of Points.
3.6. pointFacingUp() - Returns a boolean based upon whether the point is on the left hand side (*true)* or the right hand side (*false*).
3.7. pointPair(Point p2) - Returns a boolean as true if the two points are paired, false if they are not.
3.8. pointFacingRouteDirection(Route r) - Returns a boolean as true if the point is facing up when the route moving from left to right or false otherwise.

4. Signal

4.1. Signal(int sigId, boolean up) - Constructor method for creating each signal using the parameters provided. This method uses regular expressions in an OVal @MatchPattern constraint to ensure that the parameters are in the correct format
4.2. getSigId() - Returns signal id
4.3. setSigId(int sigId) - Sets signal id
4.4. isUp() - Returns a boolean based on which direction the signal controls: *true* for Up and *false* for Down
4.5. isStop() - Returns a boolean as true if the signal is set to stop.
4.6. state() - Returns a string representing the signals state, “stop” or “clear” depending on the Stop boolean
4.7. setStop() - Sets the signal to stop to prevent trains from passing
4.8. setClear() - Sets the signal to clear to allow trains to pass through
4.9. toString() - Prints out the signals unique id
4.10. getOwner() - Returns the block associated with the signal

4.11. setOwner(Block owner) - Sets the block which the signal is connected to.

5. ValidateNetwork

5.1. validateBlock(Block b) - Performs a check, whether block data entered in the input file is valid or not, if not it returns an exception

5.2. validatePoint(Point p) - Performs a check, whether point data entered in the input file is valid or not, if not it returns an exception

5.3. validateSignals(HashMap<Integer, Block> signalMap) - Performs a check, whether signal data (in the blocks) entered in the input file is valid or not, if not it returns an exception

6. CreateNetwork

6.1. blockLookup(Reader reader, int bId) - Iterates through all the blocks loaded in input file, searches for block with specific ID and return it

6.2. pointLookup(Reader reader, int pId) - Iterates through all the points loaded in input file, searches for block with specific ID and return it

6.3. addEdges(Reader reader) - Adds a connection between block with other block and block with a point, in order to create a consistent network

6.4. findPairs(Reader reader) - Iterates through all the points loaded in input file, finds points responsible for entering and exiting of the same adjacent track and sets them into pairs

6.5. populateSignalMap(Reader reader) - Adds signals and blocks containing those signals to map, where signal is key and block is a value

6.6. blockBySignal(int sigId) - Searches for block which contains specific signal by that signal ID and returns it

7. InvalidNetworkException

7.1. InvalidNetworkException(String message)

7.2. InvalidNetworkException(List<ConstraintViolation> violations)

Routes

1. Route

1.1. Route(int rId, int sourceId, int destId) - Constructor method for creating each Route using the parameters provided. This method also decides which direction the route is going (Up/Down).

1.2. getInstance(int rId, int sourceId, int destId) - Creates a Map of routes to store them in the system.

1.3. getrId() - Returns the route id

1.4. setrId(int rId) - Sets the unique route id

1.5. isUp() - Returns a boolean based on which direction the signal controls: *true* for Up and *false* for Down.

1.6. getPath() - This method returns each section of the track the route traverses across.

1.7. setPath(List<Section> path) - Sets the path which the route takes the train across.
1.8. getSourceId() - Returns the signal which the route began from.
1.9. getDestId() - Returns the signal which the route finishes at.
1.10. getRoutes() - This method returns a HashMap which stores all the routes that have been previously defined.
1.11. getSourceOwner() - Returns the block from which the route began.
1.12. getDestOwner() - Returns the block which the route ends on.
1.13. populatePath(Block sourceBlock, Block destBlock) - Depending on the direction of the route, this method will find the neighbour of a section starting with the sourceBlock, and add it to the path. It will keep adding blocks and a single point to the path using different conditional statements until the current block is the same as the destinationBlock, which is where it will stop.
1.14. hasPoint() - This method returns a boolean based on whether the route goes through a point.
1.15. getPoint() - Loops through the routes path and returns a boolean if a point is included.
1.16. printRoute() - Method which loops through the path, printing out the source and destination and all sections in between.
1.17. getPathString() - Returns the path in a String format.
1.18. validRoute() - This method performs a number of checks to ensure that the route adheres to the rules previously defined in the design, for example an up signal cannot go to a down signal. This method throws the appropriate exception method if any of these rules are broken
1.19. toString() - Prints out the route id

2. InputJourney

2.1. inputRoute() - This method creates a route by asking the user for the signals he or she wishes to have for this route. The command line asks for how many routes is in the current journey. This method calls populatePath() and validNetwork()
2.2. jMake() - Calls a command line asking the user how many routes he or she wishes to enter. This then calls input route however many times the user specified.
2.3. getJourney()
2.4. getFailCounter()

3. InvalidRouteException

3.1. InvalidRouteException(String message)
3.2. InvalidRouteException(List<ConstraintViolation> violations)

Interlocking

1. **InterlockingTableGenerator**
   - 1.1. InterlockingTableGenerator() - This method is only responsible for calling method createTable()
   - 1.2. InterlockingTableGenerator(List<Route> journey) - Responsible only for calling method createTableNoInput()
   - 1.3. createTableNoInput() - It is responsible for creating table directly from data in testing class, without any actual input from the user. This method was used only for testing purposes and was not used in final version of the program
   - 1.4. addToTable() - This method adds every journey, that is made of routes to the interlocking table
   - 1.5. createTable() - This method creates a table by asking user how many journeys he will enter, it iterates exactly this many times, every time calling method createJourneys() and at the end it calls addToTable()
   - 1.6. createJourneys() - Calls jMake() from class InputJourney responsible for inputting routes and adds them to a list (creates journeys from routes)
   - 1.7. generateSettings(Route r) - Creates array consisting of route details
   - 1.8. pointSettings(Route r) - This method implements the design of calculating the point settings
   - 1.9. signalSettings(Route r) - This method implements the design of calculating the signal settings
   - 1.10. conflictSettings(Route r) - This method implements the design of calculating the conflict settings
   - 1.11. printTable() - prints interlocking table