

SketchRail
Train Scheduling Editor and Simulator

Devendra Shelar
07D05010

Guide: Prof. Abhiram Ranade
Prof. Narayan Rangaraj

June 26, 2012

Abstract

Train scheduling has been a very challenging problem for Indian Railways because of the huge demand for many varied type of services. A software SketchRail has been developed which allows designing track geometry of complex railway sections and allows input of desired train schedules to a simulator. The simulator uses priority based scheduling and operational constraints of block occupancies. The simulator was used to suggest effective scheduling strategies for a 3-line Thiruvallur-Arakkonam railway section in Southern Indian Railways.

Acknowledgements

I would like to thank Prof. Abhiram Ranade and Prof. Narayan Rangaraj for their constant motivation, support and guidance without which this project wouldn't have been possible for me. I would like to thank to Mr. Gopalkrishnan (Sr. DOM., Western Railways), Mr. Akhilesh Yadav (RDSO), Dr. Ravi Babu (RITES), Mr. Sadiq Ali, Mr. Mishra for giving us insights behind working of Indian Railways. I would like to thank my colleague Priya Agrawal who helped me improve the simulator. I also want to mention my friend Prashant Sachdeva for giving me inputs regarding this report.

Contents

1	Introduction	3
1.1	SketchRail	5
1.2	Additional features of editor	6
1.3	Summary of Work Done	6
1.4	Outline of the report	7
2	How To Use SketchRail	8
2.1	Adding stations	8
2.2	Adding loops	9
2.3	Adding blocks	9
2.4	Adding signals	9
2.5	Adding Links	9
2.6	Adding a track characteristic	10
2.7	Adding gradient effect	10
2.8	Tuning parameters	10
2.9	Adding Trains	11
2.10	Save Project tool	11
2.11	Open Project	11
2.12	Run Project	11
2.13	Editing properties	12
2.14	Dragging using mouse	12
2.15	Deletion of diagrams	12
3	Format of the intermediate files	27
4	Implementation of SketchRail	31
4.1	Building blocks of SketchRail code	31
4.1.1	Interfaces	32
4.1.2	Abstract Classes	34
4.1.3	Global Variables	36
5	Analysis of 3-line section	37
5.1	Description of Section	37
5.2	Strategies used	37
5.2.1	Traffic indicators	38
5.2.2	Fixed time strategy	40
5.2.3	Variable time strategy	41
5.3	Conclusion	41

6	Velocity Profile Array Generation	43
6.1	Velocity Profile Array Generation of train on tiny track segment	44
6.2	Generation of list of Tiny Track Segments	46
6.3	Velocity Profile Array Generation on a list of tiny track segments	49
6.4	Generation of Entire Velocity Profile Array of train	50
7	Work on Simulator and Visualizer	53
7.1	Re-modelling simulator to handle complex railway sections	53
7.1.1	Re-modelling of links	53
7.1.2	Re-modelling of gradients and speed restrictions	53
7.1.3	Changes in visualizer	54
7.1.4	Reference Timetables of Scheduled Trains	54
8	Conclusions	55
8.1	Future Work	55
8.2	Conclusions	55
	Appendices	59
.1	Terminology	59

Chapter 1

Introduction

Indian Railways has fourth largest railway network in the world and carries over 30 million passengers and 2.8 million tons of freight daily. The operating rules, diverse geographical conditions, variety of trains, and large amount of population and great industry demand make effective utilization of railway resources extremely challenging. Planning and assessment of large scale investments becomes critical. So, it is essential to have tools for practical decision support.

A simulator [1] had been developed at IIT Bombay for IRISSET (Indian Railways Institute of Signal Engineering and Telecommunications) to model the operational behaviour of trains in Indian railway sections in 2003. SketchRail was developed with the objective of providing a graphical interface to make the necessary inputs to the simulator in an efficient way. SketchRail also provides a bird's eye view of movement of trains along the railway tracks.

In figure 1.1 a complex railway section is shown. This isn't an exact replica of the real life railway section and it is being used just as an example. It has 3 terminal stations namely, Thane, Karjat and Kasara. There are three junctions at Thane, Diva and Kalyan. There are two railway tracks (one in each direction) between Karjat and Kalyan. Similarly, there are two tracks between Kasara and Kalyan as well. There are four railway tracks between Kalyan and Diva. Two of them reach Thane directly. These two tracks are primarily used for through trains, freight trains and fast suburban trains. There are stations of Mumbra and Kalva on the other two tracks between Diva and Thane. These two lines are primarily used for slow local suburban trains. The direction towards Thane is considered as *up direction*. The numbers mark the blocks and loops. Stations are the rectangular shapes. Straight lines guarded by signals represent blocks and loops. We can see a bidirectional loop at Kalyan station which allows train movement in both directions. Generally, blocks and loops are unidirectional unless the section has just one railway line. Links are straight lines that connect two blocks. A link can cross any number of other blocks which are called crossovers. The link joining loop 23 to block 56 crosses block 57. Hence, block 57 is a crossover for that link.

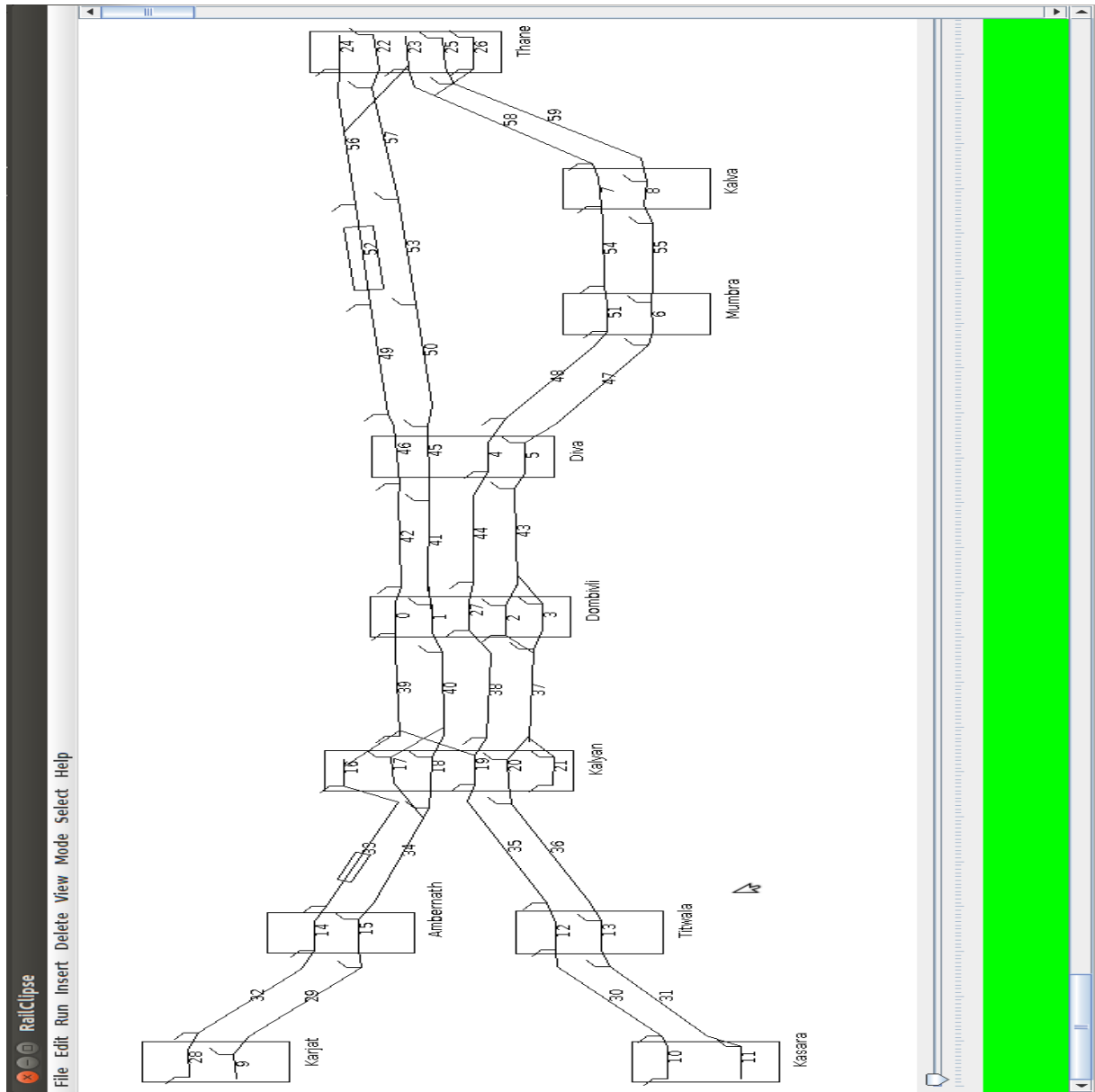


Figure 1.1: Complex Railway Section

In figure 1.2, information like train's path, its velocity profile, the signals as seen by the train, block occupancies, halts at stations etc. can be viewed.

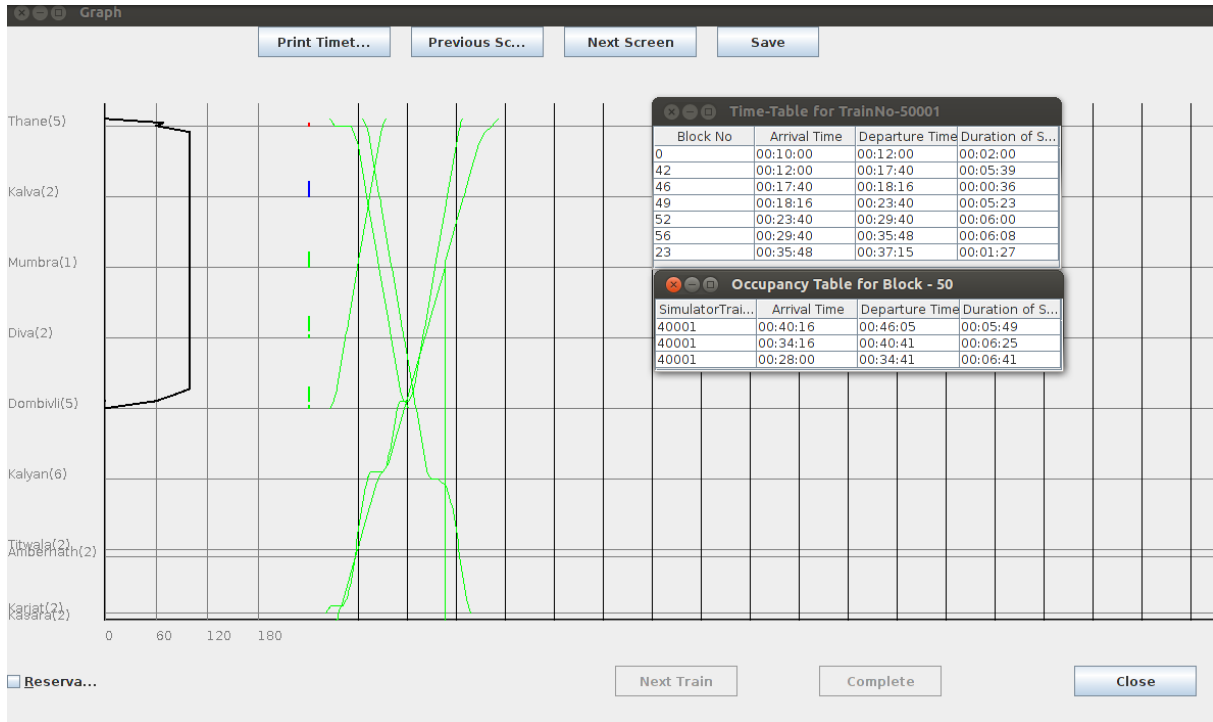


Figure 1.2: Visualizer

SketchRail has 3 major components to it which are illustrated in figure 1.3.

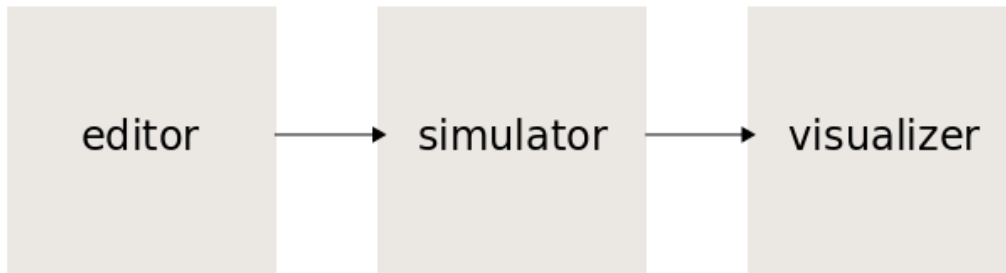


Figure 1.3: Components of SketchRail

1.1 SketchRail

Problem definition

We have to build a graphical user interface which would allow us to sketch the section track geometry and track characteristics. The interface should also allow input of information like train kinetics and their desired schedules. The GUI should be able to generate the input files for the above mentioned simulator.

It should have a feature to analyze train movements by displaying the locations of trains at various points in time. It should determine the consistency of the input and identify any errors. Additionally, it may have zoom-in zoom-out feature.

Input

The inputs to SketchRail are as follows:

- Exact track geometry of a railway section
 1. **Stations:** Their starting and ending mileposts.
 2. **Blocks and loops:** Their direction (unidirectional or bidirectional), their locations within the sections, lengths, speed limits and availability.
 3. **Links:** Their lengths, speed limits and their location relative to the blocks they join. Also, their crossovers.
- **Track characteristics:** Speed limits and/or gradients for every block along with their location and their effect on train running.
- **Kinetics of the trains:** Train's length, acceleration, deceleration, maximum speed etc.
- **Desired train schedules:** Trains' arrival and departure times at their starting and ending stations and intermediate halt stations, if any
- **Fine-tuning parameters:** Number of signal aspects, simulation time, warning distance etc.

1.2 Additional features of editor

In addition to accepting the input of the above entities and generating input files to the simulator, SketchRail should be able to do the following:

- After simulation, the editor should show movement of trains over the section over time.
- It does consistency checks. Any entities with errors in their attributes are marked in red. It should display an error if a train's destination loop is not physically reachable from its starting loop, etc.
- A zoom-in/zoom-out feature is to be added which will facilitate overlooking large railway sections.

1.3 Summary of Work Done

My contributions to this project include work on multiple things. The most important part of my project was to build SketchRail. The editor which can handle inputs of track geometry of complex railway sections like stations, blocks, loops, gradients, speed restrictions, trains, train timetables was built. Apart from providing an easy to use interface for the user, SketchRail does consistency checks on the attributes of the input. It shows logical errors in the input, if any. It facilitates addition, deletion and update of the various entities.

The simulator was used for suggesting better operating strategies for effective use of third line in a 3-line section in Southern Railways [2]. Analysis of scheduling strategies for a 3-line Tiruvallur-Arakkonam section was done as part of this study. Simulator code was modified to measure performance metrics of the section. Based upon the results obtained, we were able to decide what among the discussed strategies is the best strategy for the 3rd line for a given traffic scenario.

The simulator design was improved upon to simulate operational behaviour of trains over complex railway sections. Links, gradients and speed restrictions were remodelled. Train running model was understood and developed further.

Visualizer of the simulator was improved upon. Issues like proper depiction of train-halts were resolved.

1.4 Outline of the report

In chapter 2, how to use the editor to input a given test case is explained. Chapters 3 and ?? serve as a user manual for SketchRail . Chapter 3 includes the description of the format and content of the files generated by the editor. Chapter 4 describes the entity-relationship diagram of the entities used for SketchRail and gives the overview of how the code for editor is written. In chapter 5 application of simulator in the analysis of the 3-line railway section in Southern Railways is explained. Chapter 6 explains generation of velocity profile array for trains over blocks. In chapter 7, we explain the developments done in the simulator. Finally, we conclude the report with a discussion of the current limitations of the editor and future work to be done on SketchRail .

Chapter 2

How To Use SketchRail

In this chapter, we will describe how to draw a complex section diagram between the stations Thane, Kasara and Karjat. We intend to draw the section diagram shown as in 1.1.

A new instance of SketchRail looks like in the figure 2.1.

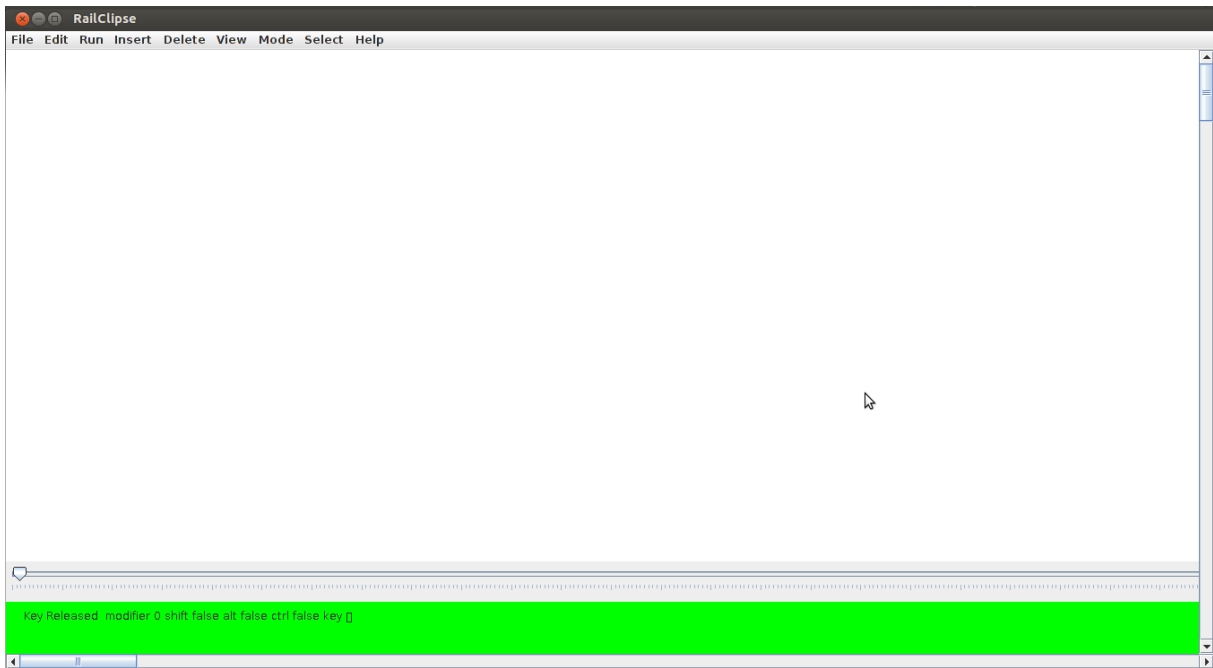


Figure 2.1: First Look

2.1 Adding stations

First stations are drawn and their attributes are set. Clicking shift once, a sufficiently large closed figure is drawn. Refer figure 2.2. The figure is interpreted as station and a station diagram is drawn. Refer figure 2.3. A corresponding entry is made in the input. Then double clicking the diagram, the user can edit the properties of the station. Refer figure 2.4. The entries are checked for wrong input. If the inputs are proper, the station attributes are changed. Refer figure 2.5. Refer figure 2.6 to see all the stations added.



Figure 2.2: Station drawn

2.2 Adding loops

Loops are then drawn. Clicking shift-key once, a line shaped figure is drawn within a station. The figure is interpreted as loop and a loop diagram is drawn. Then double clicking the diagram, the user can edit the properties of the loop. Refer figures 2.7, 2.8, 2.9 and 2.10 to see the steps for adding loops.

2.3 Adding blocks

Blocks are then drawn. Clicking shift-key once, a line shaped figure is drawn within a station. The figure is interpreted as a block and a corresponding block diagram is drawn. Double clicking the diagram, the user can edit the properties of the block. Refer figures 2.11 and 2.12 to see the steps for adding blocks.

2.4 Adding signals

Signals can be drawn to split existing blocks into smaller blocks. Clicking shift-key once, a small line shaped figure is drawn which intersects exactly one block. The figure is interpreted as signal and a signal diagram is drawn. A signal cannot be drawn on a loop. This results in creating two smaller blocks and the earlier larger block is removed from the system. Refer figure 2.13.

2.5 Adding Links

Links can be drawn to connect two blocks. Clicking shift-key once a curve is drawn intersecting more than one different block diagrams. A link diagram is drawn connecting the first and the last block diagram while the intermediate block diagrams are assigned as crossovers for the link. Refer figures 2.15, 2.5, 2.17 to see the steps for adding links. Refer figure 2.18 to view the entire track segments on the section.

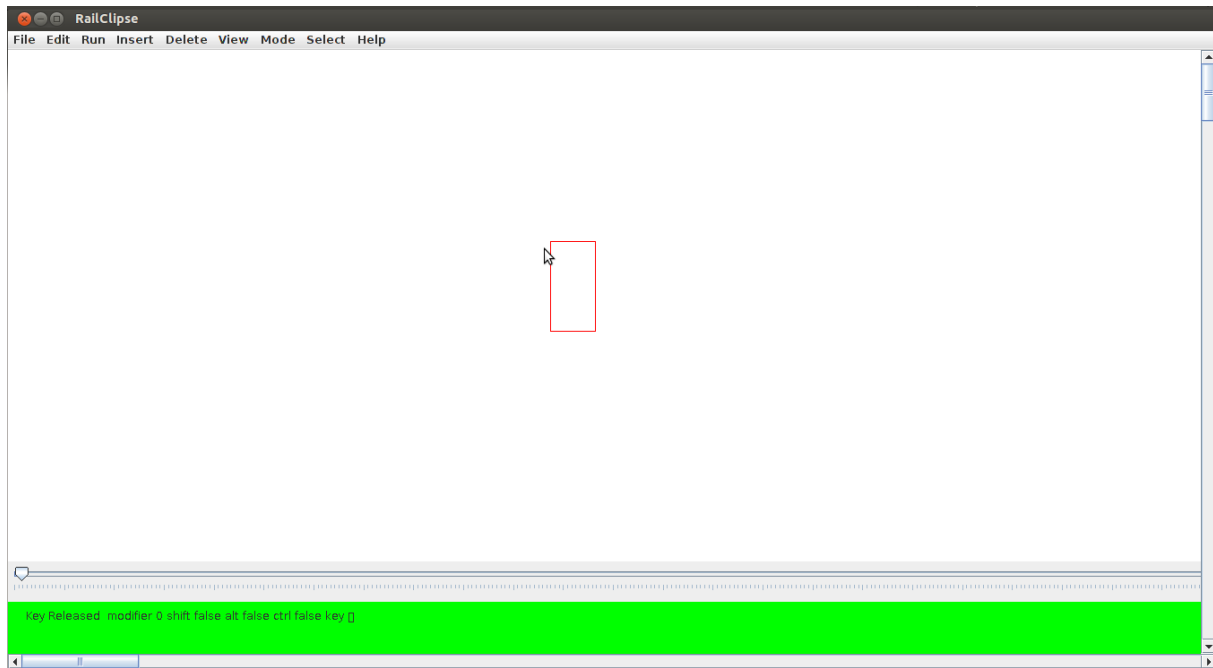


Figure 2.3: Station Added

2.6 Adding a track characteristic

Track characteristics are of two types, namely, gradients and speed restrictions. A track characteristic can be drawn on a block or a link. Shift is clicked once and a curve intersecting the same block more than once is drawn. If the curve intersects twice, it is interpreted as a speed restriction shown by a rectangle on the block diagram. If the curve intersects with the block thrice, it is interpreted as a gradient shown by a bigger rectangle. Double-clicking the diagram opens the dialog box to edit the properties of track characteristic. The projected relative start milepost is calculated by using the values of block length and relative start coordinate of the start point of the curve. Similarly, the projected relative end milepost is calculated. After clicking OK, the track characteristic is drawn to scale of the block. Refer figures 2.19, 2.20 and 2.21 to see the steps for adding gradients. Refer figures 2.22, 2.23 and 2.24 to see the steps for adding speed restrictions.

2.7 Adding gradient effect

Pressing Alt + g, opens the gradient effect properties dialog box. The user may insert the appropriate acceleration and deceleration changes that the gradient will cause. If these values were already inserted for that particular gradient slope, they will be overridden by the new values. Refer figure 2.25.

2.8 Tuning parameters

Pressing Alt + p, opens the parameters dialog box. The user may edit the appropriate parameters. Refer figure 2.26.

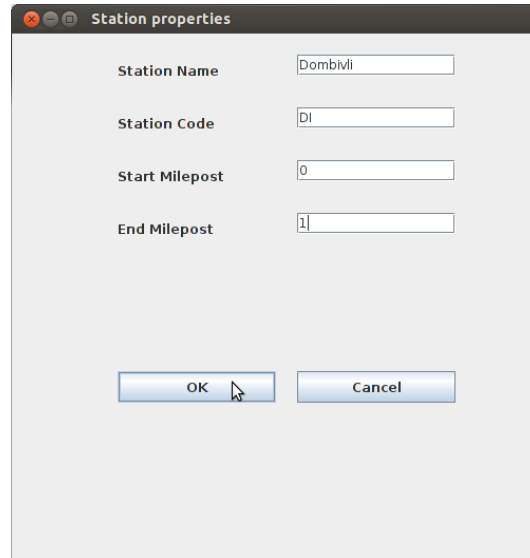


Figure 2.4: Station Properties

2.9 Adding Trains

By pressing Alt + 2, the user can see the interface for inputting train information, which we call the train-view-mode. By pressing Alt + 1, the user may go back to the interface for editing section geometry, which we call section-view-mode. In the train-view-mode, for adding a train, the user should press shift once. Then s/he should click at the point where the starting time and the line of the starting station intersects. S/he should next click on the intersection of expected arrival time at the ending station and the line of the ending station. The ending station shouldn't be the same as the starting station. A train diagram is created and a train properties dialog box pops up. Refer figure 2.27. Then user can click on the train diagram. Clicking on some station line, will add a timetable entry for the train at that station. The user can assign a loop line at the station for the train to halt. Refer figure 2.28. The user can change the arrival and departure times of the train by dragging the small rectangles. Refer figure 2.29.

2.10 Save Project tool

On completion of input, the user may save the entire work into a directory. Refer figure 2.30.

2.11 Open Project

The user may reopen the work that was previously stored in some directory. On navigating to the desired directory, clicking on open would result in reopening of the section diagram into SketchRail . Refer figure 2.31.

2.12 Run Project

After having completed the input, the user can run the simulation by pressing Ctrl + F10. Refer figure 2.32. Clicking on "Next Train" will generate distance vs time plot of the top priority train. The user may view the hidden information of block occupancies or train path occupancies by right-clicking and selecting the appropriate entity.

If the user presses Ctrl + F11, the simulations will run in background and determine the location of train as function of time. A time slider at the bottom allows to change the time for which the

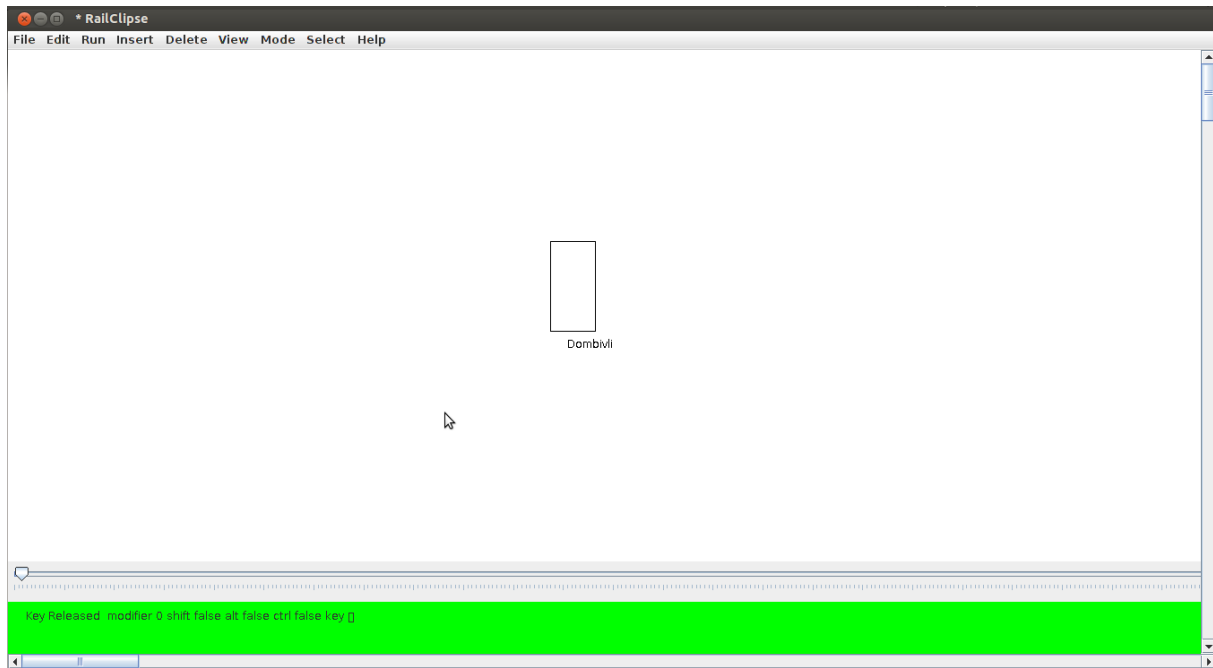


Figure 2.5: Station Completely Added

user wants to have the bird's eye view of the section. S/he can use Ctrl or Alt or Ctrl + Shift modifiers along with the left or right keys to change the value of time. Ctrl + Right arrow, Alt + Right arrow and Ctrl+Shift+Right arrow will increase time by 5 minutes, 1 minute and 10 seconds respectively. Similarly, Ctrl + Left arrow, Alt + Left arrow and Ctrl+Shift+Left arrow will decrease time by 5 minutes, 1 minute and 10 seconds respectively. Refer figure 2.33.

2.13 Editing properties

User may edit the properties of any diagram by double-left-clicking the diagram. The dialog box for the highlighted diagram will open.

2.14 Dragging using mouse

The position of the diagrams can be changed by dragging the diagrams appropriately. Only station, block, loop and train diagrams can be dragged. Hovering on a diagram will make the coordinates of the diagram visible. The user may click and drag the appropriate coordinate to resize/move the diagram. The arrival and departure times of the trains can be adjusted by dragging the time handles of that train.

2.15 Deletion of diagrams

User may delete a diagram by using the combination of Alt+Shift+LeftClick the diagram. The highlighted diagram will be deleted. Deleting a block will cause all the links and the signals associated with the block to disappear.



Figure 2.6: All Stations Completely Added

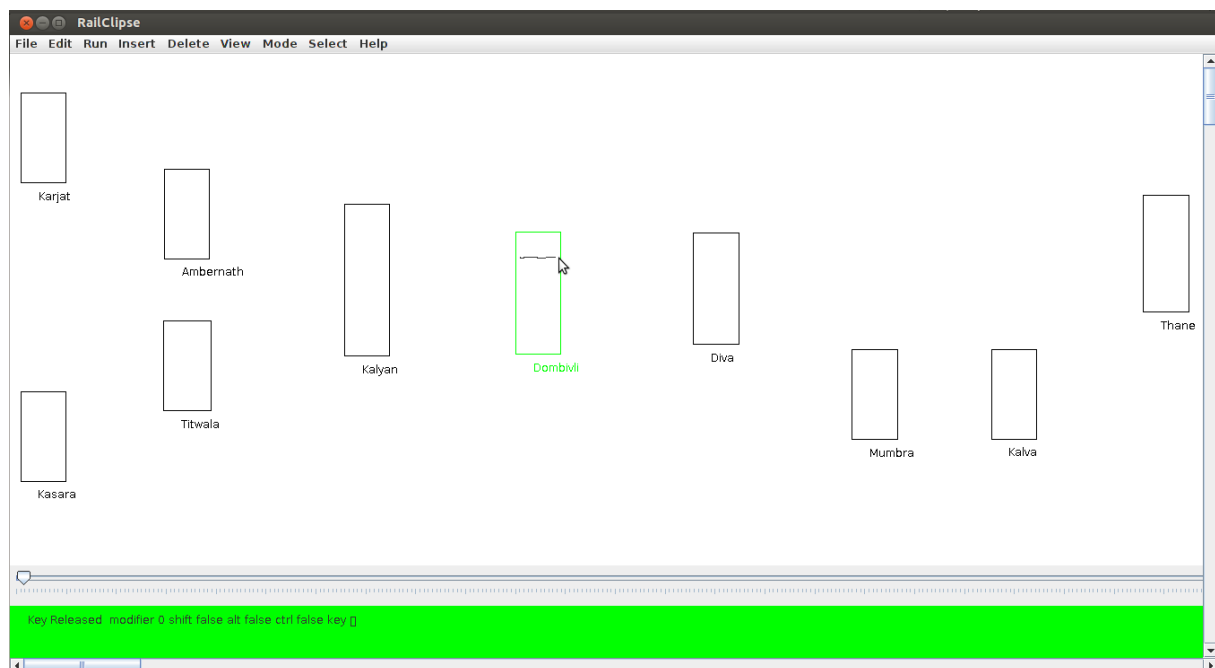


Figure 2.7: Loop drawn

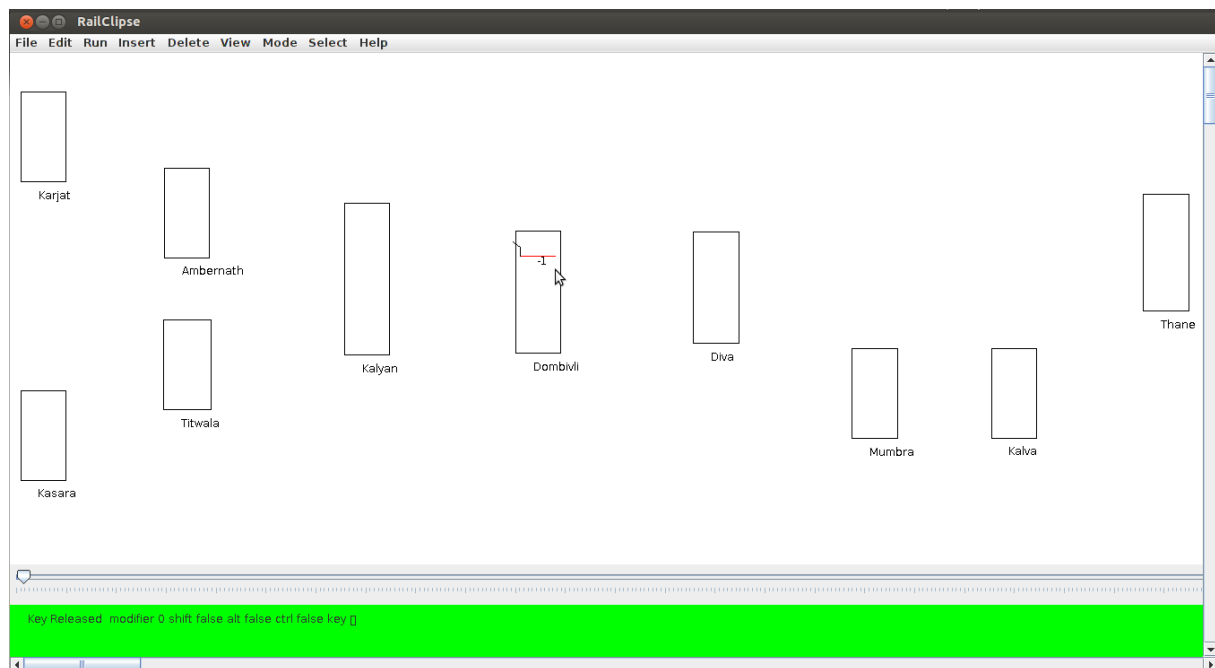


Figure 2.8: Loop Added

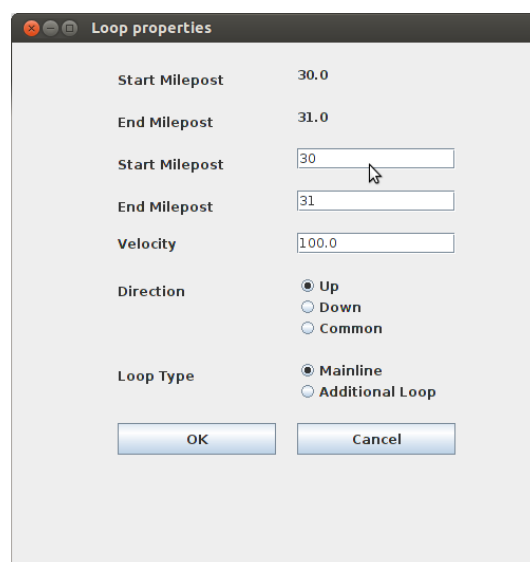


Figure 2.9: Loop Properties

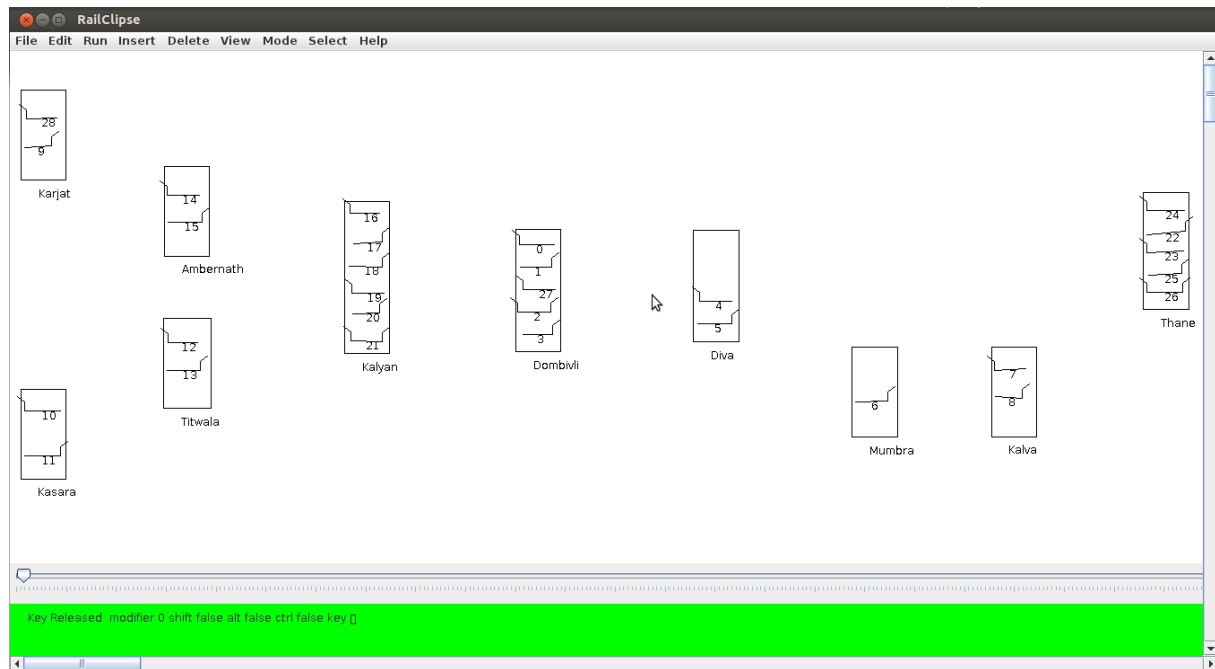


Figure 2.10: All Loops Completely Added

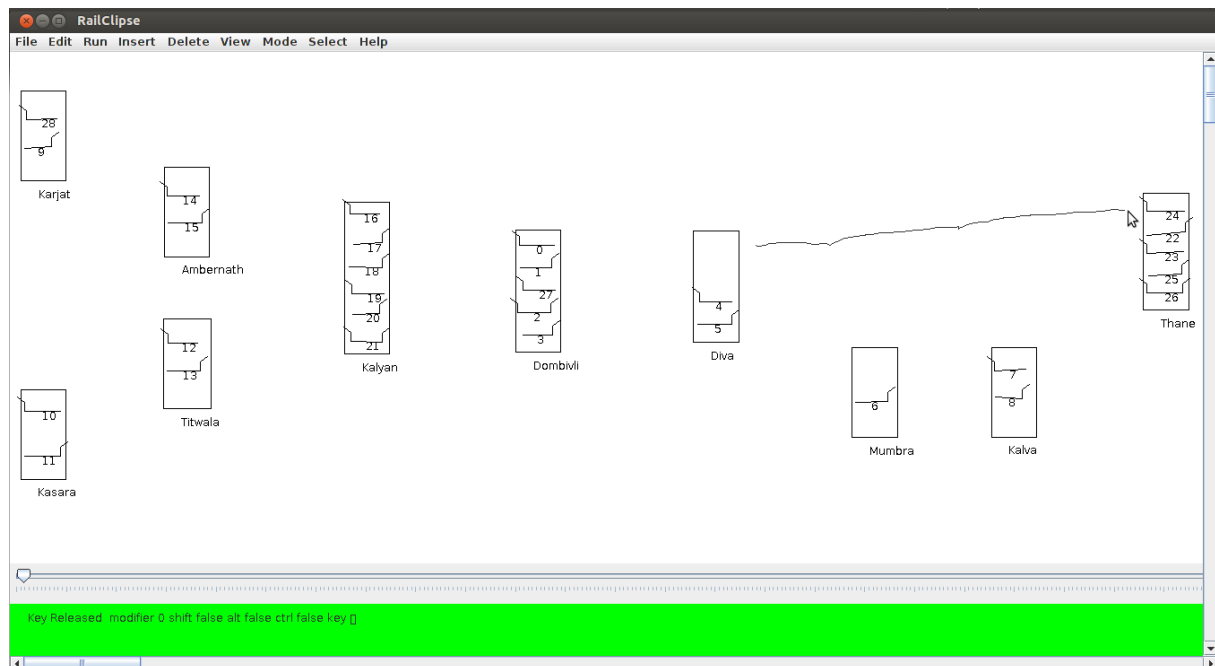


Figure 2.11: Block drawn

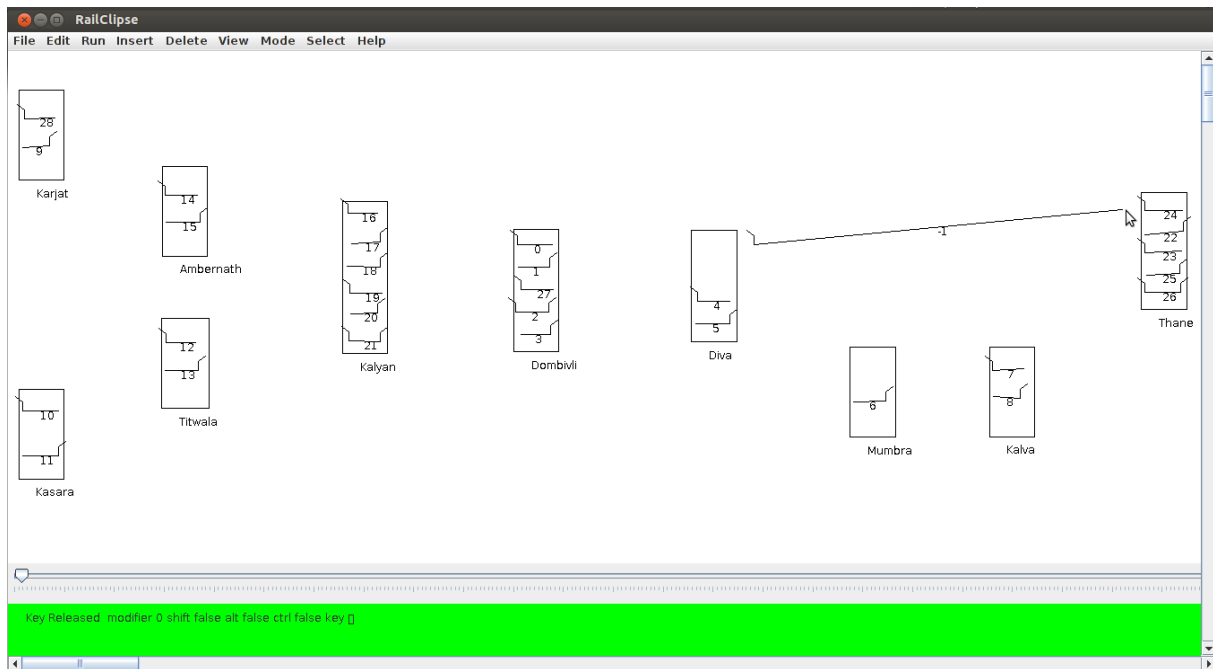


Figure 2.12: Block Added

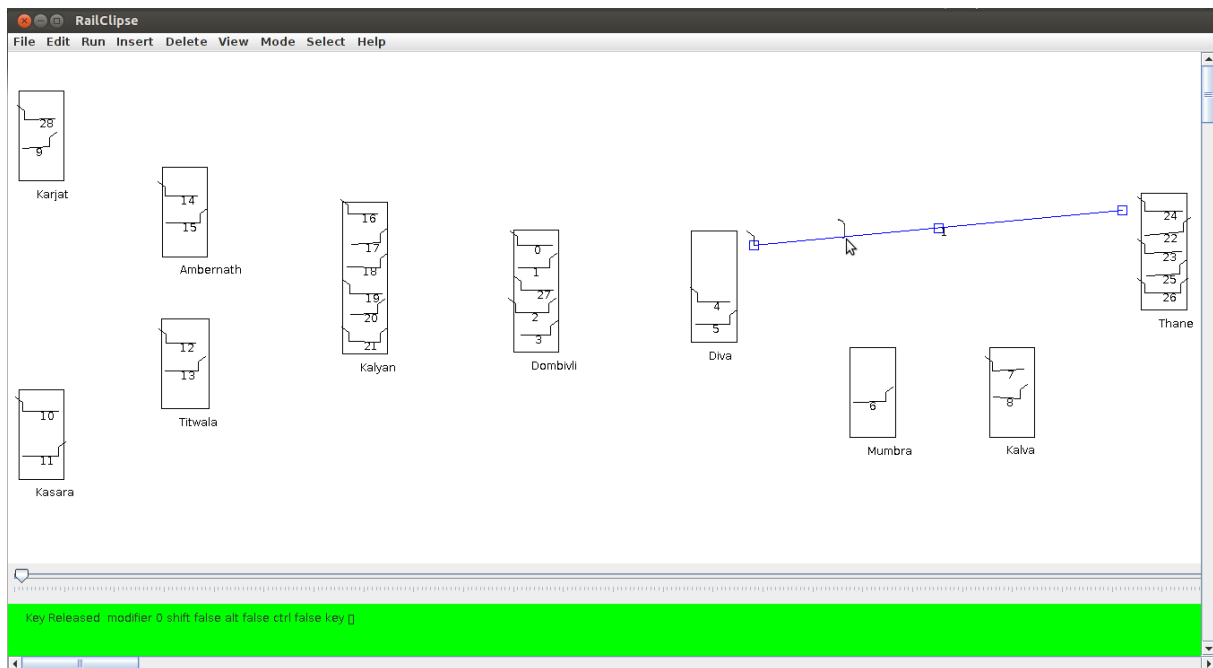


Figure 2.13: Signal drawn

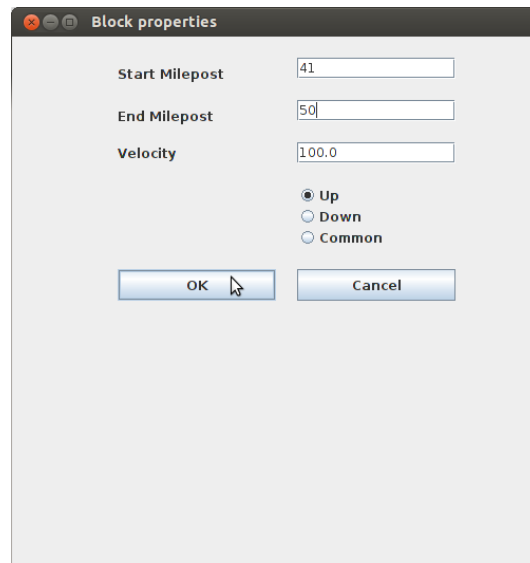


Figure 2.14: Block Properties

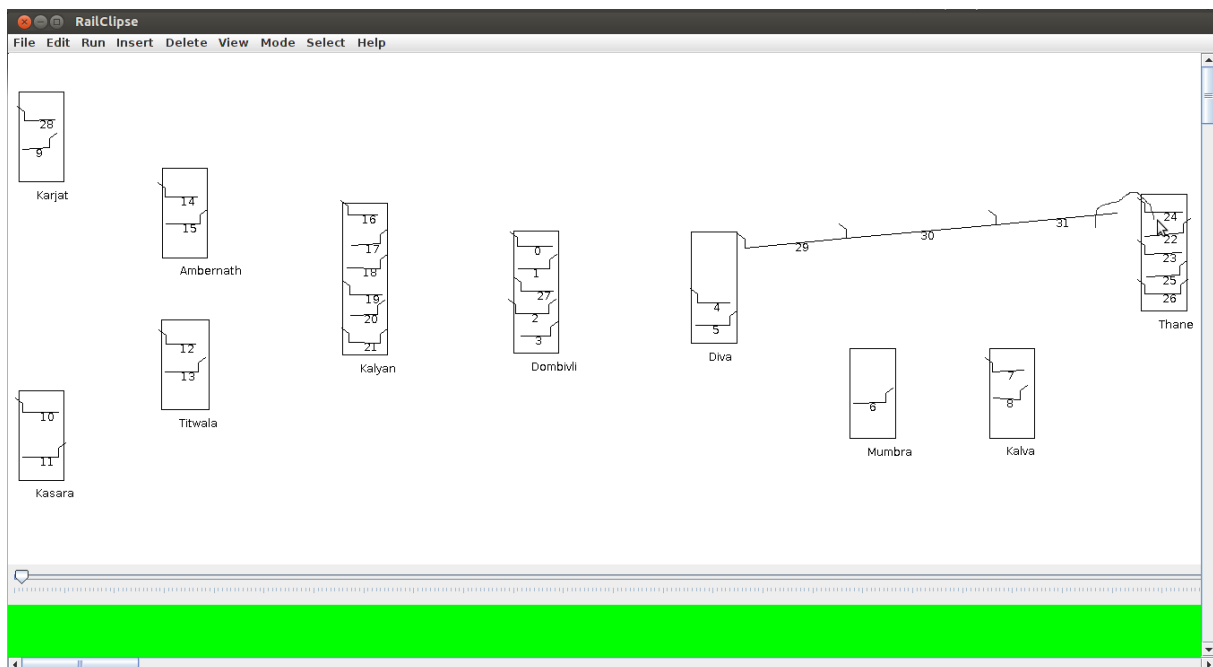


Figure 2.15: Link drawn

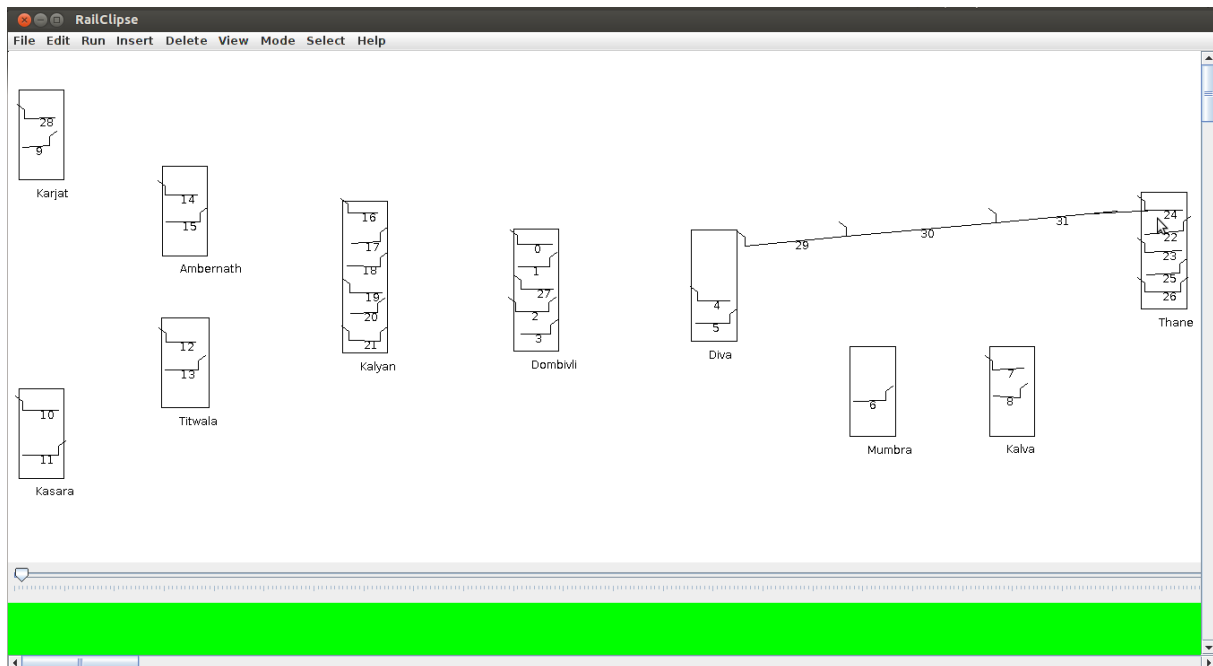


Figure 2.16: Link Added

The image shows a dialog box titled "Link between block 31 and block 24". It contains the following fields and controls:

- Previous Block Rel...: 0.0
- Next Block Relative...: 0.0
- Length: 0.0
- Priority: 1
- Maximum Speed All...: 15.0
- Direction: Up
- Buttons: OK, Cancel

Figure 2.17: Link Properties

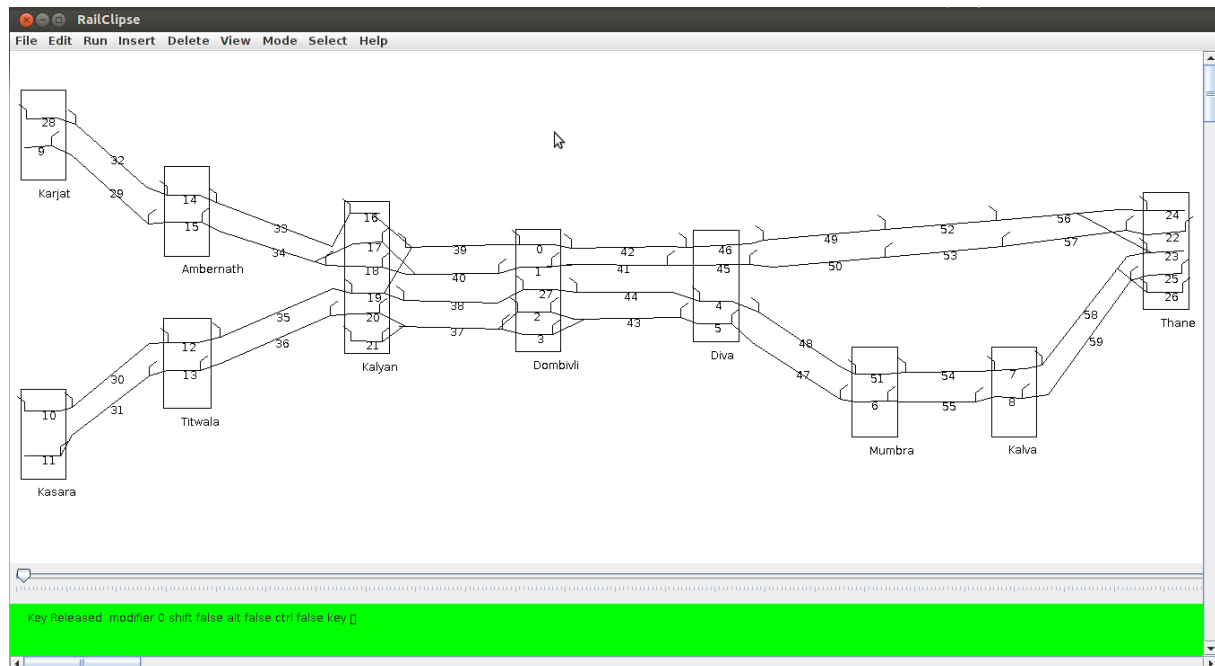


Figure 2.18: All Blocks and Links Completely Added

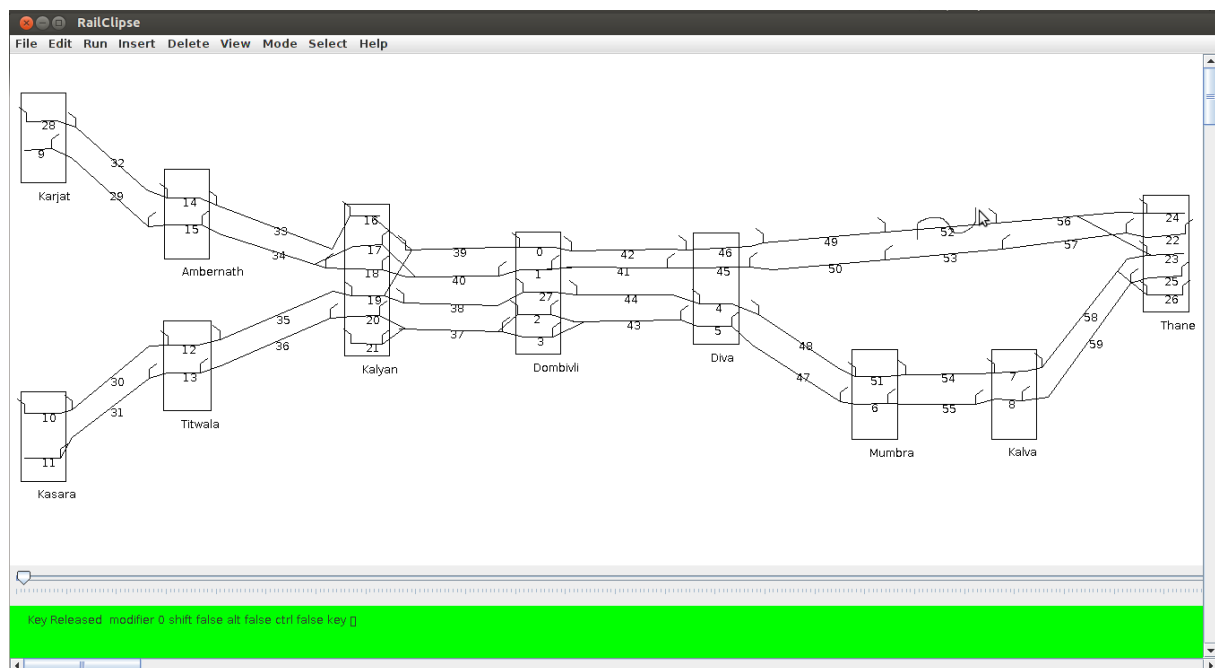


Figure 2.19: Gradient drawn

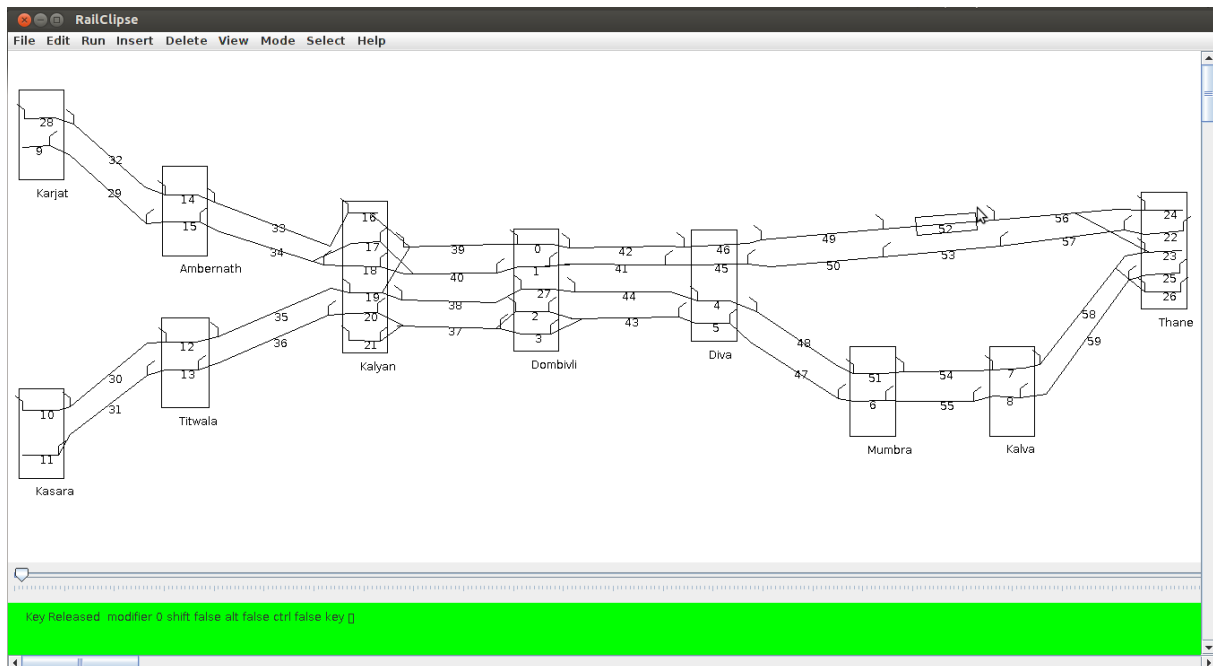


Figure 2.20: Gradient Added

The screenshot shows the "Gradient properties" dialog box. It contains the following fields and options:

- Track Segment Length 10.0
- Relative Start Mile...: 3.0
- Projected Relative Start Milepost 2.957
- Relative End Mile...: 8.3
- Projected Relative End Milepost 8.367
- Gradient Value: "1/10"
- Direction of slope:
 - ☒ UP
 - ☐ DOWN
 - ☐ LEVEL
- Buttons: OK, Cancel

Figure 2.21: Gradient Properties

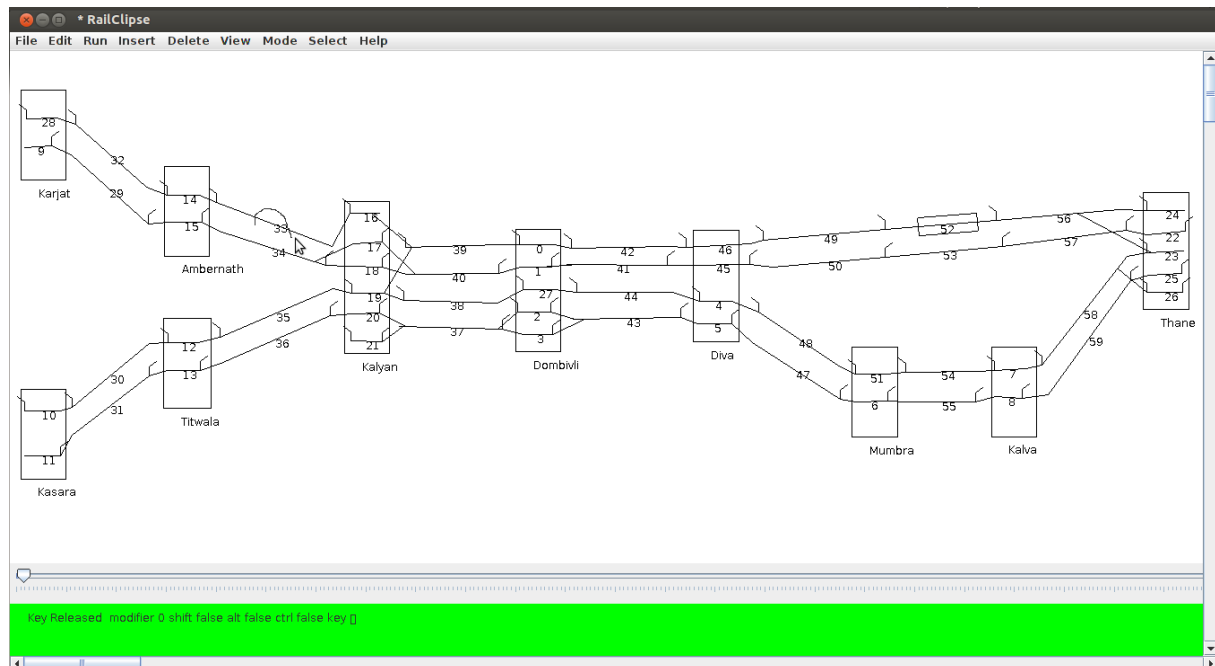


Figure 2.22: Speed restriction drawn

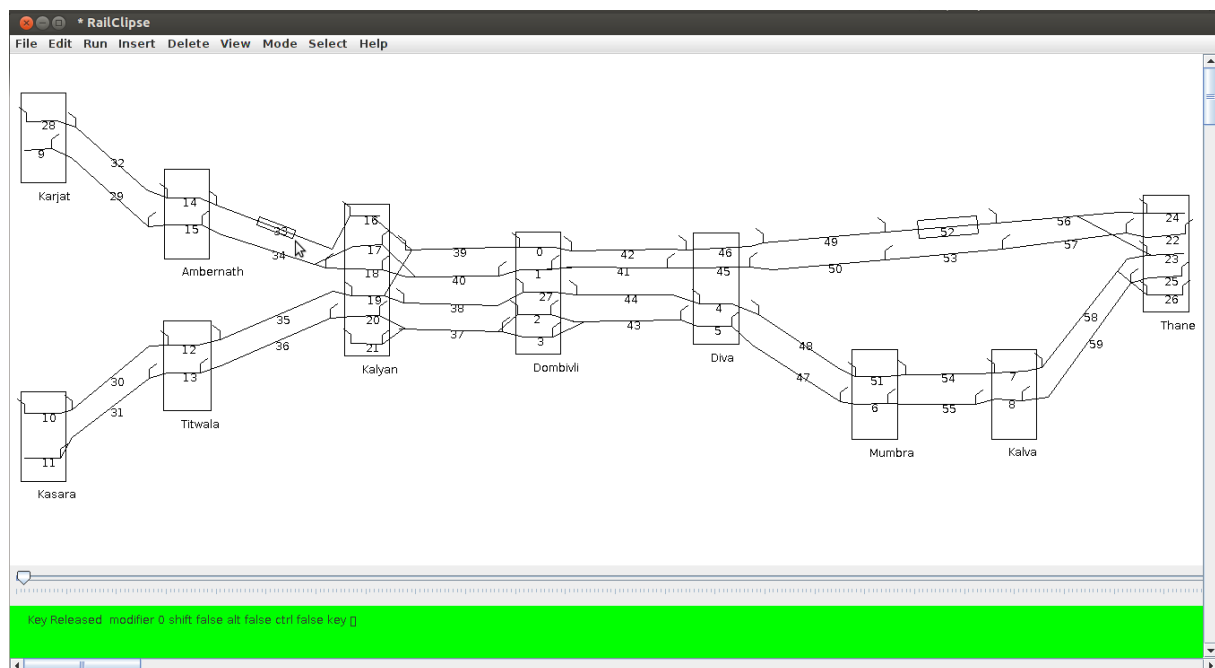
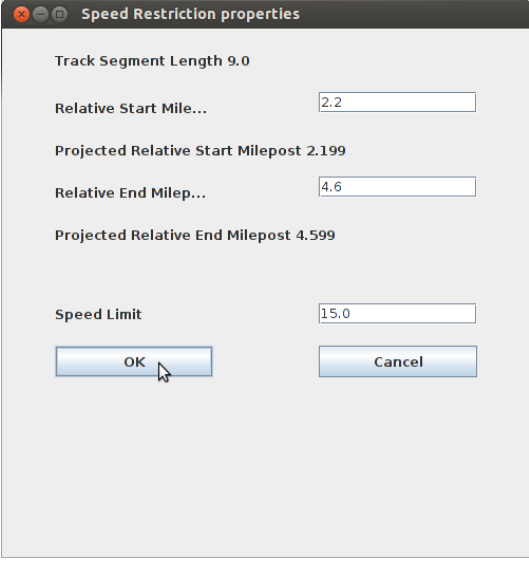


Figure 2.23: Speed restriction Added



Speed Restriction properties

Track Segment Length 9.0

Relative Start Mile...

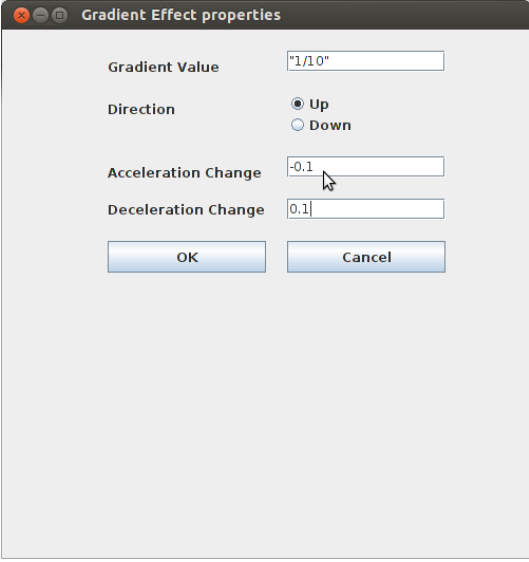
Projected Relative Start Milepost 2.199

Relative End Mile...

Projected Relative End Milepost 4.599

Speed Limit

Figure 2.24: Speed restriction Properties



Gradient Effect properties

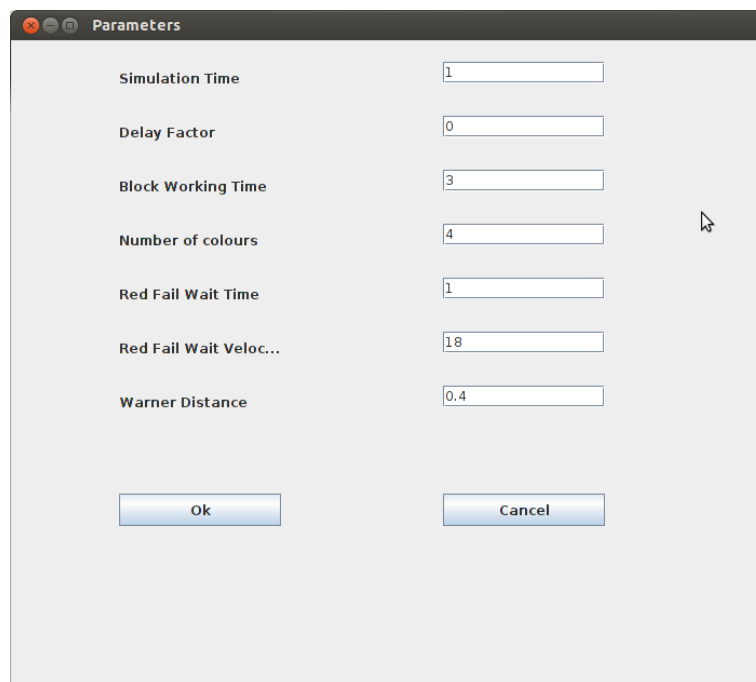
Gradient Value

Direction ☒ Up ☐ Down

Acceleration Change

Deceleration Change

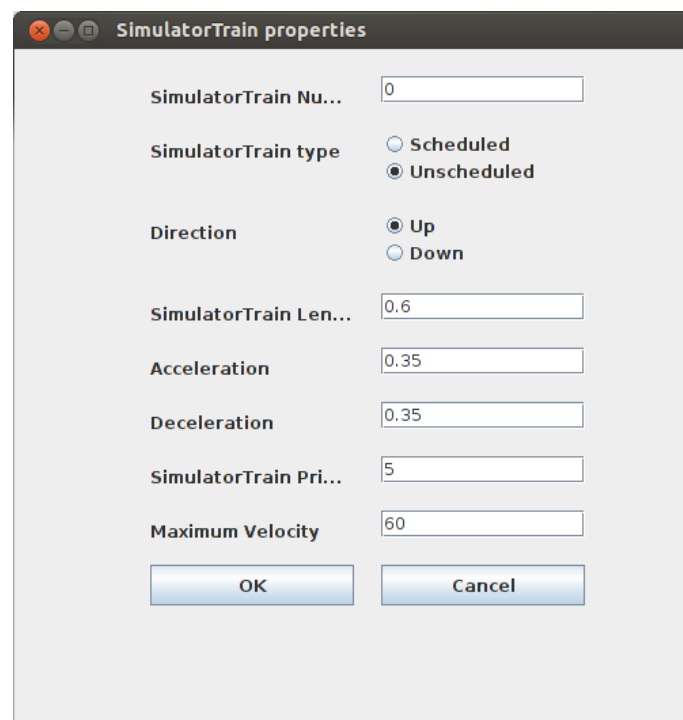
Figure 2.25: Gradient Effect Properties



A dialog box titled "Parameters" with a standard window control bar (close, minimize, maximize). It contains seven rows of parameters, each with a text label on the left and a text input field on the right. The values in the input fields are: 1, 0, 3, 4, 1, 18, and 0.4. At the bottom, there are two buttons: "Ok" and "Cancel".

Parameter	Value
Simulation Time	1
Delay Factor	0
Block Working Time	3
Number of colours	4
Red Fail Wait Time	1
Red Fail Wait Veloc...	18
Warner Distance	0.4

Figure 2.26: Parameters



A dialog box titled "SimulatorTrain properties" with a standard window control bar. It contains seven rows of properties. The first row has a text label and a text input field with the value 0. The next two rows have text labels and radio button groups. The "SimulatorTrain type" group has "Scheduled" and "Unscheduled" options, with "Unscheduled" selected. The "Direction" group has "Up" and "Down" options, with "Up" selected. The remaining three rows have text labels and text input fields with values 0.6, 0.35, and 5. At the bottom, there are two buttons: "OK" and "Cancel".

Property	Value
SimulatorTrain Nu...	0
SimulatorTrain type	<input type="radio"/> Scheduled <input checked="" type="radio"/> Unscheduled
Direction	<input checked="" type="radio"/> Up <input type="radio"/> Down
SimulatorTrain Len...	0.6
Acceleration	0.35
Deceleration	0.35
SimulatorTrain Pri...	5
Maximum Velocity	60

Figure 2.27: Train properties

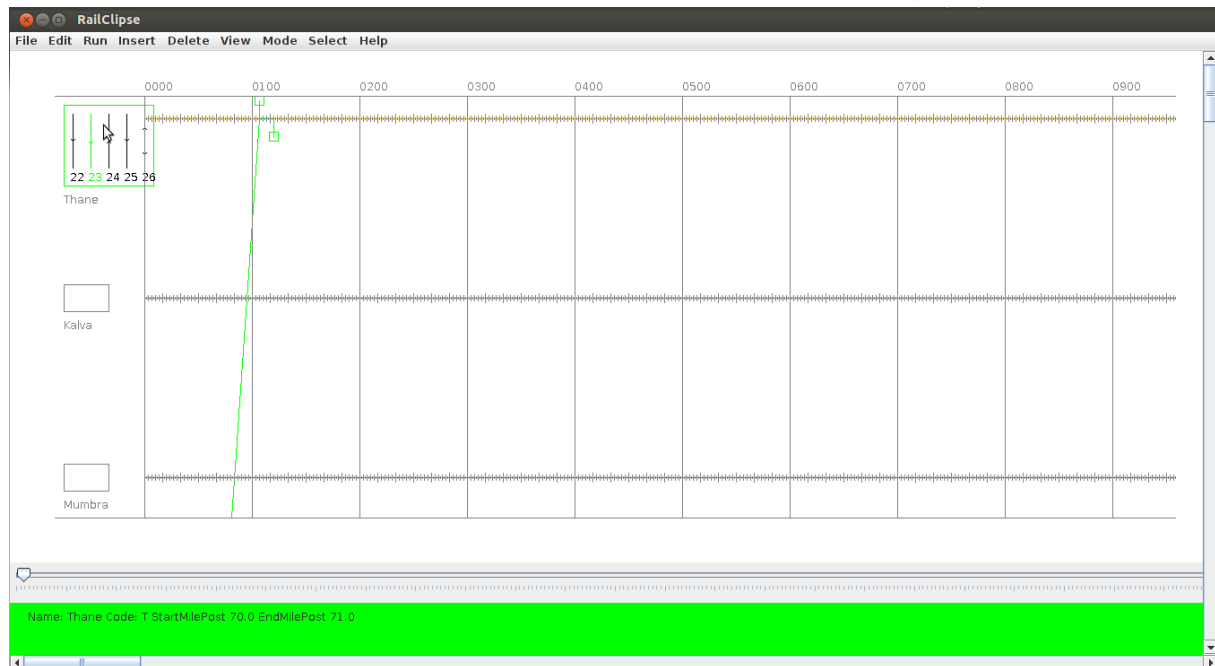


Figure 2.28: TimeTableEntry Changed

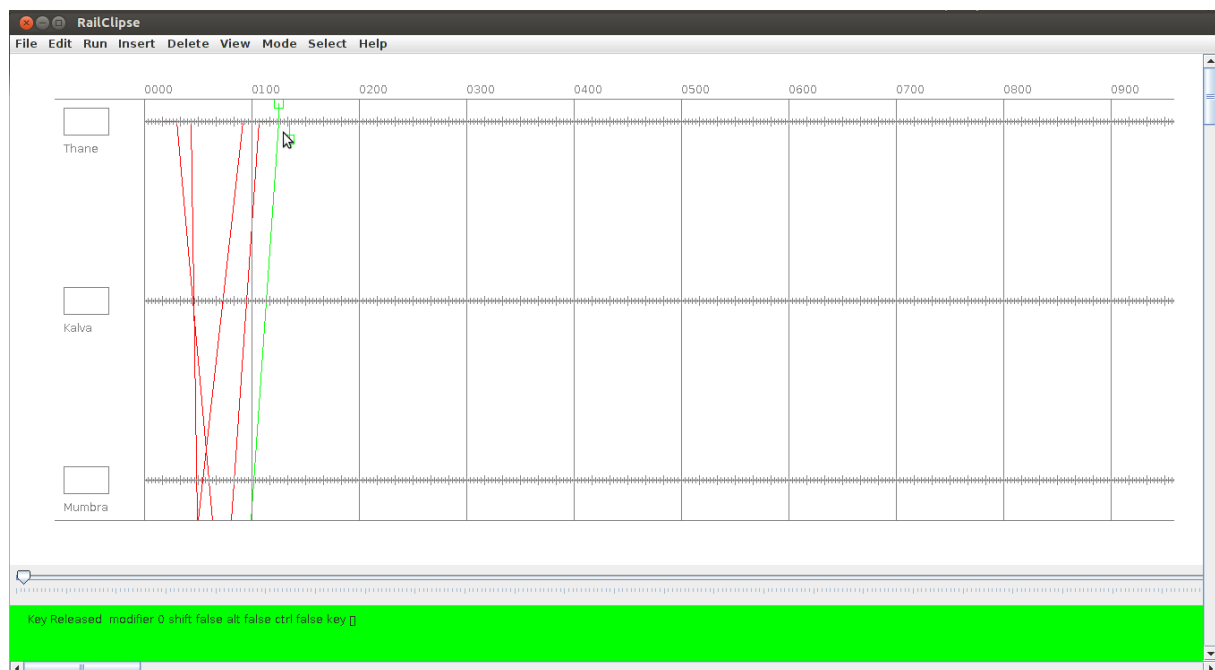


Figure 2.29: TimeTableEntry Changed

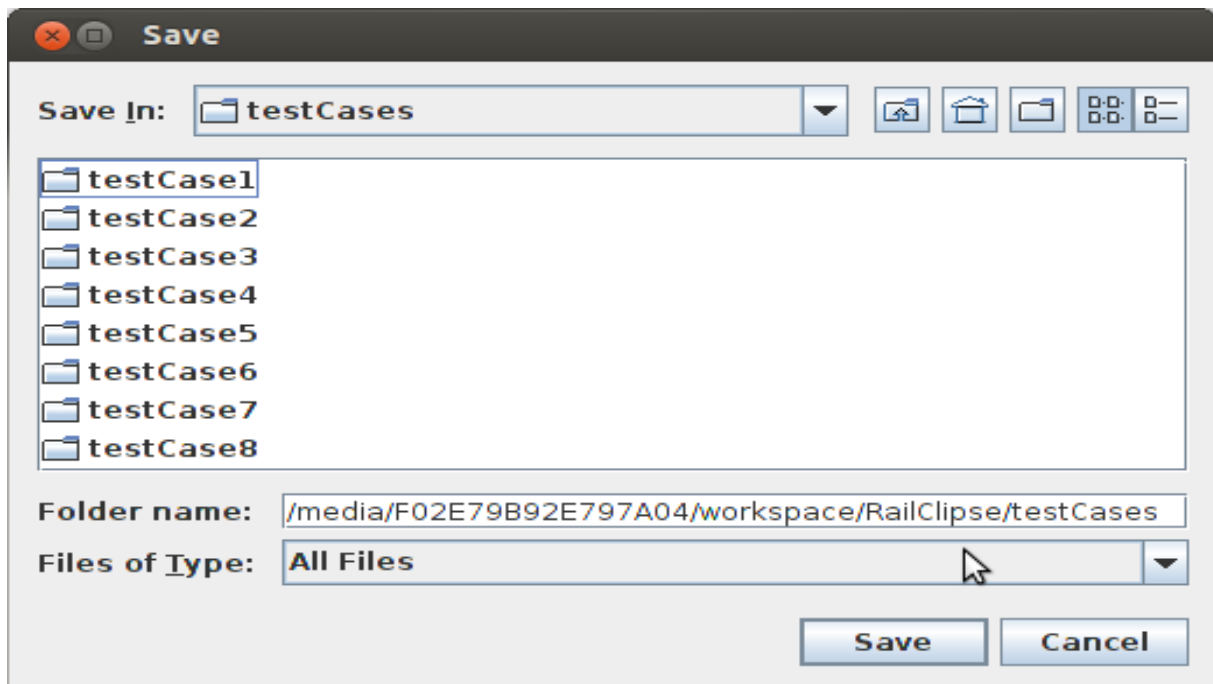


Figure 2.30: Save Project

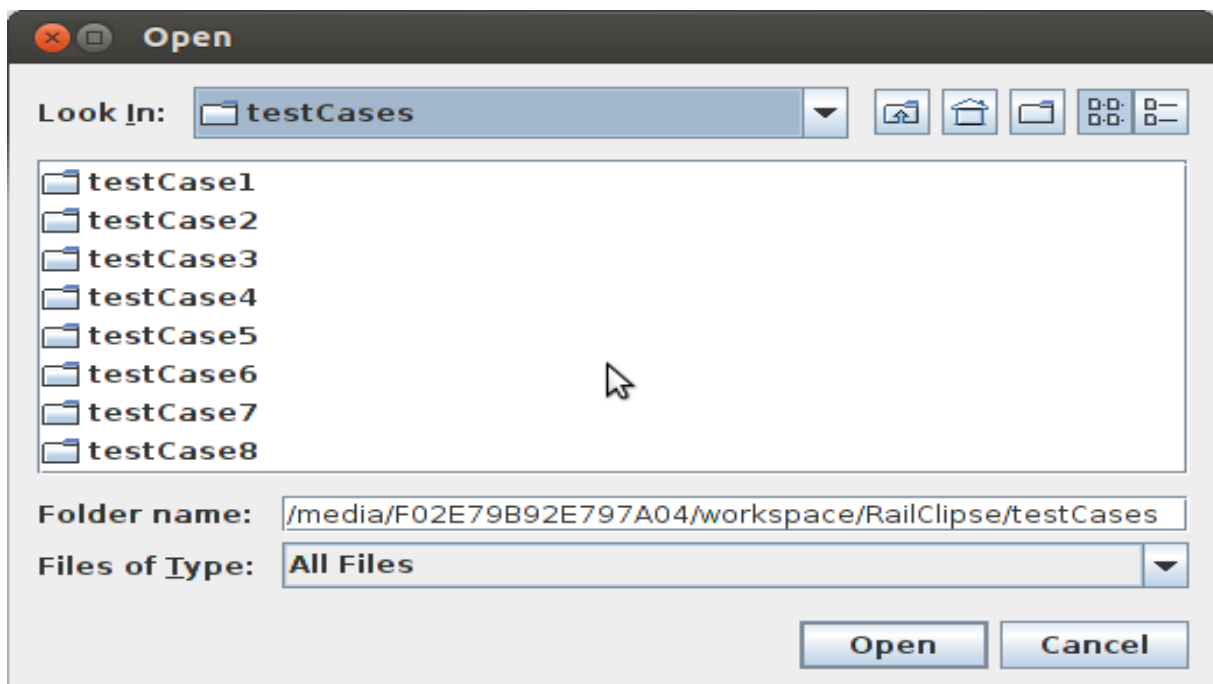


Figure 2.31: Open Project

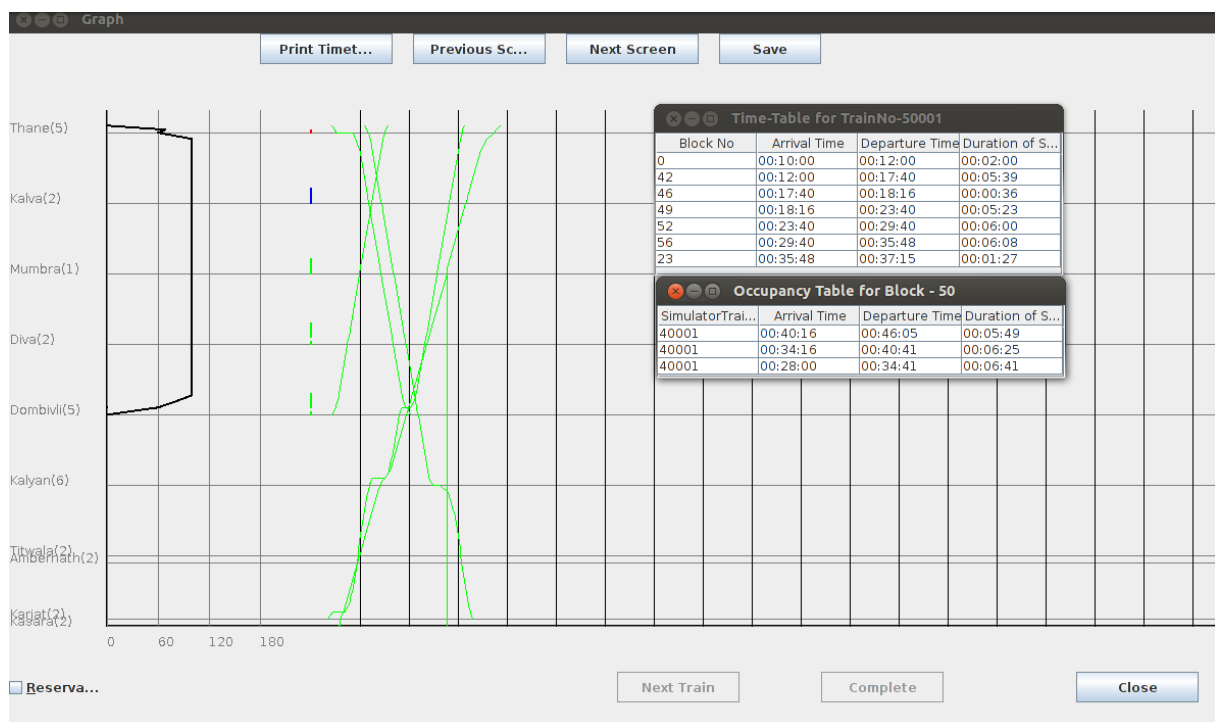


Figure 2.32: Run Project

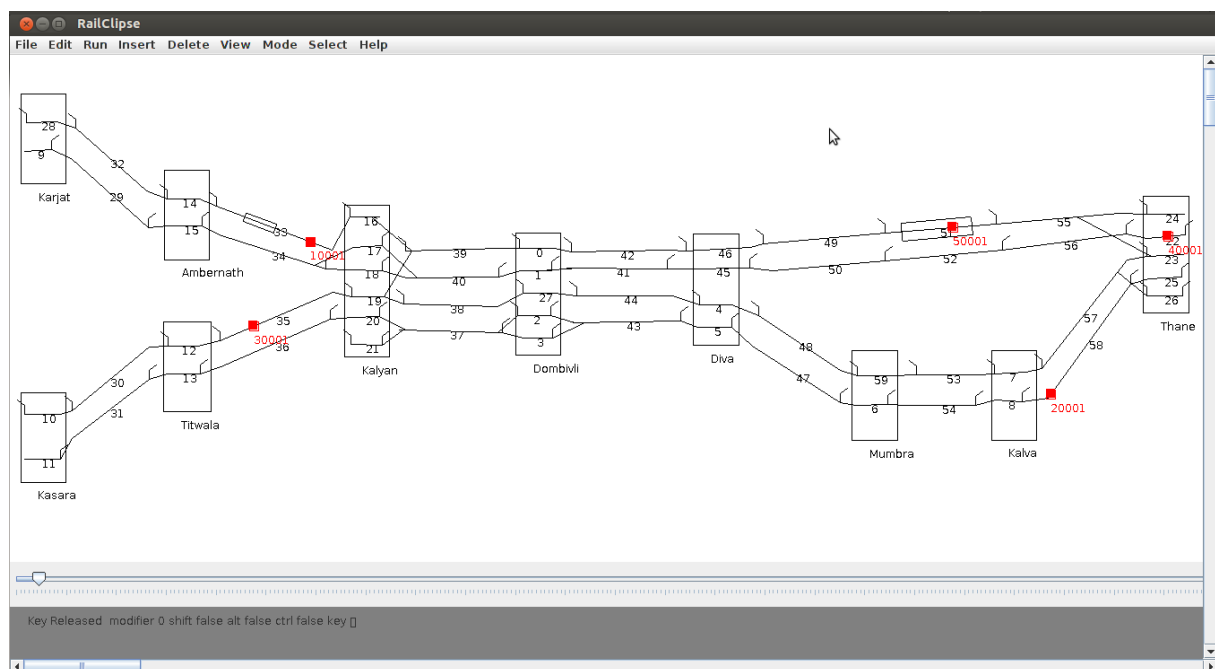


Figure 2.33: Trains moving

Chapter 3

Format of the intermediate files

In this chapter, we describe the formats of files generated by editor. These files will be taken as input by the simulator.

In the following tables, the unit for distance is kilometres. All velocity values are in kilometre/hour. All acceleration and deceleration values are in kilometre/minute-squared.

Relative Start Milepost is the physical distance between the starting of the speed restriction and starting of the block. Relative End Milepost is the physical distance between the ending of the speed restriction and starting of the block. Similarly, Relative Start Coordinate is the distance between the starting point of the block diagram and starting point of the speed restriction diagram, while Relative End Coordinate is the distance between the ending point of the speed restriction diagram and the starting point of the block diagram. Similar, terms are defined for gradient and gradient diagram as well.

Station Name	Station Code	Starting milepost	Ending Milepost	Station Entering Velocity
Kasara	N	0.0	1.0	50.0
Karjat	S	1.0	2.0	50.0
Titwala	TL	10.0	11.0	50.0

Table 3.1: Stations

Station ID	Station Code	X	Y	Width	Height
0	N	36.9	425.8	50.0	100.0
1	S	37.0	93.0	50.0	100.0
2	TL	197.5	347.1	53.1	100.0

Table 3.2: Station Diagrams

Block Number	Direction	Starting milepost	Ending Milepost	Maximum Allowed Speed
29	down	2.0	9.0	100.0
30	up	1.0	10.0	100.0
31	down	1.0	10.0	100.0

Table 3.3: Block

Block Number	x_1	y_1	x_2	y_2
29	68.0	115.0	154.0	192.0
30	69.0	396.0	155.0	325.0
31	69.0	427.0	155.0	361.0

Table 3.4: Block Diagrams

Loop Number	Direction	Loop Type	Station Name	SM	EM	MPS
0	up	ml	Dombivli	30.0	31.0	100.0
1	down	ml	Dombivli	30.0	31.0	100.0
2	common	loop	Dombivli	30.0	31.0	100.0

Note: SM, EM and MPS stand for starting milepost, ending milepost and maximum permissible speed of the loop respectively.

Table 3.5: Loops

Loop Number	Station ID	x_1	y_1	x_2	y_2
0	5	568.0	215.0	607.0	215.0
1	5	569.0	240.0	604.0	240.0
2	5	566.0	290.0	603.0	290.0

Table 3.6: Loop Diagrams

Start Block ID	End Block ID	PBRM	NBRM	Length	MPS	Crossovers
48	59	0.0	0.0	0.0	15.0	"
59	53	0.0	0.0	0.0	15.0	"
19	39	0.0	0.6	0.6	15.0	"40"

Note: MPS stand for maximum permissible speed of the link. PBRM stands for Previous Block Relative Milepost, which is the distance from the end of the previous block at which the link originates. NBRM is the Next Block Relative Milepost, which is the distance from the beginning of the next block at which the link terminates. MAS stands for Maximum Allowed Speed. If a link crosses the blocks with IDs x, y and z, the crossover will be "x;y;z".

Table 3.7: Links

Start Block ID	End Block ID	Crossovers	PBRD	NBRD
48	59	”	0.0	0.0
59	53	”	0.0	0.0
19	39	“40”	0.0	7.0

Note: PBRD stands for Previous Block Relative Distance which is the distance between starting point of the link diagram and the ending point of the previous block diagram of the link. NBRD stands for Next Block Relative Distance which is the distance between the ending point of the link diagram and the starting point of the next block diagram.

Table 3.8: Link Diagrams

Block ID	Speed Limit	Relative Start Milepost	Relative End Milepost
33	15.0	2.1	4.5

Table 3.9: Speed Restrictions

Block ID	Relative Start Coordinate	Relative End Coordinate	Original Block Diagram Length
33	33.7	70.5	137.9

Table 3.10: Speed Restriction Diagrams

vel				
Block ID	Gradient Value(String)	Direction	Relative Start Milepost	Relative End Milepost
51	“1/10”	LEVEL	4.5	7.7

Table 3.11: Gradients

Block ID	Relative Start Coordinate	Relative End Coordinate	Original Block Diagram Length
51	18.6	97.0	124.4

Table 3.12: Gradient Diagrams

Train no	Station ID	Loop ID	Arrival Time	Departure Time
10001	0	0	10	10
20001	2	6	7	7
30001	2	6	16	23
40001	0	1	73	73

Note: A train may have halt at a station but no specific loop be assigned. In that case the loop ID for the timetable entry should be -1.

Table 3.13: Time Table Entry

Train no	Type	No of Timetable Entries	List of TimeTableEntries
10001	UNSCHEDULED	2	0 0 10 10 2 5 54 54
20001	UNSCHEDULED	2	2 6 7 7 0 7 51 51
30001	SCHEDULED	3	2 6 16 23 1 3 48 57 0 7 66 68
40001	SCHEDULED	3	0 1 73 73 1 2 90 98 2 5 117 119

Note: The train number attribute of all time table entries related to one train is written only once in the column of train numbers.

Table 3.14: Train Diagrams

Train no	Direction	Arrival time	L	A	D	P	MS	Start Loop ID	End Loop ID
10001	up	0010	0.6	0.35	0.35	5	60	0	5
20001	down	0007	0.6	0.35	0.35	5	60	6	7

Note: Letters L, A, D, P, and MS stand for Length, Acceleration, Deceleration, priority and Maximum Speed of the train.

Table 3.15: Unscheduled Trains

Train no	Direction	L	A	D	P	MS	Operating Days	List of TimeTableEntries
30001	down	0.6	0.5	1.0	5	100	all	Refer 3 & 3
40001	up	0.6	0.5	1.0	5	100	all	Refer 3 & 3

Note: Letters L, A, D, P, and MS stand for Length, Acceleration, Deceleration, priority and Maximum Speed of the train

Table 3.16: Scheduled Trains

Parameter Name	Parameter Value
Simulation Time	1
Delay Factor	0
Block Working Time	3
No of signal aspect	4
Red Fail Wait Time	1
Red Fail Velocity	18
Warner Distance	0.4

Table 3.17: Simulation parameters

Chapter 4

Implementation of SketchRail

In this chapter, the code structure of SketchRail is explained to a certain extent. This chapter intends to give an overview of the classes, interfaces and logical sections of the code and what roles they play in SketchRail .

A graphical tool can be built in any of the object-oriented programming languages like C++, Java, Python, etc. My choice for using Java for this particular project was primarily because the simulator code was written in Java. A graphical tool requires the following list of functionalities that are present as in-built utilities in Java:

- Frames: A window for users to interact with the code.
- Panel: A display object where standard figures and shapes can be drawn and displayed. It can also display strings. It can have different stroke colours, fill colours for the various shapes, etc.
- Event listeners: Specifically, mouse listeners, mouse motion listeners and keyboard listeners are required so that users can send commands to the software. Mouse listeners provide a rich set of action listeners for events like mouse-pressed, mouse-moved, mouse-dragged, mouse-click-count, etc. Keyboard listeners also act as modifiers to these mouse events. These are part of `java.awt.*` libraries.
- Some other additional features like dialog boxes, labels, text fields, radio buttons, submit buttons etc. which are quite essential to ease the process of input. These are part of `java.swing.*` libraries.

There are currently two modes in which SketchRail operates, namely the section-view-mode, which is used to represent information regarding all the stations, blocks, links, gradients and speed restrictions; and the train-view-mode which is used to represent information regarding the trains, their reference schedules, desired loops at stations, etc.

In the train-view-mode, SketchRail has a stations vs time plot. A plot joining the (station, time) coordinates is used to represent a desired schedule for some train.

4.1 Building blocks of SketchRail code

In this section, we describe what are the base abstract classes and what interfaces do they implement. We describe their relations with each other, what attributes they have and what functions do they implement.

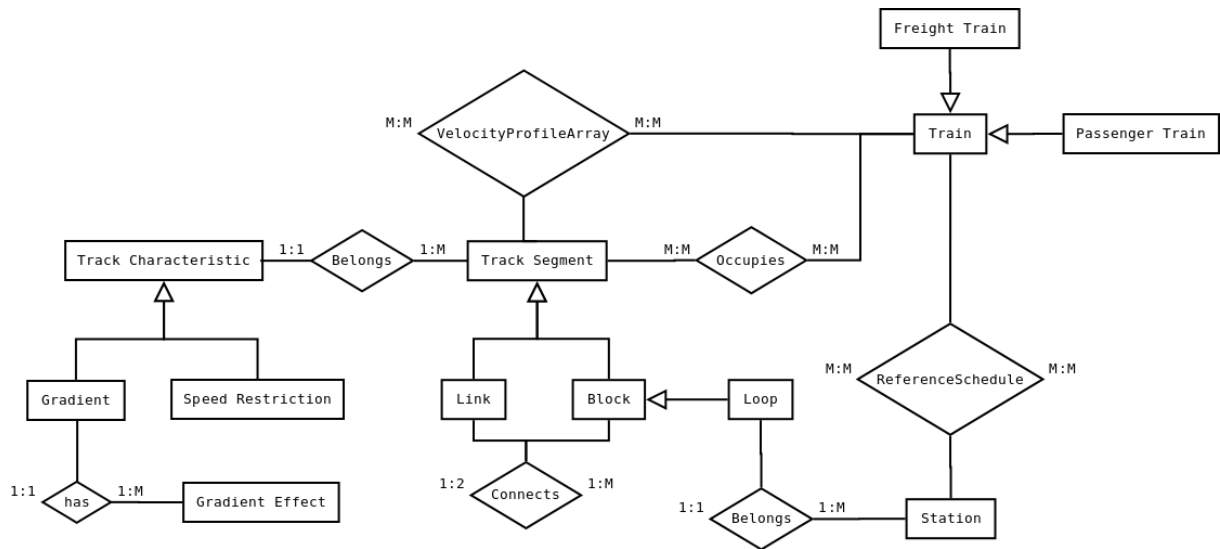


Figure 4.1: Entities and their relationships

The following elements are the building blocks of the code:

- Interfaces:
 - ListInterface
 - DrawingInterface
 - ManagerInterface
- Abstract classes:
 - Entity
 - Diagram
 - Manager
 - InputDialog
- Global variables:
 - DiagramList
 - diagramHighlighted
 - diagramEdited

4.1.1 Interfaces

ListInterface

```

public interface ListInterface<SectionDiagram> {

    public boolean addToList(SectionDiagram sectionDiagram);

    public void sortMe();

    public boolean deleteFromList(SectionDiagram sectionDiagram);
  
```

```

    public ArrayList<SectionDiagram> getList();

    public int size();
}

```

DrawingInterface

There are two views in SketchRail . The section-input-view and the train-input-view. These functions are called whenever an event occurs like mouse moved, mouse pressed etc.

```

public interface DrawingInterface {
    public void drawMe(Graphics2D g2);
    public void drawMeInTrainView(Graphics2D g2);
    public void drawMeInSectionView(Graphics2D g2);
}

```

ManagerInterface

There are four major operations to be performed by a manager namely addition, deletion, updation and find. The actions to be performed for the first three operations depends upon the mouse events like mouse-moved, mouse-pressed, mouse-double-clicked, mouse-dragged or keyboard events like pressing the modifiers with arrow keys etc.

```

public interface ManagerInterface<SectionDiagram> {
    public void insertToolMouseMovedTask(Rectangle2D net ,
        Point2D adjustedPoint , MouseEvent me);

    public void selectToolMouseMovedTask(Rectangle2D net ,
        Point2D adjustedPoint , MouseEvent me);

    public void deleteToolMouseMovedTask(Rectangle2D net ,
        Point2D adjustedPoint , MouseEvent me);

    public void insertToolMousePressedTask(Rectangle2D net ,
        Point2D adjustedPoint , MouseEvent me);

    public void selectToolMousePressedTask(Rectangle2D net ,
        Point2D adjustedPoint , MouseEvent me);

    public void deleteToolMousePressedTask(Rectangle2D net ,
        Point2D adjustedPoint , MouseEvent me);

    public void mousePressedTask(Rectangle2D net ,
        Point2D adjustedPoint , MouseEvent me);

    public void mouseMovedTask(Rectangle2D net ,
        Point2D adjustedPoint , MouseEvent me);

    public void mouseDraggedTask(Rectangle2D net , Point2D adjustedPoint ,
        MouseEvent me);

    public SectionDiagram findDiagram(Rectangle2D net , Point2D adjustedPoint ,
        MouseEvent me);
}

```

```

public void drawMe(Graphics2D g2);

public void writeAll() throws IOException;

public void write(SectionDiagram sectionDiagram);

public void readAll() throws IOException;

public SectionDiagram getDiagramFromId(int id);

public void showInputDialog(SectionDiagram sectionDiagram);
}

```

4.1.2 Abstract Classes

Entity

Entity is an abstract class which is the super class of the classes used to represent the physical items like stations, track segments, trains etc. The subclasses contain information regarding location, physical properties and constraints, ids etc. For example a station entity consists of its name, starting and ending mileposts, maximum velocity with which the trains may enter the stations, etc. Derived classes are Station, Signal, TrackSegment, Train, GradientEffect and TrackCharacteristic (Gradient, Speed Restriction).

Diagram

A diagram is used to graphically represent some entity on the panel. For example the class *StationDiagram* which represents the entity *station* on the panel is derived from this super class i.e. “Diagram”. This class contains information regarding the type of entity it represents, its coordinates (visibleRects, actualRects), colour of diagram, whether it is being highlighted or dragged or edited, etc. It implements the *DrawingInterface*. Derived classes are *StationDiagram*, *GradientEffectsDiagram*, *TrackSegmentDiagram*, *TrackCharacteristicDiagram*, *SignalDiagram* and *TrainDiagram*.

```

public abstract class SectionDiagram implements DrawingInterface {
    protected Manager manager = null;
    protected Entity entity;

    protected int id = -1;
    protected Rectangle2D.Double[] visibleRects;
    protected Rectangle2D.Double[] actualRects;

    protected boolean showRects = false;
    protected int selectedIndex = -1;
    protected boolean dragging = false;
    protected boolean highlighting = false;
    protected boolean editing = false;

    protected Color color = null;
}

```

Manager

A manager is an abstract class which handles information regarding a set of diagrams of the same type. For example, a station-manager which is sub-class of *Manager* handles information regarding all the stations. Similarly, a gradient-manager handles information about all the gradients and so on and so forth. While adding a diagram, its manager attribute is appropriately initialized. Every manager is instantiated exactly once. A manager implements the *ListInterface* and *ManagerInterface*. Derived classes are *TrackSegmentManager*, *TrackCharacteristicManager*, *StationManager* and *TrainManager*.

In addition to addition/deletion/updation of diagrams, a manager has following sub-roles to perform:

- Maintain information regarding consistency that all the diagrams and their entities under the manager must adhere to. For example, a loop must have starting and ending mileposts which lie in between the starting and ending mileposts of its station. Otherwise, the loop diagram will be shown as red in colour to highlight the error on the panel.
- Assigning integer ids to diagram: While saving, all the diagrams must be assigned a unique identity. This is essential for generating input files to the simulator. It is primarily for properly setting the foreign keys of certain entities by using the id information. For example, when a speed restriction is stored into the file, it is essential to store which block does it belong to.
- Get List: It is important to mention that manager does not maintain the list of diagrams. Whenever required, the manager iterates through the global list of diagrams and retrieve the appropriate ones. For example, when station-manager calls *getList*, it retrieves the list of station diagrams from the global diagram list.
- Finding diagram by id: The manager simply iterates through the list obtained by *getList*, and returns the object having the given id (not necessarily the integer Id). A station-manager can return a station if its station name is provided.
- writeAll: Whenever save function is called in *SketchRail*, a manager stores the information in appropriate files. For example, on calling save function, a block manager has to call retrieve the list of blocks by calling *getList* and their attributes in appropriate format in the block file.
- readAll: When an open project function is called, a manager opens the appropriate file(s), read the contents, and add corresponding diagrams in the global diagram list. For example, open project action causes train manager to open *trainDiagram* file, *scheduled trains* file and *unscheduled trains* file and add all the train diagrams in the global diagram list.

InputDialog

This is the base class of all input dialog boxes. Following is the skeleton of the class *InputDialog*. Of these, the logic of determining the consistency and correctness of the input given goes in the function *processInput*.

```
public abstract class InputDialog extends JFrame {
    public SectionDiagram sectionDiagram;
    public Manager manager;

    protected abstract boolean processInput();
}
```

```
protected abstract void setAttributes();  
  
protected abstract void setFieldsFromAttributes();
```

4.1.3 Global Variables

DiagramList

DiagramList is a global datastructure where all the diagrams are maintained as part of a list. It is updated on events like addition/deletion/updation of certain entities or diagrams. For example, when a block is input or removed from the panel it is accordingly added or deleted from this global list. Or when the positions of station diagrams is changed on the panel, the order in which they appear in the list is changed. This list consists of objects ordered in increasing order of their sizes. This is important because when we hover the mouse over a group of diagrams and we have to delete only one of them, the smallest diagram is least likely to get picked up. We wouldn't want to accidentally delete a different diagram because of its large size. Hence, the order.

Chapter 5

Analysis of 3-line section

In this chapter, we explain the use of existing simulator to determine operational strategies for 3 line sections. My specific contribution in this work has been to conduct experiments and provide results to support the hypothesis described more in the chapter.

Problem Definition

Given a 3 line railway section, with one dedicated up line, one dedicated down line and a bi-directional line, determine a strategy for effective use of the third line for better performance of the section across various performance measures.

In [2], a simulation-based approach to determine a strategy for the effective use of the third line in a 3-line railway section is described. In general, the third line can be in the middle of the other two lines. The objective of this study was to determine in what direction the third line be used given the traffic in both directions and the network configuration of the section. In particular, the performance of Tiruvallur-Arakkonam section (abbreviated as TRL-AJJ), in Southern Railways, was analyzed. The measure used to quantify the section performance is weighted average traversal time. Two strategies namely, fixed time interval reservation and variable time interval reservation based on traffic indicators were analyzed and compared.

5.1 Description of Section

3-line sections can have either the up line, down line or the common line in the middle. TRL-AJJ section has up line in the middle. There are two intermediate stations Kadambattur and Tiruvalangadu (abbreviated as KBT and TI respectively). There are other intermediate stations where local trains halt. But, they have only as many loops as the railway lines, namely 3. At the stations KBT and TI, there are 5 loop lines each. Hence, the trains can change tracks or overtake at these stations.

It is observed that during some intervals of time the section traffic has a directionality. Intuitively, the direction of third line should be in the direction in which there is more train traffic. So, experiments were conducted which gave some weight to this intuition.

5.2 Strategies used

A set of parameters to measure the section performance were defined. N_u and N_d are the measures of traffic in up and down direction respectively. They capture the intensity of traffic and usage of resources over a certain time period.

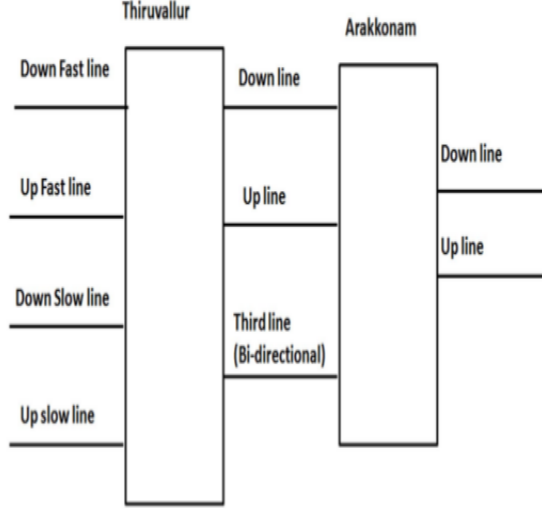


Figure 5.1: Line arrangement of TRL-AJJ section

5.2.1 Traffic indicators

The traffic indicators N_u and N_d are computed as follows. Determine a time horizon T (e.g. two hours) from current time.

- *Definition 1:* N_u is the weighted sum of trains in the up direction that are in the section or will be in the section during the time interval $[0, T]$. The current time is 0.
- *Definition 2:* N_d is the weighted sum of trains in the down direction that are in the section or will be in the section during the time interval $[0, T]$. The current time is 0.
- *Definition 3:* K is the measure of threshold difference between the up-traffic and down-traffic that would suggest sufficient directional dominance in the down direction.
- *Definition 4:* L is the measure of threshold difference between the up-traffic and down-traffic that would suggest sufficient directional dominance in the up direction.

We wanted to prove the following claims by simulation of the TRL-AJJ section with the help of the simulator.

- If the third line is currently operating in up direction, change over to down IF $N_d - N_u = K$ (i.e. continue in up direction if $N_d - N_u < K$)
- If the third line is currently operating in down direction, change over to up IF $N_u - N_d = L$ (i.e. continue in down direction if $N_u - N_d < L$).

In other words, move to a different regime if there is sufficient directional dominance of traffic. The parameters K and L and the method of computation of the traffic indicators N_u and N_d will be described below.

Let the up trains considered for scheduling be $u_1, u_2; \dots; u_m$ with priorities $p_{u_1}; p_{u_2}; \dots; p_{u_m}$. Also, let the down trains considered for scheduling be $d_1; d_2; \dots; d_n$ with priorities $p_{d_1}; p_{d_2}; \dots; p_{d_n}$.

- The traffic measure N_u in the up direction is given by:

$$N_u = \frac{p_{u_1} + p_{u_2} + \dots + p_{u_m}}{m} \quad (5.1)$$

- The traffic measure N_d in the down direction is given by:

$$N_d = \frac{p_{d_1} + p_{d_2} + \dots + p_{d_n}}{n} \quad (5.2)$$

Now, let us define the traversal time of a train. A train once into the section of interest can either halt at an intermediate station which is the train's destination station or exit the section completely. Therefore, the traversal time of the train is defined either to be the time taken for train to halt at its destination station in the section (if any) or the time taken by train to exit the section completely adhering to the schedule specified by the simulator, since its arrival into the section. For e.g. if a train tr enters the station at time t_a and halts at its destination station or exits the section at time t_e then the traversal time t_{tr} for train is:

$$t_{tr} = t_e - t_a \quad (5.3)$$

We can further define a weighted average traversal time for all the trains considering the specific time window. Let's say that $tr_{u_1}; tr_{u_2}; \dots; tr_{u_m}$ be the traversal times for the up trains and $tr_{d_1}; tr_{d_2}; \dots; tr_{d_n}$ be the traversal times for the down trains. The priorities of the trains are mentioned earlier. Then the weighted average traversal time for these trains is defined as:

$$WATT = \frac{\sum_{i=1}^m p_{u_i} * tr_{u_i} + \sum_{i=1}^n p_{d_i} * tr_{d_i}}{m + n} \quad (5.4)$$

Given a general scenario, the average traversal time when the 3^{rd} line is used only in the up direction and when it is used only in the down direction are bound to be different. Let us call these times as $WATT_u$ and $WATT_d$ respectively. On the basis of the average traversal times $WATT_u$ and $WATT_d$ we compute the values of K and L as shown below. These definitions of K and L are specific only for 3-line sections which have the up line in the middle.

There are two major cases based upon the values of N_u and N_d .

- $N_u \geq N_d$: In this case, we claim that $WATT_u > WATT_d$ as shall be shown in the results of the experiment later.
- $N_u < N_d$: In this case, there are two sub cases:
 - $WATT_u > WATT_d$: This would be the case when N_u is not much less than N_d .
 - $WATT_u \leq WATT_d$: This would be the case when N_u is much less than N_d .

Then, let

$$K' = N_d - N_u \quad (5.5)$$

The smallest such value of K' is K . Mathematically defined as

$$K = \{min N_d - N_u | WATT_u \leq WATT_d\} \quad (5.6)$$

L is defined similarly.

When the difference between the traffic measures $N_d - N_u$ becomes as greater than or equal to K we will want to change the direction of the 3^{rd} from up to down. However, once the traffic sets in the section and direction of 3^{rd} line is in down direction, if N_u which is initially less than N_d , gradually becomes greater than N_d , because of existing traffic changing the direction of 3^{rd} line may not reduce the $WATT$. Hence, we change the direction of 3^{rd} line only when the difference $N_u - N_d$ becomes greater than the threshold L . i.e. when $N_u - N_d > L$, change the direction of the line from down direction to up direction to reduce $WATT$.

5.2.2 Fixed time strategy

Varying the headway of freight trains

The indicative values of K and L (and T) are suggested through a simulation exercise on the TRL - AJJ section. The time window for the experiment was taken to be one hour. We scheduled 4 down passenger trains each with priority 4 for the experiment. We vary the parameters u (upHeadway in minutes) and d (downHeadway in minutes) for the freight trains which are of priority 8 each. The priorities are defined on scale 1-10. We normalize the priorities by dividing it by 11 which is $maxPriority + 1$. The freight trains start coming into the section since the beginning of the time window at a regular interval of headways. That is the first freight trains in both directions reach the beginning of the section at 0000 hrs. Since the time window is of duration 1 hour, the number of trains fired in up direction and down direction are given by n_u and n_d as follows:

$$n_u = \frac{60}{upHeadway} \quad (5.7)$$

$$n_d = \frac{60}{downHeadway} \quad (5.8)$$

Further the weighted traffic in up direction and down direction will be as follows:

$$N_u = n_u * \frac{8}{11} \quad (5.9)$$

$$N_d = \frac{n_d * 8 + 4 * 4}{11} \quad (5.10)$$

Varying the headway of freight trains

The approach used in the experiments related to table 5.1 is that, if for certain period of time we find the up traffic heavier than the down traffic above a threshold, we dynamically change the direction of the third line from down to up direction (if required). That is we keep monitoring the values for the traffic and determine the direction of the third line.

For the static allocation of direction for third line, we do a pre-analysis of traffic and fix a schedule for the third line. Say for example, for half an hour the third line is in up direction followed by half an hour for which the third line is in down direction, followed by one hour for which third line can be used in both the directions.

Unscheduled trains are fired for an interval of 1 hour. We can see that the direction of the third line is changed when the difference between the traffic measures $N_d - N_u$ becomes as greater than or equal to K we will want to change the direction of the 3rd from up to down.

In the table 5.1 we have not specified the results for all the up headways and down headways. This is because, as the down head way increases, or the down traffic decreases, the usage of third line changes from down direction to up direction. However, if we reach a stage where $WATT_u$ and $WATT_d$ both are equal, that means even if we further reduce the down traffic, the $WATT$ values will be the same whether the third line is used in the up direction or the down direction. The reason behind this is that the traffic in both the directions is so less in these cases that the third line is not at all used.

Scheduling trains as per the given time table

The time slots considered for these experiments is 6 hours i.e. a set of 4 such experiments over a period of 24 hours. The results of the same have been shown in table 5.2. The 3rd track is reserved in the up and down direction for all the for sets, $WATT_u$ and $WATT_d$ are calculated respectively.

up Headway	down Headway	N_u	N_d	$WATT_u$	$WATT_d$	Direction of 3 rd line
5	5	3.270	5.810	17.403	19.482	up
5	6	3.270	5.273	16.132	18.366	up
6	5	2.727	5.810	17.570	18.442	up
7	5	2.455	5.810	17.948	17.820	down
8	5	2.182	5.810	18.023	17.440	down
9	5	1.909	5.810	18.075	17.221	down
9	8	1.909	4.727	15.000	15.011	up
10	13	1.636	3.909	13.201	13.123	down
10	14	1.636	3.909	13.023	12.921	down
11	7	1.636	5.000	15.590	15.385	down
12	10	1.360	4.180	14.123	14.027	down
13	12	1.360	3.909	13.626	13.508	down
14	13	1.360	3.909	13.428	13.147	down
15	16	1.091	3.636	13.367	13.367	-
16	15	1.091	3.636	13.510	13.266	down

Table 5.1: Fixed time strategy results

Time slots	N_u	N_d	$WATT_u$	$WATT_d$	Direction of 3 rd line
0-6	7.727	3.818	13.032	16.799	up
6-12	5.727	4.182	8.78	17.671	up
12-18	4.636	8.727	11.801	14.127	up
18-24	5.727	8.818	8.189	11.839	up

Table 5.2: Fixed strategy results for real data

5.2.3 Variable time strategy

In this section the time slots of 6 hours are considered and the direction of the 3rd line is calculated based on the traffic intensity for each hour. In this case (while calculating $WATT_u$ the direction in the respective time slots are:

- {0-1,1-2,2-3,3-4,4-5,5-6} = {down, down, up, up, up, down}
- {6-7, 7-8, 8-9, 9-10, 10-11, 11-12} = {down, down, up, up, up, up}
- {12-13, 13-14, 14-15, 15-16, 16-17, 17-18} = {down, down, down, down, down, down}
- {18-19, 19-20, 20-21, 21-22, 22-23, 23-24} = {down, up, up, up, up, down}

The directions are taken opposite while calculating $WATT_d$. The results are shown in table 3.

5.3 Conclusion

The operational strategy for utilizing a resource like the third line in heavy traffic sections is not obvious. It depends on a number of factors like the location of the third line with respect

Time slots	N_u	N_d	$WATT_u$	$WATT_d$	Direction of 3 rd line (per hour)
0-6	7.727	3.818	13.712	13.712	down, down, up, up, up, down
6-12	5.727	4.182	10.760	10.760	down, down, up, up, up, up
12-18	4.636	8.727	14.172	14.172	down, down, down, down, down, down
18-24	5.727	8.818	10.391	10.391	down, up, up, up, up, down

Table 5.3: Variable strategy results for real data

to the other two, the location of crossover points, the traffic density, the traffic mix and finally the performance measures of relevance to the rail operator. Simulation provides a good means for evaluating different strategies.

In this paper, we reviewed the performance measures like, traffic intensity and weighted average traversal time which are used as the decision making parameters for the 3 line railway section. Simulation also plays a major role in railway scheduling which has been done using the simulator developed by IIT Bombay

We experiment with the simple one of fixed time regimes and compare it with more complex strategies like variable time strategy involving some system state measures. More sophisticated control strategies using long run simulations can also be proposed and evaluated by other analytical techniques in future. The ones we have suggested have the advantage of being implementable with minimum additional infrastructure and training of personnel involved in the decision.

Chapter 6

Velocity Profile Array Generation

In this chapter, we describe how to generate a velocity vs distance plot given the track characteristics of the path of a train and the train kinetics. We begin by describing the classes and the relevant attributes. Then we show how to generate a velocity vs distance plot on tiny track segment. We show what kind of velocity profiles can result on a tiny track segment for a particular train. We then show how to generate tiny track segments, given a track segment and its list of speed restrictions and gradients. Then we show how the velocity profile array of a train looks on a list of consecutive tiny track segments. We complete the chapter by showing how to prepare the list of relevant tiny track segments and how to generate velocity vs distance plot over the entire block.

A train's kinetics are described by its maximum acceleration $A > 0$, maximum deceleration $D > 0$ and maximum velocity $V_{max} > 0$,

```
public class Train {
    public double A;
    public double D;
    public double V_max;
}
```

Relative start mileposts are the distances of starting mileposts of an entity from the start of the track segment it belongs to. Relative ending mileposts are the distances of ending mileposts of the entity from the start of the track segment it belongs to. These are the attributes of an abstract class “TrackPart” which means that it is a part of a track.

```
public abstract class TrackPart {
    double r_s;
    double r_e;
}
```

A tiny track segment has 4 additional attributes, V_{max} , dir , a_c , d_c which describes respectively, the maximum speed allowed or possible for the train, the direction of the gradient and the maximum changes it will have on the acceleration and deceleration of the train.

```
public class TinyTrackSegment extends TrackPart {
    public double V_max;
    public String dir;
    public double a_c;
    public double d_c;
}
```

```

public class TrackSegment {
    public double L;
    public ArrayList<SpeedRestriction> S;
    public ArrayList<Gradient> G;
    public double V_max;
    public ArrayList<TinyTrackSegment> T;
}

```

A velocity profile is as described below. where l is the length of the profile. a is the constant acceleration with which the train is travelling and v is the starting velocity of the profile.

```

public class VelocityProfile {
    public double l;
    public double a;
    public double v;
}

```

```

public class VelocityProfileArray extends ArrayList<VelocityProfile> {
    TrackSegment ts;
}

```

A gradient has 3 additional attributes namely the dir , a_c and d_c which describe the direction of the gradient, the maximum change caused in acceleration and maximum change caused in deceleration respectively.

```

public class Gradient extends TrackPart {
    public String dir;
    public double a_c;
    public double d_c;
}

```

A speed restriction has an attribute V_{max} which is the maximum speed allowed.

```

public class SpeedRestriction extends TrackPart {
    public double v_max;
}

```

6.1 Velocity Profile Array Generation of train on tiny track segment

Consider a tiny track segment ts and a train T . We wish to generate a velocity profile array for the train on the tiny track segment.

```

public VelocityProfileArray getVelocityProfileArray(TinyTrackSegment tts,
    Train t, double V_s, double V_e) {
    VelocityProfileArray VPA = new VelocityProfileArray();
    VelocityProfile vp1 = new VelocityProfile();
    VelocityProfile vp2 = new VelocityProfile();
    VelocityProfile vp3 = new VelocityProfile();

    double v_max, a, d;
}

```

```

if (tts.dir == "up") {
    a = t.A - tts.a_c;
    d = t.A + tts.d_c;
} else if (tts.dir == "down") {
    a = t.A - tts.a_c;
    d = t.A + tts.d_c;
} else {
    a = t.A;
    d = t.D;
}

v_max = Math.min(tts.V_max, t.V_max);
double S = tts.r_e - tts.r_s;
double v_f = Math.min(v_max, Math.sqrt(V_s * V_s + 2 * a * S));
double v_b = Math.min(v_max, Math.sqrt(V_e * V_e - 2 * d * S));

double v_s = Math.min(v_b, v_max);
double v_e = Math.min(v_f, v_max);

double s1 = (v_max * v_max - v_s * v_s) / (2 * a);
double s2 = (v_max * v_max - v_e * v_e) / (2 * d);

if (s1 + s2 > S) {
    v_max = Math.sqrt((V_e * V_e * a - V_s * V_s * d - 2 * S * a * d)
        / (a + d));
    s1 = (v_max * v_max - v_s * v_s) / (2 * a);
    s2 = (v_max * v_max - v_e * v_e) / (2 * d);
}

if (s1 > 0) {
    vp1.l = s1;
    vp1.a = a;
    vp1.v = v_s;
    VPA.add(vp1);
}

if (S - s1 - s2 > 0) {
    vp2.l = S - s1 - s2;
    vp2.a = 0;
    vp2.v = v_max;
    VPA.add(vp2);
}

if (s2 > 0) {
    vp3.l = s2;
    vp3.a = -d;
    vp3.v = v_max;
    VPA.add(vp3);
}

return VPA;

```


}

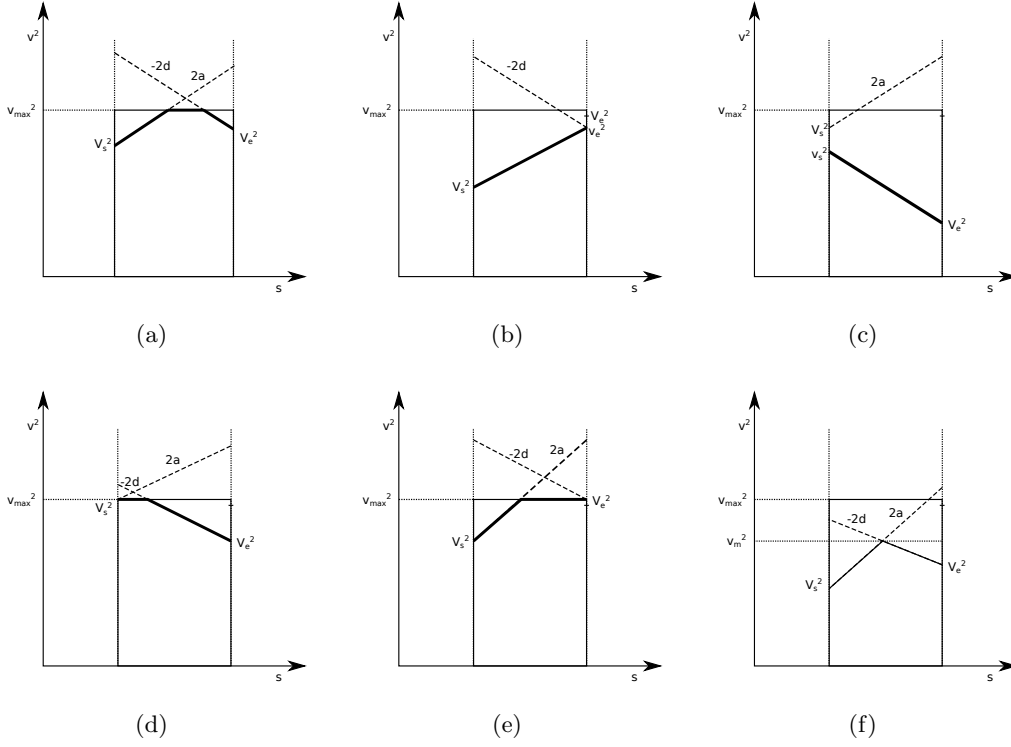


Figure 6.1: Velocity Profiles

Clearly, following holds:

$$V_t \geq v \geq 0$$

A velocity profile array is an ordered list of velocity profiles
 $vp_j = VPA.get(j)$ is the velocity profile for j^{th} part of the tiny track segment. $vp_j.l$ are the lengths of consecutive parts of the track segment. These parts do not mutually overlap with each other and they completely exhaust the track segment. We observe that

$$ts.r_s - ts.r_e = \sum_{j=0}^n vp_j.l$$

Also, for a continuous velocity graph,

$$\forall j < n, vp_{j+1}.v^2 = vp_j.v^2 + 2 * vp_j.a * vp_j.l$$

.

Within ts the only acceleration values that the train can take are a , 0 and $-d$.

6.2 Generation of list of Tiny Track Segments

```
public ArrayList<TinyTrackSegment> generateTinyTrackSegments (TrackSegment ts)
    ArrayList<TinyTrackSegment> ttsl = new ArrayList<TinyTrackSegment> ();
    ArrayList<Double> M = new ArrayList<Double> ();
```

```

    for (SpeedRestriction sp : ts.S) {
        M.add(sp.r_s);
        M.add(sp.r_e);
    }

    for (Gradient g : ts.G) {
        M.add(g.r_s);
        M.add(g.r_e);
    }

    M.add(0.0);
    M.add(ts.L);

    removeDuplicates(M);
    Collections.sort(M);

    int N = M.size();

    for (int i = 0; i < N - 1; i++) {
        double r_s = M.get(i);
        double r_e = M.get(i+1);
        SpeedRestriction sp = findOverlappingSpeedRestriction(ts.S, r_s, r_e);
        Gradient g = findOverlappingGradient(ts.G, r_s, r_e);

        TinyTrackSegment tts = new TinyTrackSegment();
        tts.r_s = r_s;
        tts.r_e = r_e;
        tts.V_max = sp.v_max;
        tts.dir = g.dir;
        tts.a_c = g.a_c;
        tts.d_c = g.d_c;
        ttsl.add(tts);
    }

    return ttsl;
}

```

Certain assumptions are $r_e > r_s$, $v_{max} > 0$, $A > a_c \geq 0$, $D > d_c \geq 0$

Typically, when a train uses links, it does not traverse the entire block but just a region of it. Hence, we have to choose the relevant tiny track segments for generating the velocity profile array. After generating the exhaustive list of tiny track segments for the block the function to obtain the list of tiny track segments within a certain region is as follows:

```

public ArrayList<TinyTrackSegment> getTinyTrackSegments(TrackSegment ts, double sm, double em) {
    ArrayList<TinyTrackSegment> ttsl = new ArrayList<TinyTrackSegment>();
    ArrayList<TinyTrackSegment> T = generateTinyTrackSegments(ts);

    for (TinyTrackSegment tts : T) {
        // lies withing sm (starting milepost) and em (ending milepost)
        if(sm <= tts.r_s && tts.r_e <= em)
            ttsl.add(tts);
    }
}

```

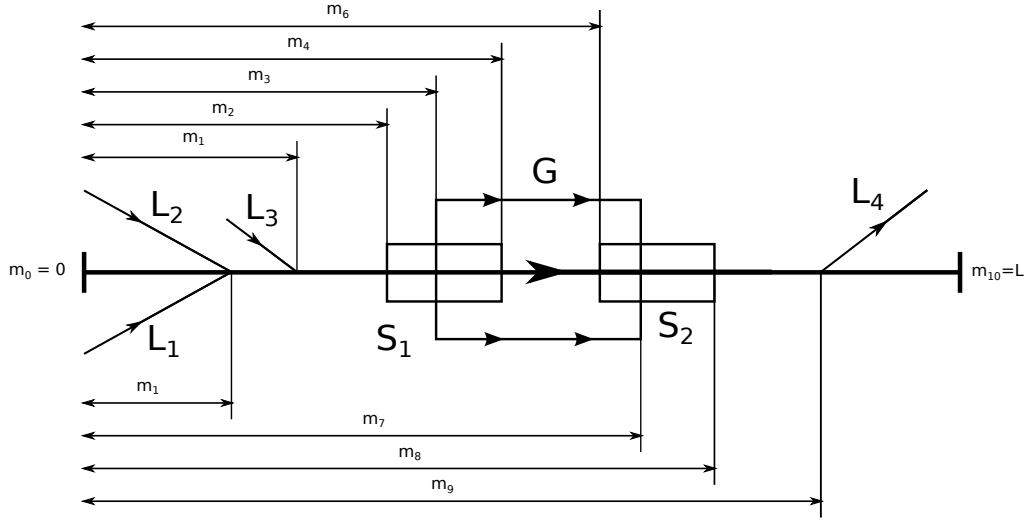


Figure 6.2: Generating Tiny Track Segments

```

    return ttssl;
}

public ArrayList<TinyTrackSegment> getTinyTrackSegments(ArrayList<Link>, Block s,
    VelocityProfileArray VPA = new VelocityProfileArray();
    ArrayList<TinyTrackSegment> ttssl = new ArrayList<TinyTrackSegment>();

    int N = LL.size();
    double sm = 0, em;

    Block block = startBlock;
    for (Link link : LL) {
        em = link.pbrm;
        ttssl.addAll(getTinyTrackSegments(block, sm, em));
        ttssl.addAll(generateTinyTrackSegments(link));

        block = link.nextBlock;
        sm = link.nbrm;
    }

    em = 0;
    ttssl.addAll(getTinyTrackSegments(block, sm, em));

    return ttssl;
}

```

6.3 Velocity Profile Array Generation on a list of tiny track segments

Now, given is a list of tiny track segments *TTSL* and a train *T*. The tiny track segments appear consecutively. The train encounters the tiny track segments in the same order as they are present in the list.

$$\forall j, 0 \leq j < TSL.size(), TSL.get(j).r_e == TSL.get(j+1).r_s$$

We have to generate a velocity profile array for the region occupied by the tiny track segments in the list *TTSL*.

```
public VelocityProfileArray getVelocityProfileArray(
    ArrayList<TinyTrackSegment> TTSL, Train t, double V_start,
    double V_end) {
    VelocityProfileArray VPA = new VelocityProfileArray();
    int N = TTSL.size();

    // forward velocities
    double[] v_f = new double[N + 1];

    // back velocities
    double[] v_b = new double[N + 1];

    v_f[0] = V_start;

    for (int i = 1; i < N + 1; i++) {
        double v = v_f[i - 1];

        // previous
        TinyTrackSegment ttsp = TTSL.get(i - 1);
        // current
        TinyTrackSegment ttsc = TTSL.get(i);

        double a;
        if (ttsp.dir == "up")
            a = t.A - ttsp.a_c;
        else
            a = t.A + ttsp.a_c;

        // maximum velocity possible
        double v_max = Math.min(t.V_max, ttsc.V_max);

        // length
        double s = ttsp.r_e - ttsp.r_s;

        // next forward velocity
        v_f[i] = Math.min(v_max, Math.sqrt(v * v + 2 * a * s));
    }

    v_b[N] = V_end;
    for (int i = N - 1; i >= 0; i--) {
        double v = v_b[i + 1];
```

```

// next
TinyTrackSegment ttsn = TTSL.get(i);
// current
TinyTrackSegment ttsc = TTSL.get(i-1);

double d;
if (ttsn.dir == "up")
    d = t.D + ttsn.d_c;
else
    d = t.D - ttsn.d_c;

// maximum velocity possible
double v_max = Math.min(t.V_max, ttsc.V_max);

// length
double s = ttsn.r_e - ttsn.r_s;

// next forward velocity
v_b[i] = Math.min(v_max, Math.sqrt(v * v + 2 * (-d) * (-s)));
}

// start velocities for tiny track segments
double[] v_s = new double[N];

// end velocities for tiny track segments
double[] v_e = new double[N];

for (int i = 0; i < N; i++) {
    v_s[i] = Math.min(v_f[i], v_b[i]);
    v_e[i] = Math.min(v_f[i+1], v_b[i+1]);
}

for (int i = 0; i < N; i++) {
    TinyTrackSegment ts = TTSL.get(i);
    VelocityProfileArray vpa = getVelocityProfileArray(ts, t, v_s[i], v_e[i]);
    VPA.addAll(vpa);
}

return VPA;
}

```

6.4 Generation of Entire Velocity Profile Array of train

Suppose a train T starts from block *startBlock* and uses a list of links LL . The *startBlock* along with LL tells us the exact path followed by the train. Thus, we generate the list of tiny track segments covered by the train using the following function.

```

public ArrayList<TinyTrackSegment> getTinyTrackSegments(ArrayList<Link>, Block s,
    VelocityProfileArray VPA = new VelocityProfileArray();
    ArrayList<TinyTrackSegment> ttsl = new ArrayList<TinyTrackSegment>();

```

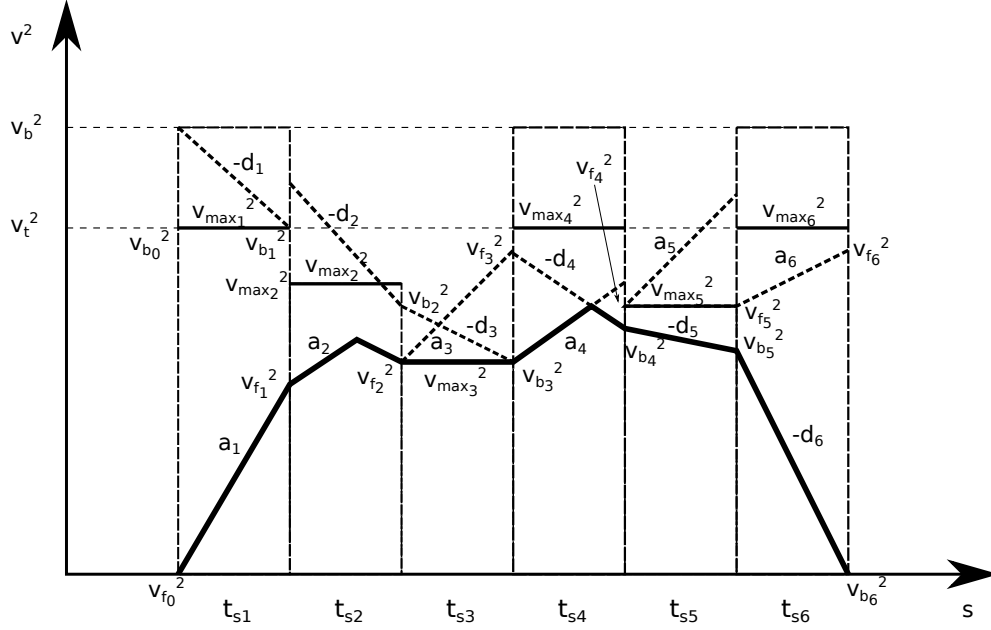


Figure 6.3: Velocity Profile for List of Tiny Track Segments

```

int N = LL.size();
double sm = 0, em;

Block block = startBlock;
for (Link link : LL) {
    em = link.pbrm;
    ttsl.addAll(getTinyTrackSegments(block, sm, em));
    ttsl.addAll(generateTinyTrackSegments(link));

    block = link.nextBlock;
    sm = link.nbrm;
}

em = block.getLength();
ttsl.addAll(getTinyTrackSegments(block, sm, em));

return ttsl;
}

```

The velocity profile array for the entire path of the train is then generated by making a call to the following function:

```

public VelocityProfileArray getVelocityProfileArray (Train t,
    Block startBlock, ArrayList<Link> LL, double V_start, double V_end) {

```

```
ArrayList<TinyTrackSegment> TTSL = getTinyTrackSegments(LL, startBlock);
VelocityProfileArray VPA = getVelocityProfileArray(TTSL, t, V_start, V_end);
return VPA;
}
```

Chapter 7

Work on Simulator and Visualizer

7.1 Re-modelling simulator to handle complex railway sections

Initially, the simulator was able to handle only simple sections. Such sections do not have any junction stations. Every station, block, loop, gradient and speed restriction had fixed mileposts. So, multiple paths of different lengths could not be accurately modelled. In order to properly model the track characteristics for tracks of different lengths and locations some remodelling of certain entities had to be done. Assigning a of having starting and ending milepost for a block can become tricky in a complex railway section. As one may see in figure 1.1, block 23 can be reached via the through lines between Diva and Thane or via the slow suburban lines through Mumbra and Kalva stations. Hence, our final goal is to get rid of the notion of absolute mileposts and just introduce the lengths of the blocks and loops. However, there were some intermediate steps which had to be taken care of before doing this.

7.1.1 Re-modelling of links

Links, typically, have finite length. A link connects two blocks. One of them is called previous block, the other one is called next block. Generally, a link allows motion only in one direction. Hence, the notion of previous and next block. A link, as modelled in the simulator, began at the exact end of the previous block of that link and ended at the exact beginning of the next block of the link. Further, the length of the link was always taken as 0. Moreover, links which connect blocks on different tracks have a speed restriction of 15km/hr or 30 km/hr. (Two consecutive blocks on the same track are also connected, but they are not connected by a separate physical entity as link. But to ensure uniformity, even consecutive blocks on the same track are said to be “connected by link”, although that link is of zero length). Links are mostly as long as 90 metres. However, the train has to come to a speed of 15 km/hr before entering its head into the link and has to travel with the slow speed till its tail exits the link. Assuming the length of train as 300 metres, it has to travel around 390 metres with that slow a speed. That comes to around 2 minutes, which is a significant time while considering scheduling. In case of longer freight trains, the crossing time becomes even more significant. So changes were made to take into account the link’s exact beginning and ending position with respect to the previous and next block, the speed limit of the link and the length of the link. Changes were made in the algorithm to consider the link while determining the velocity profile array for the train as well as its arrival and departure times throughout the section.

7.1.2 Re-modelling of gradients and speed restrictions

Originally, gradients and speed restrictions had a fixed beginning and ending milepost. So, all the blocks or loops which overlapped with these mileposts were assigned gradients and speed restrictions in that much region. For example, suppose a block has 0, 4 as starting and ending

mileposts respectively, while a gradient has 2, 6 as starting and ending milepost. Even though this gradient causes track characteristics to change for other tracks and not this particular block, a gradient from milepost 2 to milepost 4 would be assigned to the above mentioned block. Hence, gradients and speed restrictions were re-modelled to be specifically assigned to individual blocks as and when needed. The positions of the gradients and speed restrictions are indicated by their relative mileposts. If a block with block number b of length 4 units, has a gradient G with slope “1/10”, direction as “up”, and starts at 2 units from the start of the block and end at 3 units from the start of the block, it is represented as $(b, “1/10”, “up”, 2, 3)$. Even the locations of speed restrictions are given by their relative mileposts to the start of the block. (Note: For a up-block and a common-block, the start of the block means that side of the block from where the up-trains enter. For down block, start of the block means that side of the block from where the down trains exit.)

Reasons for Speed restrictions

Unlike cars, trains do not have differential for their wheels. So, while going along a turn, banking has to be provided to allow trains to move at a greater speed. Otherwise, the outer wheels would slip. But banking is provided for fixed speeds. This would mean that faster trains would have to travel at this slower speed while slower freight trains would have to travel at this faster speed. When the radius of curvature is big enough, banking is a good solution to have reasonably faster speeds. But, during crossing, there is space constraint, so there cannot be any banking. If the trains go at faster speed, there will be some slipping which can lead to derailment. Hence, there is such a speed limit on the crossings.

7.1.3 Changes in visualizer

The arrival and departure times for all the blocks that a train visits are recorded in the “timetables” datastructure of the train. If a block is denoted by its starting and ending milepost as (s,e) and the corresponding arrival and departure times for that block as (a,d) , then on the distance vs time plot of the train, points (s,a) and (e,d) would be joined to show the train movement. This only showed the proper movement of trains from block to block. It was further improved to show a more appropriate distance vs time movement of the train within every block in the path of the train. If the train came to halt at some loop and waited there for some time, it now shows a horizontal line on the plot indicating that the train did not move during that much period of time. Refer figure 2.32. At Thane station, we can see the first down train waiting for some time before it actually starts moving.

7.1.4 Reference Timetables of Scheduled Trains

In a simple section, the reference timetable of the scheduled train would have had to include arrival and departure times at all the stations. This would be quite erroneous, if the user wanted to simulate a complex railway section. For example, in figure 1.1, if a train should be simulated between Kasara and Thane, giving reference time table entries for the same train for stations at Karjat, would be erroneous. Plus, the simulator would actually try to find a direct path between Kasara and Karjat, which there isn't. Ultimately, when the simulator couldn't find a direct path, it would cancel simulation of the train going from Kasara to Thane. Also, it wasn't possible to simulate the scheduled trains by starting and terminating them at intermediate stations. Now, the user may input only the arrival and departure time entries for the stations where the train has to halt. The order of the reference timetable entries should be in the order in which the train visits the stations. The user may also provide a timetable entry at a station with same arrival and departure times to indicate that the train passes that particular station but does not halt there. This can be useful, when a path has to be explicitly specified for a train.

Chapter 8

Conclusions

8.1 Future Work

The purpose of this project is to make the analysis of train scheduling extremely easy. More consistency checks can be added into the editor. Removing of blocks and loops affect the paths of the trains. The editor can be made to reflect if a path of train is completely obstructed because of such transactions.

More accurate train running model could be designed and tested with using the simulator. Currently, while a train's length is considered for scheduling, while determining the velocity profile the train is considered as a point object while determining the velocity profile.

There is a Kandivali-Dahisar section on Western Railways. SketchRail along with the simulator will be used to determine the impact on section capacity and train running because of crossover movements. Over 100 out of 400 local train services get delayed. The primary reason for such delays is suspected to be these crossover movements. The task will be to identify such conflicts and plan them ahead of time so that the timetables can be made more robust. The objective will be to reduce passenger inconvenience.

We would like to extend the current version of the simulator and editor to handle train scheduling and test-case designing to a general network of railways. In the output, we would expect a 3D visual of time vs location plot of trains. Thus in this particular view we would be able to have a 3D view of the entire velocity profile of the trains.

A module to analyze different scheduling strategies could be developed. The simulator could be further developed to suggest changes in the infrastructure to improve section performance. The user may also be able to run two strategies and then visually compare their performances. The performance metrics could be the maximum delay corresponding to one train, or the average deviation of the train from the desired schedule of the trains.

The simulator can probably consider public demand at every station and suggest more services. It may be able to suggest changes like repositioning of signals or splitting of blocks which would improve section performance.

8.2 Conclusions

An editor for inputting track geometry and train reference schedule for general sections on rail network was designed at IIT Bombay. The editor provides a transparent and usable interface for

representing stations, track segments and viewing train movements over the section. It provides an easy to use interface for the simulator developed at IIT Bombay in 2003.

The simulator which is very useful for theoretical analysis and practical decision support was further improved upon to handle complex railway sections.

Bibliography

- [1] N. Rangaraj, A. Ranade, K. Moudgalya, C. Konda, M. Johri, and R. Naik. Simulator for railway line capacity planning. In *International Conference of the Association of Asia Pacific Operational Research Societies*, December 2003.
- [2] Devendra Shelar, Priya Agrawal, N. Rangaraj, and A. Ranade. Simulation of multiple line rail sections. In *International Simulation Conference of India*, February 2012.
- [3] Viswanath Nagarajan and Abhiram G. Ranade. Exact train pathing. *Journal of Scheduling*, 11(4):279–297, August 2008.
- [4] Oracle java standard edition documentation.

Appendices

.1 Terminology

- section: A section can be described as a linear group of stations. The railway lines are assumed to be parallel to each other.
- Freight train: Train used to transport goods and raw materials.
- Passenger train: Train used to transport public.
- Block: Any part of a railway track can be considered a block. It has definite starting and ending locations to mark its position on the railway track.
- Station: It is a designated location in a railway section where the trains can halt so that the public may get onboard or offboard or the goods may be loaded or unloaded.
- Loop: A block in a station area is termed as loop.
- Link: The part of a track connecting two blocks or a block and a loop on same or different railway tracks.
- Uplink: A link which connects two blocks in up direction is termed an uplink.
- Downlink: A link which connects two blocks in down direction is termed as a downlink.
- Crossover: It is a link that crosses a line reserved for opposite direction in order to connect two parallel tracks.
- Velocity Profile: It gives us information about the various speeds of a particular train at various points in a particular block.
- Block occupancy: The various times for which a certain block is occupied by various trains.
- Loop occupancy: The various times for which a certain loop is occupied by various trains.
- Up main line: The railway line designated for the transport of trains in the up direction. We may sometimes refer to the up main line as simply up line.
- Down main line: The railway line designated for the transport of trains in the down direction. We may sometimes refer to the down main line as simply the down line.
- Third line: We will sometimes refer to the bidirectional line in a 3-line section as the third line.
- Throughput of section: The number of trains crossing a section in any direction in unit time.
- Latency of section: The average time taken by a train to cross the section.
- Direction switch: It is the event of switching the direction of the bidirectional line.