Software components for simulation of guideway transit systems

J. Rodriguez

Institut National de Recherche sur les Transports et leur Sécurité, Centre de Recherche et d’Évaluation des Systèmes de Transport Automatisé, 20, rue Elisée Reclus, F-59650 Villeneuve d’Ascq, France

ABSTRACT

This paper describes software components for simulating a wide range of guideway transit systems. The objective is to build a library of object classes which can be reused in different applications. The developed program presents the following features: provision of on-line graphical animation, user interaction with all simulation objects and data collection and visualisation. Two applications have been been created with the proposed object classes.

1 INTRODUCTION

Over the last years, INRETS has developed various simulation programs for the purpose of assessing guideway transit systems ([1],[2],[3]). To avoid unnecessary software developments, a set of software components has been created which can be reused for future simulation applications. We have chosen an object oriented programming approach and Smalltalk 80 as software development environment. Section 2 presents some concepts and terms of object oriented programming.

For simulating a wide range of guideway transit systems, we must use as general as possible a model. Three categories of object classes modelling the behaviour of a guideway transit system have been identified. The categories and the sections to be examined are: (1) track topology in section 4, (2) rolling stock in section 5, (3) train motion control in section 6. The library also includes useful classes for developing and validating the model, these classes are grouped into the user interface category which is detailed in section 7. Section 8 gives a description of the applications followed by the conclusion.

2 OBJECT-ORIENTED PROGRAMMING

This chapter attempts to present object-oriented terms and concepts. A programming language is object-oriented rather than procedure-oriented if an application is no longer a collection of procedures that act on a database, but a collection of data objects that interact with one another via built-in routines called methods. An object is made up of one or more private variables (the data)
combined with a set of methods for manipulating that data. Each method is a specialised subroutine. These two parts of an object are also known as **state** and **behaviour**. The name of a method is sometimes called a **message** but technically a message consists of the method name plus any arguments. A method is a sequence of expressions, each expression being an object reference (the receiver) followed by a message. For example, let `trainA` refer to an object modelling a train, the expression to slow down its speed to 5 m/s is:

```
trainA slowDownTo: 5.
```

A set of objects which have the same data template and the same behaviour is called a **class**, each object is called an instance of the class. In some languages (e.g. Smalltalk) a class is an object, then a class can be viewed as a method-holding object affixed to a data template. To create an instance, a message is sent to the class. For example, let `Train` (we adopt the Smalltalk convention for class names, i.e. the first letter is capitalised) be a class of train objects; the following expression with the **new** message creates the previous train object:

```
trainA := Train new.
```

The classes are organised in a hierarchy of specialisation. A class inherits all of the methods and variables of its super-classes. Programming in an object-oriented language amounts to building new class objects and reusing the existing ones. Reusing can be limited to the creation of instances if there exists a class which models correctly the state and the behaviour of a real object. Otherwise, reusing is done by defining a subclass which specialises or increases either the methods or the variables.

### 3 OUTLINE OF THE SIMULATION MODEL

Figure 1 illustrates a simplified overall architecture of objects designed for modelling the behaviour of a guideway transit system. In the picture of an object, its class can be identified by the upper black background box. The labelled overlap boxes on the left side are the messages understandable by an object, and the inside boxes are the variables. Not all messages, variables and links between objects are represented to simplified the figure. The four categories of objects are delimited by dashed lines.

All the objects used in a simulation can also be divided in two categories, the ones whose behaviour is dependent on the time and the ones which are not directly dependent on time. Within the latter category, some objects can be completely static for example a track piece, and other objects have variables changing over time but controlled by another object for example a block is controlled by a blockController. All classes of "time step dependent" objects must define a **step** message, this message will be sent by a simulation clock during each time step. The rate between cpu-time and simulation time can be set at the start up and changed at any time during the simulation, i.e. the simulation can be slowed down or accelerated (of course only if the host machine limits are not reached). The method performed by the **step** message defines the dynamic behaviour of the receiver during the time step interval. For example, when a blockController receives this message, the method will test each block for trains still within the limits of the block and reset the status of the block (released/held) and the aspects of the protecting signals.

A considerable effort has been devoted to the definition of tools for building, evaluating and updating the model. These objects are grouped in the category user interface, they are based on Model-View-Controller objects architecture provided by Smalltalk environment ([4]).
The next chapters give more details on the objects of each category.

Figure 1: Overall architecture of the objects

4 TRACK TOPOLOGY

In order to simulate any kind of guided transit systems, no limitation can be assumed on the track layout. We have modelled the track topology with two kinds of object: the track pieces and the switches. The track pieces are connected by switches. The result can be viewed as a graph whose nodes are the switches and the links are the track pieces. This representation of railway track networks is frequently used but in our case the information concerned with these objects is strictly limited to (1) the topology of the network, (2) the civil engineering data e.g. length, curve, gradient.

Figure 2 shows the connection of three track pieces with one switch. A switch has two states: right or left. Each state corresponds to a possible pair of connected track pieces. These states are exclusive, i.e. only one pair of track pieces can be connected at the same time.

A track piece has a local reference, by this means different kinds of position can be defined. The class Position we have used defines instances with two variables: track pieces and abscissa. All positions can be converted within different references but the default and unambiguous position value is defined with the local reference of its track piece. A useful message in the class Position is the one whose expression gives the distance between two instances. Figure 3 (a) shows that this message must include a direction argument to suppress the ambiguity to compute the distance value. The direction is coded by the sign w.r.t. the local reference of the track piece. But a distance message also needs an extra argument: a criterion for seeking a path. Figure 3 (b) illustrates a case of three possible paths in the same directions between two positions. The default criteria for finding the path we have used is the current state of the switches. As
in [5], any search strategy can be used (e.g. depth first, best first) by changing the path search message which is sent by a distance method. Notice that the double vertex representation ([5]) is replaced here by the double states of the switches used by the path searching messages. The expression below is an example of a distance expression specifying all arguments:

\[
\text{positionA distanceTo, positionB withDirection: (-1)}
\]

\[
\text{withStrategy: #depthFirst.}
\]

![Diagram of switch object](attachment:switch_diagram.png)

**Figure 2: Definition of a switch object**

The piece N°1

Possible connections:
- right: [Track piece N°1-Track piece N°2]
- left: [Track piece N°1-Track piece N°3]

![Diagram of distances](attachment:distance_diagram.png)

**Figure 3: Distances with the two directions (a) and three distances with the same direction (b)**

5 ROLLING STOCK

All classes of train objects are grouped in the rolling stock category. Each train has its own kinematic variables (position, speed, acceleration, jerk), mechanical characteristics (length, weight) and motorization characteristics. Not all these characteristics are needed for a particular application. For example, we have not used the motorization ones for the applications done (though the jerk and acceleration were limited). A good use of the motorization characteristics needs a very specialised code which is suited to energy minimisation studies for example; its description is outside the scope of this paper.

Private methods defined in the Train class provide the integration of the kinematic variables starting from previous step values and the new value of a key variable (e.g. jerk). A train receives from its environment two kinds of order for controlling its kinematics: kinematic targets and speed limits. A driving rule for controlling the progression of a train can be: "the kinematic variables of the train must match its target as soon as possible". But any kind of driving algorithm can be implemented and tested depending on the key variable and the simulation specifications (e.g. minimising criteria such as run-time, energy, comfort, ...). We have implemented two driving algorithms, one has the jerk as key variable [8] and the other one has the speed as key variable. These algorithms can be viewed as the model of a "driver behaviour" but it is the train simulation control objects which set the orders to the object trains.
6 TRAIN MOTION CONTROL

We have seen in the previous section that the train objects receive two kinds of orders: kinematic targets and speed limits. A kinematic target is a pair of values <position, speed>. Kinematic targets are nodes of an oriented graph. The kinematic target of a train is deduced from its position: each time a train has reached a target position, the next target is assigned as a new target to reach. There are two kinds of kinematic targets: static targets and dynamic targets. The former have a constant position and speed value whereas the latter have time varying position and speed. Generally, dynamic targets are attached to the kinematics of a train and are used in the simulation of moving block systems. A speed limits table maps the speed limits of the all network and contains values which can be changed at any time. Figure 4 (a) shows a simplified layout of a track, figures 4 (b) and (c) show the mapped kinematic targets and speed limits.

Controlling train motion (stopping signal, slow speed signal, establishing a route, ...) is basically realised by changing the graph of targets and the speed limits table. Basic operations (e.g. deletion, insertion) on the graph of targets and speed limits table have been defined. The control device objects must manage the graph targets and speed limits table of the network area which they have in charge. By these means, headway can be controlled by fixed blocks, fixed moving blocks, as well as flexible moving blocks. In the applications described above (fixed block systems), this control has been carried out by the instances of BlockController, RouteController and PlatformController. But only the instances of BlockController execute directly the basic operations on the graph of targets and speed limits table, the other two controllers are client objects.

Figure 4: Mapping of the graph of kinematic targets and the speed limits
7 USER INTERFACE

The objects of the user interface category are all based on the model-view-controller (MVC) triad. MVC is an object architecture widely used in the Smalltalk environment for structuring the user interface so that all or part of it can be reused in other applications ([4]). The first component is the model which handles the data storage and processing. In our case, model will be the objects modelling the behaviour of the simulated system. The second component is the view for displaying output and the third component is the controller that enables the user to interact with, or control, the application. The controller handles user input such as menu selections and keyboard activity.

During the simulation, at the top of the user interface tools we have a "launcher view" which is in fact a menu for controlling the simulation, opening tools and selecting customised operation processes (see figure 5).

![Figure 5: Launcher view, global and magnified view of a network](image)

The first user interface tool we developed was an animated view of the all network (figure 5). The objects are represented as much as possible with scale-pictures (i.e. track pieces, platforms, trains) as well as the movement of animated pictures (i.e. trains). The model of the MVC triad is a set of objects, this set can be modified and the rules for composing the pictures can be
changed. The view and controller are respectively subclasses of `StandardSystemView` and `StandardSystemController` to inherit of the close, frame, move and collapse operations. When this view is opened, the user is requested to drag the rectangle area of the view. As the network can be too complex, a subclass has been defined which scales can be changed and more objects or details can be added. Of course in these "magnified views", only a part of the network is represented. Part of the network can be selected from the global view of the network, (see figure 5 as an example).

An important specification of the simulation tools sought was to be able to interact with the system simulated and more precisely to be able to introduce any kind of perturbations. To do this, we have defined an "instances browser" (see figure 6). An instances browser view is compose by four subviews, the first one is a `SelectionInListView` on a list of objects classes. When a class is selected, the list of the instances of this class is displayed on the second view. If an instance is selected, the third view allows selection of a variable and displays its value in the fourth view. But this last view is also a Smalltalk interpreter, i.e. Smalltalk expressions can be evaluated within the context of the selected instance. In the figure 6 for example, the message `break` is sent to the selected instance of Train. The most recently developed user interface tools are a graphic editor to ease the definition of the layout of the tracks [7] and a monitoring tool which can visualise any sequences of data coming from the simulation.

![Figure 6: Instances browser view](image)

8 APPLICATIONS

These software components have been used firstly for a simulation of a single track system with fixed block control and with automatic train operation. This application was part of an economical and technical assessment study of single track systems named Vulcain [6]. The line simulated for Le Havre town is 6.25 km in length and has 9 stations (see figure 5). The aim of the simulation was to evaluate the performances of this kind of transport system, namely the effects of degraded operating procedure as a push-out recovery procedure. This procedure is triggered when a train has failed, it is then pushed to the maintenance area by an operating train. This procedure has not been carried out automatically because some decision problems had to be resolved (choosing the operation trains or the maintenance area, ...). Customised commands have been
programmed and included on the launcher menu (figure 5) to play scenarios of push-out recovery procedures.

A second application is currently being developed, it is line 2 of the VAL system of Matra-Transport for the town of Lille. It is also a fully automated transport system line. This line is 30 km in length and has 40 stations. This line can have different headways in the same direction due to turn back facilities in different parts of the line. The trains can go through a central loop route or go through a complete loop route. The goal of this simulation is to evaluate the impact on headway of different kinds of perturbations.

9 CONCLUSION

The software components presented in this paper have been used in two applications using train protection based on fixed block system. This shows that the use of kinematic targets and speed limits assignment for controlling train motion is not only suited for implementing moving block systems. This also means that with this analysis, mixed fixed/moving block systems can be simulated without tedious efforts.

The user interface objects allow extended facilities to interact with the simulation and the simulated components. Playing a scenario with any kind of perturbation is very easy. Future work will be the development of applications on other kinds of transit systems like a tramway line by including in the library "cross-roads object" classes.

ACKNOWLEDGEMENT

The author wish to thank Mr. Y. David (Director of INRETS-Villeneuve d'Ascq) for his encouragements and J.L. Lévéque (Engineer at INRETS-Villeneuve d'Ascq) who initialized this work and has provided helpful suggestions and comments. I also thanks P. Ely for proof-reading this paper.

REFERENCES