SIMULATION FOR CONTROL SYSTEM DESIGN AND TRAFFIC ANALYSIS

Richard D Peters
Peters Research Ltd, 47 The Crescent, High Wycombe, Bucks HP13 6JP, UK
fax: +44 (0)1494 452359, email: richard.peters@peters-research.com

ABSTRACT

Elevate is a development platform for elevator control systems and an advanced elevator traffic analysis tool. The program is written in the C++ language using the latest object oriented programming techniques, and has a Microsoft Windows[†] user interface. Passengers are generated automatically from arrival rates entered by the user. The elevators answer passenger calls as directed by the selected dispatcher algorithm. An analysis of passenger waiting and transit times is given. *Elevate* applies research in ideal elevator kinematics giving total control of elevator speed profiles.

1. INTRODUCTION

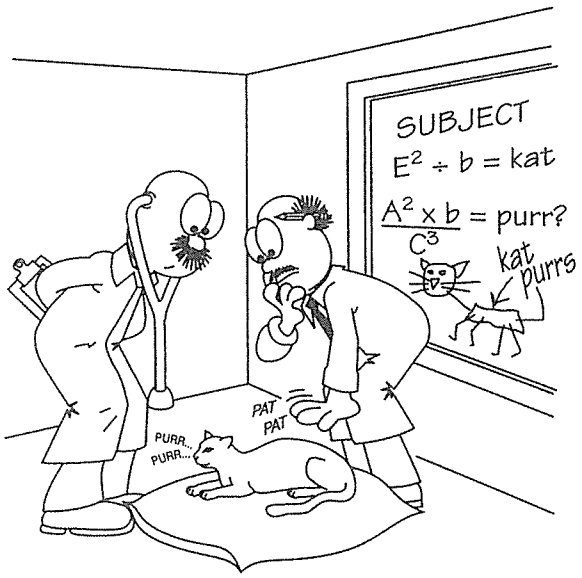
Elevate is a development platform for elevator control systems and an advanced elevator traffic analysis tool. *Elevate's* features and functions include:

- Dynamic simulation providing a visual display of the elevators as they answer passenger landing and car calls.
- A Microsoft Windows interface.
- Advanced traffic analysis tool for planning elevator installations.
- Full user control over the inputs to the systems, e.g. number of elevators, speeds, passenger arrival rates.
- Standard and Advanced options allowing the user to enter basic information for a quick analysis, or comprehensive data for a detailed model.
- Kinematics research applied to generate accurate elevator speed profiles.
- Results include graphs of passenger waiting and transit times.
- Data and results can be transferred to a spreadsheet for further analysis if required.
- Programmed in Microsoft Visual C++ using object oriented technology.
- Platform for developing, testing, and demonstrating control systems.

In this paper we will review how *Elevate* is designed using object oriented technology, give an overview of its interface, and discuss some applications.

2. OBJECT ORIENTED PROGRAMMING

Traditional structural programming techniques break a program into several smaller tasks by defining a set of functions. Object oriented programming (OOP) builds on structured programming techniques by introducing objects. The variables and functions of an object are defined by the class to which it belongs.



For an example of how OOP is mimicking the real world, consider *Ginger* the cat in Figure 1.

The world has a class *cat*. Everything in the *cat* class has a set of the same variables (no of paws, age, sex, etc.) and a range of functions (if you chase it, it runs; if you pat it, it purrs). "Ginger" is an object, and an instance of the *cat* class. He has all the functions and variables of a cat.

Figure 1 Ginger the cat graphic from [1]

Once a class is defined, its complexities are hidden, so we can create and apply objects simply. In this way OOP helps break down complex problems into manageable parts that are easy to work with as they represent familiar ideas or components.

Applying this approach, consider a *circle* class with a radius variable *r*, and functions *CalculateArea()* and *DrawCircle()*. Here is an extract of a C++ application using the *circle* class (the functionality of which has been defined elsewhere):

```
//(double lines precede comments)

//create a circle object, called "cir" with a radius r
circle cir(r);

//display the circle's area
cout << "The area is " << cir.CalculateArea();

//draw the circle on the screen
cir.DrawCircle();
```

Returning to elevators, we can define the class *elevator* with variables such as *capacity* and *speed*, and functions such as *StartJourney()*. We can create as many elevator objects as we need; each elevator object is independent, but may use all the variables and functions defined by the class.

3. APPLYING OOP TO ELEVATE

Elevate has six main classes which it uses to implement a simulation. They are:

building The *building* class defines the building in terms of number of stories and story heights.

motion The *motion* class generates elevator kinematics curves, examples of which are given in Figure 2. To use this class, an "elevator" specifies the journey distance, rated velocity, acceleration and jerk; *motion* outputs the current distance travelled, and velocity at any time, t since the journey began.

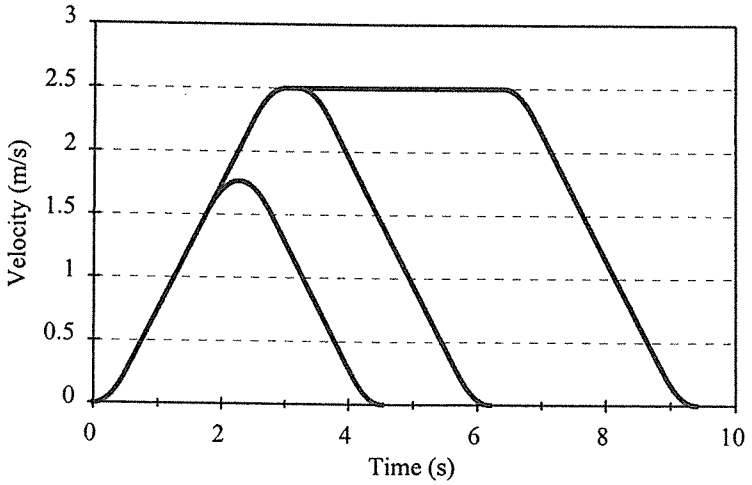


Figure 2 Example velocity curves produces by *motion* class.

elevator The *elevator* class defines an elevator (rated speed, capacity, floors served, etc.) and its current status (position, speed, load, etc.). The *motion* class is applied to enable the elevator to move according to the selected journey profile. The *elevator* class includes algorithms to allow elevators to answer landing and car calls according to the principles of directional collective control. (Most elevator control systems adopt a directional collective control strategy regardless of the complexities of their dispatcher algorithm.)

dispatcher The *dispatcher* class defines rules for deciding which elevator is to serve each landing call. The standard dispatcher logic has been based on conventional group control with dynamic sectoring as defined by Barney and dos Santos[2].

person The *person* class defines a person, what time he/she arrives at the landing station, where he/she wants to go, their mass, etc. Once the journey is complete, the class provides details about passenger waiting and transit times.

traffic The *traffic* class converts user information about the passenger traffic into a corresponding set of *person* objects.

4. DATA ENTRY

Elevate's interface is Windows based, and allows the user to edit all the system data in dialog boxes containing standard Windows controls (radio buttons, drop downs, etc.), and a spreadsheet-like control for tabular data entry. The program uses a multi-document interface, so the user can be working on a number of different simulations at the same time. In addition to the standard Windows features (save, print, etc.) there are five data entry dialog boxes which can be accessed via the menus or button bar:

Building data In *Building data* the user enters floor names and levels.

Elevator data In *Elevator data* the user enters details about the elevators. The dialog box has two modes. In Standard mode only basic information is entered, e.g. number of elevators, capacity, speed; *Elevate* selects default values for other variables. In Advanced mode, the user can edit all parameters including the acceleration, jerk, motor start delay, and dwell times.

Passenger data In *Passenger data* the user enters details of the passengers that will be transported by the elevators. Again, there are two modes of operation. In Standard mode the user enters basic information, e.g. a five minute handling capacity and floor populations (as a basis for determining passenger destinations). In Advanced mode the user can define complex traffic flows using arrival rates and destination probabilities.

Simulation data In *Simulation data* the user can select the control algorithm, time slice, and frequency of the graphical display being updated.

Job data In *Job data* the user can enter job and calculations titles.

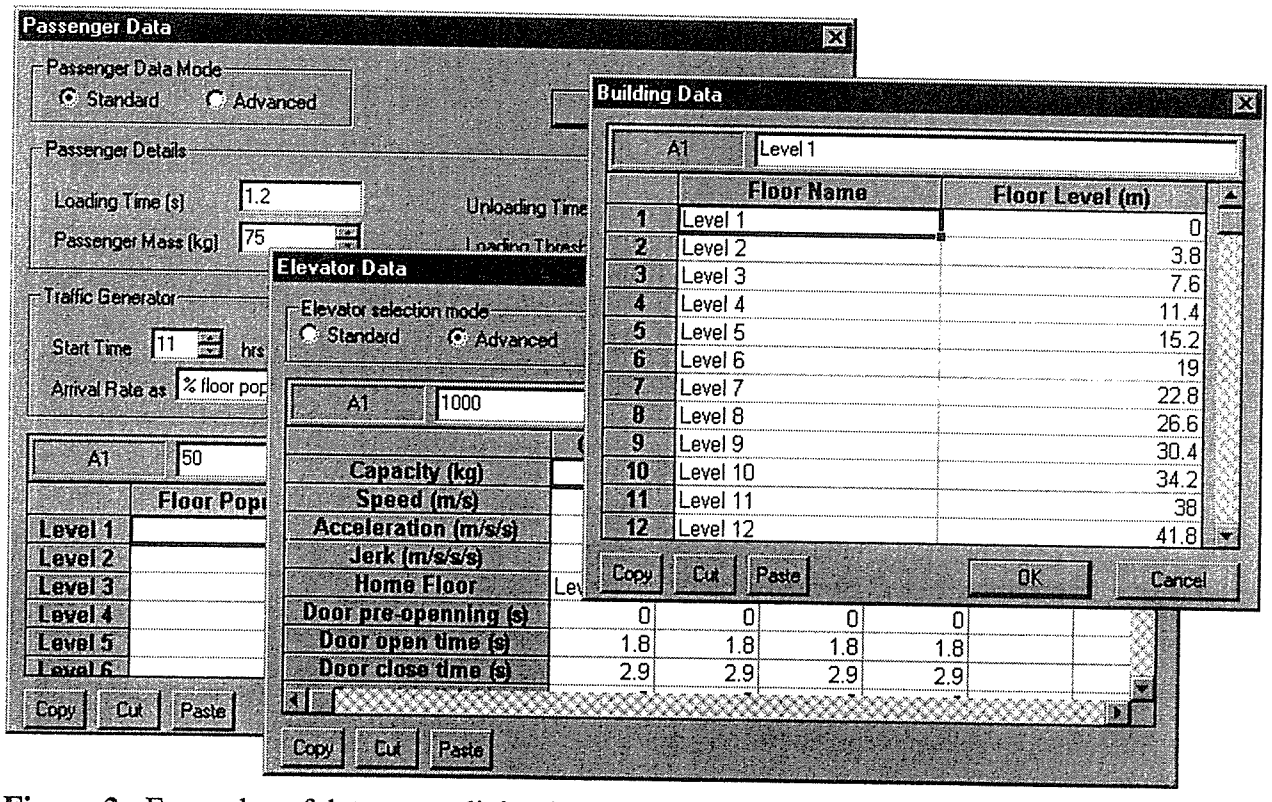


Figure 3 Examples of data entry dialog boxes

5. RUNNING THE SIMULATION

The passenger generator uses the *traffic* class to create passengers (*person* objects) in software based on the *Passenger data* entered by the user.

The program then performs a time slice simulation; it calculates the status (position, speed, etc.) of the elevators, increments the time, re-calculates status, increments time, and so on. Provided that there are not too many other demands on the computer's processor, the simulation will run faster than real time on a Pentium PC using a time slice of 0.01 seconds.

The main area of the screen is used to give a visual display of the simulation as shown in Figure 4. The user can zoom in/out of this display using buttons on the Toolbar.

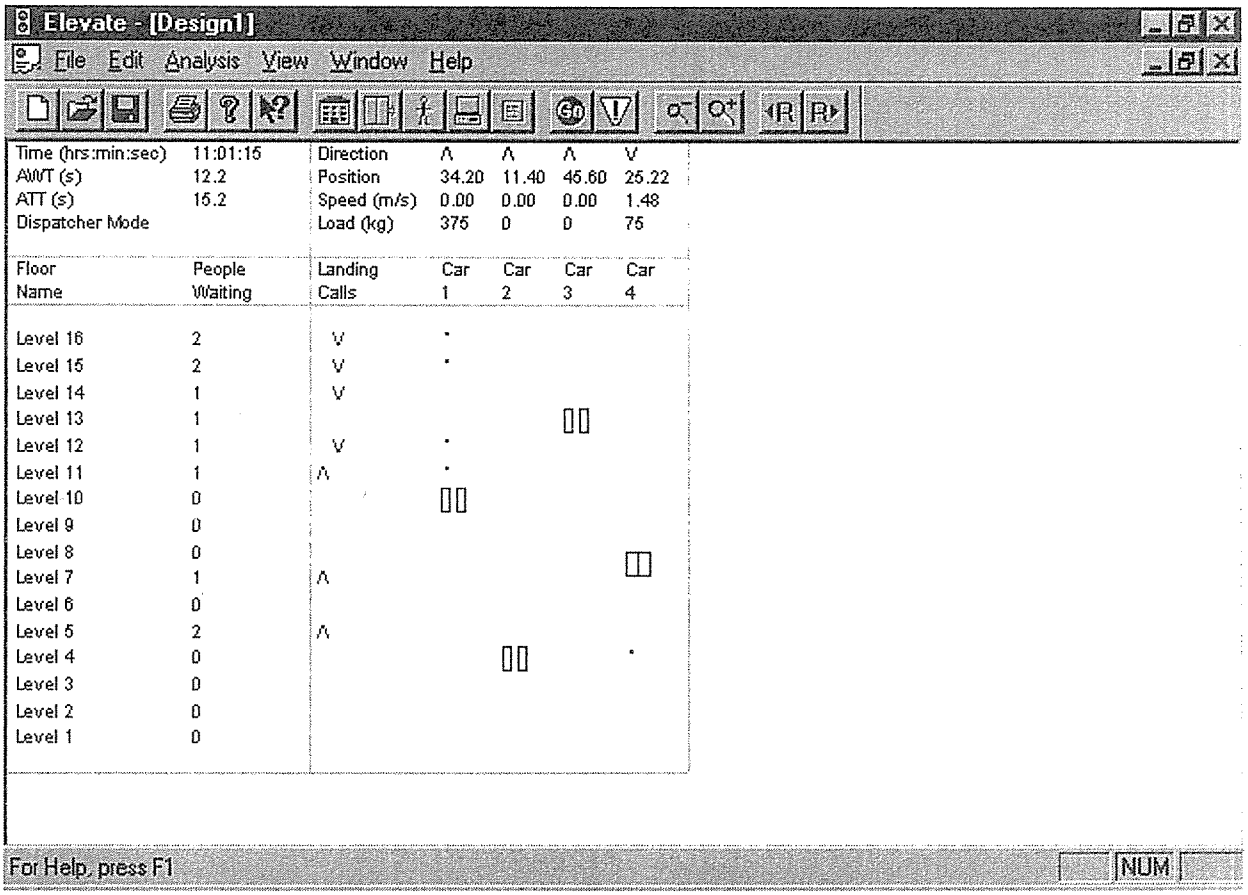


Figure 4 Simulation display

Elevators are displayed according to their current position and door status:

Indicates that the elevator's doors are fully closed.

Indicates that the elevator's doors are opening or closing.

Indicates that the elevator's doors are fully open.

Landing and car calls are displayed according to their status:

∧∨	Red arrows indicate which up and down landing calls have been registered by waiting passengers at each floor.
■	Red squares indicate car calls registered by the passengers travelling in each elevator. Car calls are aligned with the floors for which they are registered.
P	Indicates a parking call used to re-locate an "idle" car (used in up peak algorithms).

The current direction, position, speed and load is displayed above each elevator. The number of passengers queuing is displayed at each floor.

6. RESULTS

Once the simulation is complete, a print preview of the data and results are displayed as shown in Figure 5. Results include:

- average waiting time, longest waiting time, and a plot of the waiting time distribution.
- average transit time, longest transit time, and a plot of transit time distribution.

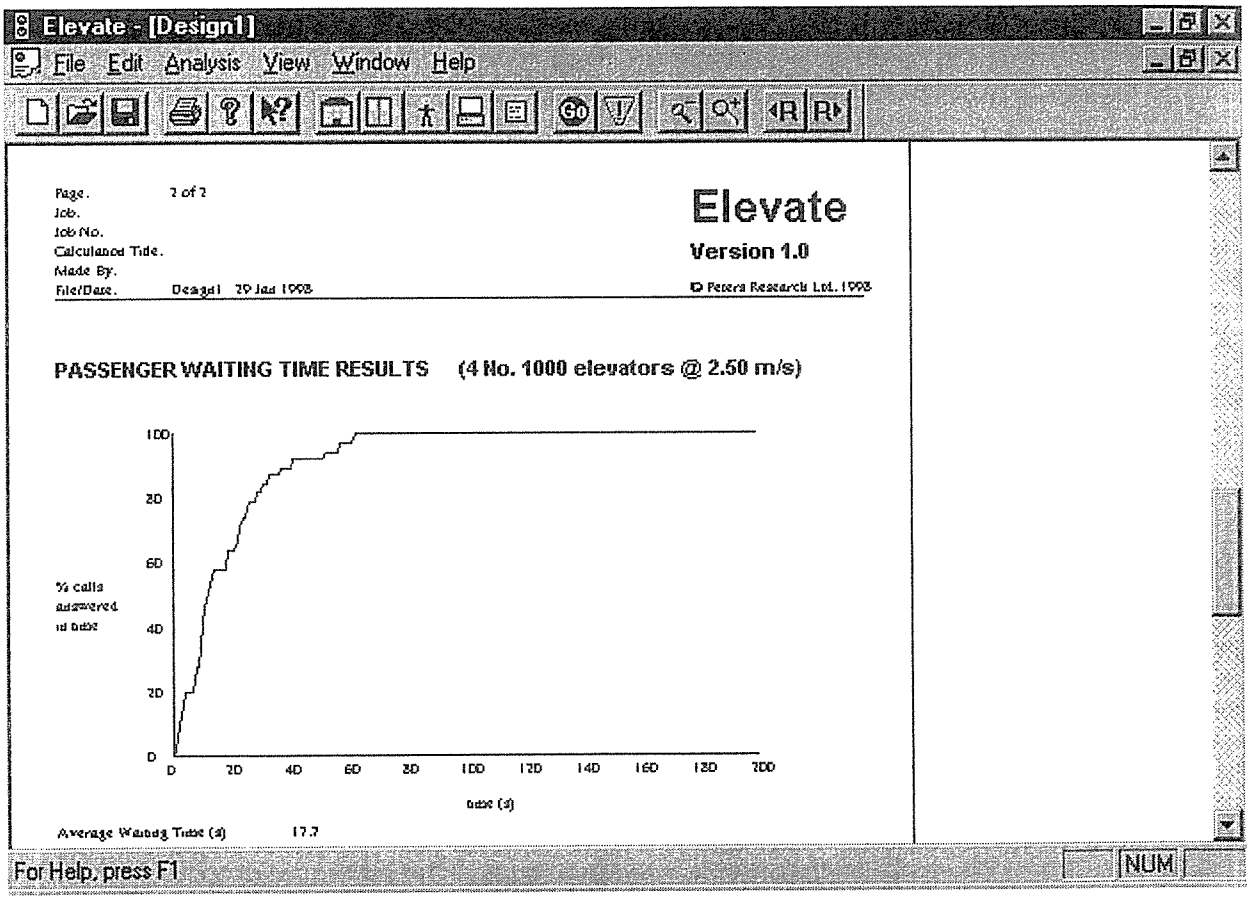


Figure 5 Results displayed on screen

In addition to these results, a file is written to disk that includes details of every passenger generated by *Elevate*: what time they arrived, at which floor, what was their destination, what were their individual waiting and transit times, etc. This file can be loaded into a spreadsheet for further analysis if required.

7 APPLICATIONS

7.1 Control System Design

The object oriented nature of *Elevate* makes it relatively straightforward to design new control systems by revising the *dispatch* object.

The *dispatch* object has to review new landing calls, and allocate them to an appropriate elevator. The elevators can be queried to find out their position, direction of travel, load, etc. For example, given an array of elevator objects, *e*

```
//the current position (m above reference) of elevator 2 is
e[2].m_CurrentPosition;

//the direction (where -1 down, 0 none, 1 up) of elevator 2 is
e[2].m_Direction;

//this allocates an up landing call to elevator 2 for floor 5
e[2].m_UpLandingCalls[5]=1;
```

Once calls have been allocated to an elevator, the elevator will answer the landing calls in the right order, and accept car calls from the passengers. These and all the other features of *Elevate* are available to the new control system without additional programming.

The result of this approach is that new control systems can be implemented in Elevate in as little as 300 lines of software code.

Elevate has been used to develop and test a range of control systems. Its application in the research and development of energy saving, green elevator control systems is discussed in [3].

7.2 Traffic Analysis

Elevate can be used to analyse the elevator performance of all building types including offices, hotels, hospitals, shopping centers, flats, warehouses, etc. Unlike conventional "up peak" round trip time calculations, you are not restricted to passengers loading at the ground floor; all types of passenger and goods loads can travel between all floors.

An example print out of data/results in Appendix 1 shows how *Elevate* can be used to model the performance of a elevator system in a shopping center. Note that in a shopping center:

- people will be arriving at all floors
- some passengers will be alone, others may be with children in buggies. By entering passenger traffic in "periods" *Elevate* allows you to have different types of passengers

using the elevators at the same time. In this example, an adult with a child in a buggy can be set to take more room in the elevator, and to take longer to get into and out of the elevator.

8. CONCLUSIONS

Elevate has been designed as a development platform for elevator control systems and as an advanced traffic analysis tool.

Elevate is written in Microsoft Visual C++. It uses object oriented techniques, breaking down the programming tasks into classes. These classes represent objects (e.g. *elevator, person, building*) which are straight forward to conceptualise, and therefore easier to work with.

The interface is Windows based. The user enters data about the system into dialog boxes titled: *Building data*, *Elevator data*, *Passenger data*, *Simulation data* and *Job data*.

The program performs a time slice simulation, providing a graphical representation of the elevators as they serve the passenger calls. Once the simulation is complete, *Elevate* displays results on screen in a print preview format. These results include details of input data, waiting times and transit times.

REFERENCES

1 Perry G, Ross J *Visual C++ By Example* (Indianapolis: Que Publishing) (1994)
2 Barney G C, dos Santos S M *Elevator Traffic Analysis Design and Control* (London: Peter Peregrinus) 2nd edition (1985)
3 Peters R D, Mehta P *Green Lift Control Strategies* The International Journal of Elevator Engineers, Volume 2 (1998)

BIOGRAPHY

Dr Richard Peters has a degree in Electrical Engineering, and was awarded a Doctorate for his research thesis *Vertical Transportation Planning in Buildings*. He pursues a broad range of professional interests including Mathematical Modelling, Computing, Vertical Transportation, and Environmental Engineering. He began writing traffic analysis software in 1987 and has subsequently developed a range of analysis techniques and software programs which are applied internationally. He is a Director of Peters Research Ltd.

Page:	1 of 2
Job:	APPENDIX 1
Job No:	Simulation for Control System Design and Traffic
Calculation Title:	Shopping Centre with Car Park
Made By:	rdp
File/Date:	Example 3.elv 31 Jul 1998

Elevate
Version 1.1

BUILDING DATA

Floor Name	*Floor Level (m)*
Mall 1	0.00
Mall 2	3.80
Mall 3	7.60
Park 1	11.40
Park 2	15.20

ELEVATOR DATA

	Car 1	*Car 2*	*Car 3*	*Car 4*
Capacity (kg)	1600	1600	1600	1600
Speed (m/s)	1.00	1.00	1.00	1.00
Acceleration (m/s/s)	0.40	0.40	0.40	0.40
Jerk (m/s/s/s)	0.80	0.80	0.80	0.80
Home Floor	Mall 1	Mall 1	Mall 1	Mall 1
Motor Start Delay (s)	0.50	0.50	0.50	0.50
Door pre-openning time (s)	0.00	0.00	0.00	0.00
Door open time (s)	1.80	1.80	1.80	1.80
Door close time (s)	2.90	2.90	2.90	2.90
Door dwell 1 (s)	6.00	6.00	6.00	6.00
Door dwell 2 (s)	3.00	3.00	3.00	3.00

PASSENGER DATA (Period 1)

Start Time	0:00
End Time	0:05
Loading Time (s)	1.50
Unloading Time (s)	1.50
Passenger Mass (kg)	75.00
Loading Threshold (%)	60.00
Notes	Passengers

Floor Name	*Arrival Rate (Persons /5 mins)*	*Dest. Prob Mall 1 (%)*	*Dest. Prob Mall 2 (%)*	*Dest. Prob Mall 3 (%)*	*Dest. Prob Park 1 (%)*	*Dest. Prob Park 2 (%)*
Mall 1	20.00	0.00	25.00	25.00	25.00	25.00
Mall 2	20.00	25.00	0.00	25.00	25.00	25.00
Mall 3	20.00	25.00	25.00	0.00	25.00	25.00
Park 1	15.00	33.30	33.30	33.30	0.00	0.00
Park 2	15.00	33.30	33.30	33.30	0.00	0.00

PASSENGER DATA (Period 2)

Start Time	0:00
End Time	0:05
Loading Time (s)	2.50
Unloading Time (s)	2.50
Passenger Mass (kg)	150.00
Loading Threshold (%)	50.00
Notes	Adult with child in pram/buggy

Floor Name	*Arrival Rate (Persons /5 mins)*	*Dest. Prob Mall 1 (%)*	*Dest. Prob Mall 2 (%)*	*Dest. Prob Mall 3 (%)*	*Dest. Prob Park 1 (%)*	*Dest. Prob Park 2 (%)*
Mall 1	7.00	0.00	25.00	25.00	25.00	25.00
Mall 2	7.00	25.00	0.00	25.00	25.00	25.00
Mall 3	7.00	25.00	25.00	0.00	25.00	25.00
Park 1	5.00	33.30	33.30	33.30	0.00	0.00
Park 2	5.00	33.30	33.30	33.30	0.00	0.00

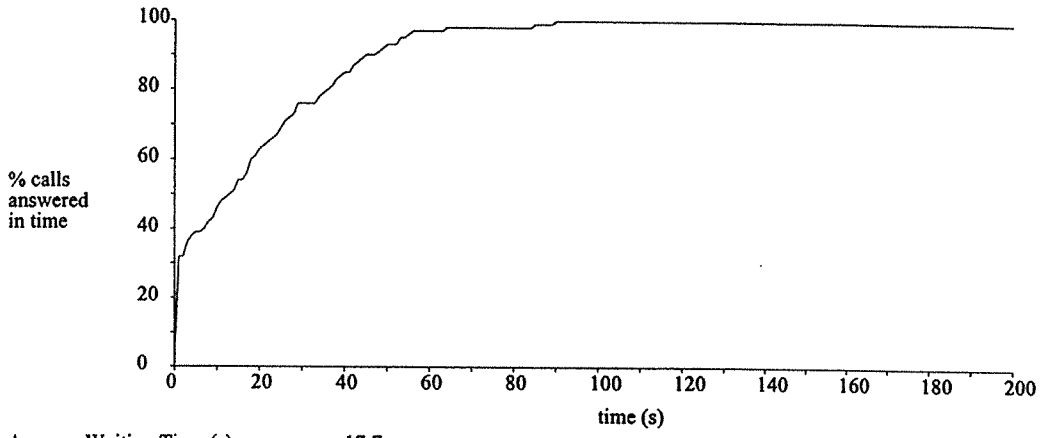
SIMULATION DATA

Dispatcher Algorithm	Group Collective
Time slice between simulation calculations (s)	0.01
No of time slices between screen updates	10

Page:	2 of 2
Job:	APPENDIX 1
Job No:	Simulation for Control System Design and Traffic
Calculation Title:	Shopping Centre with Car Park
Made By:	rdp
File/Date:	Example 3.elv 31 Jul 1998

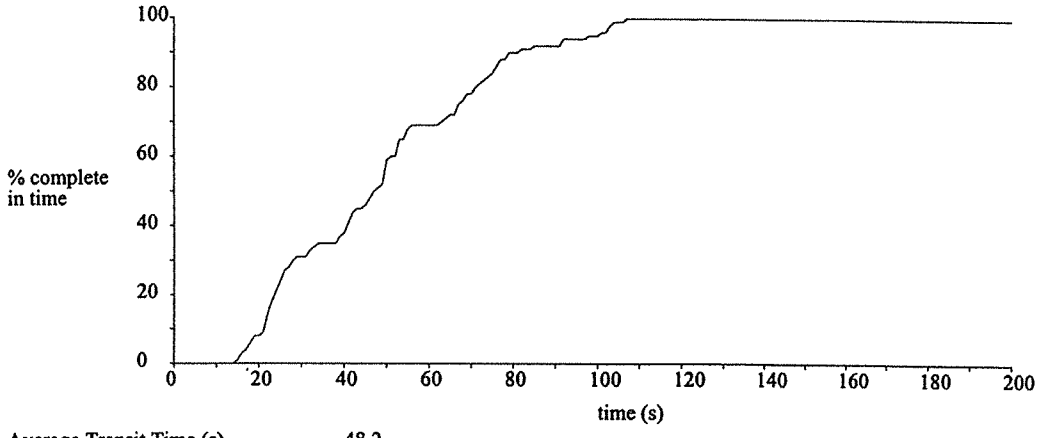
Elevate
Version 1.1

PASSENGER WAITING TIME RESULTS



Average Waiting Time (s)	17.7
Longest Waiting Time (s)	89.6

PASSENGER TRANSIT TIME RESULTS



Average Transit Time (s)	48.2
Longest Transit Time (s)	106.1