The University of York Department of Computer Science

Submitted in part fulfilment for the degree of BEng.

Algorithms for Two-Cars-in-One-Shaft Elevator Control

Jonathan Lyon

6th May 2014

Supervisor: Alan Frisch

Number of words = Word Count, as counted by `wc -w`.
This includes the body of the report only.

Abstract

Abstract to go here

Dedicating text stuff

Acknowledgements

Acknowledging stuff

Contents

Part I

Introduction

1 Introduction

Introduction will be going in here.

Part II

Literature Review

2 Chapter Placeholder

2.1 Section 1 Placeholder

2.2 Section 2 Placeholder

Part III

Elevator Simulator

3 Core Elevator Simulator

The development of the Core Elevator Simulator, as described in this chapter, was done in conjunction with another student, Craig Gosnay, who was working on a similar project to develop scheduling algorithms for elevator groups containing double-decker cars. The writing in this chapter is exclusively my own.

3.1 Introduction

A significant portion of the project was dedicated to the development of a simulation tool on which to test the performance of various scheduling algorithms. Existing literature uses a variety of tools, including one professionally available simulator named Elevate, but none of these tools were available at low cost for use in this project. There were some open-source simulation tools available online, but in general they did not support the specific case of two-cars-in-one-shaft (TCOS). It was decided that, instead of attempting to understand and extend an existing tool, I would develop my own from scratch. I felt that, given the various different methods available of representing and abstracting the specific physical domain, building my own solution would be a good way to ensure that the design decisions and assumptions made were reasonable, as well as being a useful way to consolidate my own understanding of the problem domain.

Following discussions with my supervisor, Alan Frisch, it was agreed that I would work together with Craig Gosnay to build the Core Elevator Simulator. Once this was complete, Craig and I would go our separate ways to extend the Core simulator to meet the requirements of our specific scenarios (in my case TCOS). This decision was made for two main reasons: firstly, that it would expedite the development process, allowing Craig and I to both move on to algorithm development for our respective scenarios; secondly, that it would allow our two specific solutions to be merged into one general-purpose simulator for use in future projects at the university (following the completion and submission of our own projects). I would like to thank Craig for his contributions to the Core Elevator Simulator.

3.2 Requirements

While Craig and I discussed the requirements informally together, the codification and justification printed here are my own.

3.2.1 Specification of Requirements

The following is a list of requirements that had to be met by the Core Elevator Simulator:

Functional Requirements

1. To be able to simulate the use, and evaluate the performance, of algorithms designed for the allocation of calls to cars in traditional elevator systems (i.e. those where one shaft contains exactly one car). This includes those algorithms that are described in existing literature. The evaluation objectives must include:
   a) Average time spent waiting for a car to arrive (from making a call)
   b) Average time taken to get to destination (from making a call)
   c) Average *squared* time spent waiting for a car to arrive
   d) Average *squared* time taken to get to destination
   e) Longest time spent waiting for a car to arrive
   f) Longest time taken to get to destination
2. The elevator groups used within the simulator must be user-configurable. The parameters available to the user must include the following:
   a) Number of shafts
   b) Range of floors (per shaft)
   c) Maximum capacity of cars (per car)
   d) Maximum speed of cars (per car)
   e) Acceleration rates of cars (per car)
   f) Deceleration rates of cars (per car)
   g) Time taken for doors to open (per car)
   h) Time taken for doors to close (per car)
   i) Time taken for car to change operating direction (per car)
3. The behaviour of passengers used within the simulator must be user-configurable. The parameters available to the user must include the following:
   a) Time taken to board a car
   b) Time taken to alight a car
4. At the choice of the user, the simulator must be able to generate a new passenger arrival schedule (passenger distribution) or load an existing one for use during simulation.
5. New passenger distributions must be generated with respect to the Poisson distribution, with the mean arrival rate specifiable by the user. A separate mean should be provided for each combination of origin and destination floors in the system.
6. Passengers should arrive in Passenger Groups made of one or more Passengers. Passenger Groups should be treated atomically by the allocation system, and never split up. The size of Passenger Groups should also be generated in terms of a Poisson distribution.
7. The system should be able to simulate Destination Control systems, as well as those without Destination Control.

Non-Functional Requirements

8. The code must be written in a generally maintainable fashion. This requirement entails the following:
   a) Classes, methods, etc. must be properly documented using appropriate comments for the chosen language
   b) Class, method and variable names must follow established conventions for the chosen language
9. Furthermore, this Core simulator must be built in such a way as to facilitate extension for use with non-traditional elevator systems. Specifically:
   a) Those where single shafts may contain two independent cars
   b) Those including double-decker cars, that serve two floors at once
   c) Systems including combinations of traditional cars, double-decker cars and shafts with two independent cars
10. The simulator should output appropriate logs, which can be used for validation and troubleshooting purposes.
11. It should be easy to configure the simulator for a specific scenario, and to replicate a simulation run. Thus, the configuration of a simulation should be stored in some file that can be loaded by the tool.
12. All files used to configure simulation runs must be formatted in such a way that a technically-aware user can easily modify the configuration.

3.2.2 Justification of Requirements

The requirements above can be justified as follows.

The various evaluation objectives specified in requirement 1 are provided to allow for scenario-specific interpretation of the results. For example, previous literature has demonstrated that in some cases one algorithm can outperform a second algorithm in terms of objective a (average waiting time), while the second can be better in terms of objective b (average time to destination) **[CITATION]**. Clearly the decision as to whether the first or second algorithm is better depends on which of the objectives is deemed to be most important in the scenario. Objectives c and d are provided because they penalise times superlinearly, thus preferring a narrower spread of results over a broader one where the raw averages are similar. The provision of objectives e and f (when combined with a and b respectively) also allows the user to make inferences about the spread of results, but has the advantage that the user is dealing with intuitively meaningful figures, whereas the figures given by c and d (though more useful for blind ranking of algorithms) are somewhat meaningless without some manipulation. In addition to these reasons, all evaluation objectives are provided for consistency with existing literature.

The parameters specified in requirement 2 are required firstly to allow the general purpose simulator to be used to model many different systems as might be appropriate. For example, it is likely that the normal parameters of a TCOS installation are different to the normal parameters of a traditional installation as service speeds are likely to be more critical in

those contexts. Secondly, they are provided to allow the simulator to be used for verification of results in previous literature, which is not generally consistent about these parameters.

The parameters specified in requirement 3 are provided to allow compatibility with the different values used in existing literature.

The passenger arrival schedule (requirements 4 and 5) must be generated randomly by a poisson distribution for consistency with existing literature. The requirement to be able to load a previously generated passenger arrival schedule is to allow algorithms to be fairly evaluated against the exact same schedules. Inferences about the comparative benefits of one algorithm over another should be based upon samples of trials against multiple different schedules to avoid anomalous results. The need for separate Poisson parameters for arrivals for each combination of origin and destination floors is to allow for different types of passenger traffic to be generated (i.e. Up-peak, Down-peak, Lunch-time, Inter-floor).

While existing literature generally only uses individual Passengers in its simulations, this simulator will use Passenger Groups consisting of multiple Passengers. This is because Destination Control systems allow a group of passengers to come to the console and specify that they are a group of n people, and they must then be kept together for their journey. Two people might arrive at the same place with the same destination but not be part of the same group, in this case they will be treated as independent passengers in the simulation.

3.3 Technology Decisions

It was decided that the Core Elevator Simulator would be developed in C#, taking advantage of the C# experience gained on previous projects by both me and Craig. There were several reasons for this. Firstly, the elevator domain can be very conveniently represented in an object-oriented fashion. Secondly, it was considered important that we use a strongly-typed compiled language to minimise the risk of undetected errors in the code. Thirdly, we felt that some features of the C# syntax would come in very useful during the project, specifically Properties and LINQ expressions. Finally, we did not consider that the benefits of other languages (e.g. Java) such as cross-platform compatibility and being open-source were relevant to our project.

We decided that configuration files should use an XML format, as this is widely understood by the Computer Science community, and because there are built-in libraries for handling XML files within C#.

3.4 Architecture

The first architecture decision made was whether to use a discrete event-based simulator or a 'continuous' time-sliced simulator **[Have these concepts been introduced (in lit review)? Have citations been given?]**. It was initially assumed that a time-sliced one would be used since it would provide a more conceptually accurate model of the real world domain and thus be convenient to understand while developing and maintaining the tool. However, it was noted that this method had the problem of being very time-consuming and resource-intensive. For example, if the model were to enter an idle state, it would be necessary for the simulation to work through every time slice waiting for some event to happen (i.e. the arrival of a passenger) that would cause the state of the system to change. This would seem to be a very inefficient use of resources.

The event-driven approach would overcome this issue by moving the simulation time directly from one event to the next, only needing to perform computations at times when the state of the system changed. However, this added the complication that the domain system, which is inherently continuous, needed to be represented in a discrete but meaningful and practical manner. Another issue with the event-driven approach was that it would make it much harder to include a real-time visualisation of the simulation, and that such a visualisation would have to be generated retrospectively from some log of the events. However, as there was no requirement to implement any visualisation the decision was made to go ahead with an event-based simulator for its performance benefits.

The general architecture of the tool consists of two main logic components, which will be described in greater detail later on. These two components are kept largely independent, allowing easier testing of each one. The first component is the domain model. The domain model performs all computations about the behaviour of the elevator group system, including decisions about lift movements, etc. and these decisions are all based purely on the current state of each individual car and the calls that are currently allocated to that car. The second component is the scheduler. The purpose of the scheduler is simply to allocate passenger hall calls to elevator cars in the domain model. The scheduler is aware of the full state of the domain model, and may use any state information in its allocation decisions, however it cannot directly modify the state. The only other interaction between the two components is for the scheduler to allocate hall calls to specific cars within the domain model.

A third component is the agenda and main control loop. The agenda contains two types of events: those pertaining to passenger arrivals (i.e. hall calls), and those to state changes of specific cars in the domain model. The control loop pulls each event off of the agenda in order of timestamp, and triggers either the scheduler or a car in the domain model to act upon them as appropriate. Each of these events is handled synchronously; all of the computation pertaining to one event occurs before the next event is taken from the agenda.

The final notable component is the passenger distribution generator which, as per the requirements, generates passenger arrival schedules based on Poisson distributions as configured by the user.

Other uninteresting components include the simulation configuration loader, which parses the XML simulation configuration file, and the logger which writes content into a log file and (optionally) prints them to the console at run time.

A significant decision had to be made about how different schedulers could be written and used with the simulator. One possibility was that scheduler methods could be written separately and compiled to Dynamic Linked Libraries (DLL), which could then be dynamically pulled into the simulator at runtime, but this had the problem that the schedulers would then not be able to have any memory of their own, which might be desirable for algorithms involving learning. Another possibility was to do it the other way around – to have the scheduler pull the simulation logic in as a DLL – but then the memory of the domain model would have to be passed around via the scheduler all the time, breaking some of the cohesion and making the domain logic slightly harder. After discussion with Alan Frisch, it was decided that, for the time being, the different schedulers would simply be built into the same executable as the rest of the simulator and, should another scheduler be added, the system would need to be recompiled.

3.5 Domain Assumptions

In order to simplify the logic used in the implementation of the simulator, the following assumptions were made about the domain:

- Passengers will not board the lift while the doors are opening and closing.
- Passengers will travel to the same destination as originally given; they will not change their minds.
- Passenger Groups will not split up.
- Passengers will board the first available car travelling in their destination that can accommodate them, except in Destination Control systems when they will board the car to which they have been allocated when it is travelling in their direction. **[Could this be written better?]**
- All cars in an elevator system can call at all floors between the minimum and maximum floor specified.
- If the lowest floor is l and the highest floor is h, then there are $(l - h) + 1$ floors in total, represented by the integers in the range $[l, h]$.
- All cars accelerate and decelerate constantly between 0 and their maximum speed.
- Acceleration and deceleration rates and travel speeds are not affected by parameters such as location in the shaft, direction of travel or current load.
- All passengers take the same amount of time to board and alight cars. The time taken for n passengers to board or alight increases linearly with n.
- The distance between consecutive floors is the same throughout the building.
- Passengers will not attempt to travel from one floor to the same floor (origin and destination floors will not be equal).
- Continuous allocation algorithms will only revoke previous call allocations when new calls come in (they will not generally monitor the state of the system except at times when they are invoked by new passengers making hall calls). **[Is this clear?]**
- In a Destination Control system, if a Passenger Group is assigned to a car which turns out not have enough capacity to accommodate the group when it arrives, the entire group will wait at the floor until the same car comes back again. Schedulers should not allocate passengers to cars which cannot accommodate them.
- Passenger Groups will not contain more people than can be accommodated by the smallest car when it is otherwise empty.

3.6 Detailed Description of Components

3.6.1 Domain Model Component

As previously described, the domain model component contains a conceptual model of the elevator group, as well as the logic used to control the movement of each car. Each car maintains its own list of allocated calls, and all movement decisions are made based on the calls on the call list. The calls list is held in three groups:

- P1 (First passage) – calls that can be served by the car without reversing.
- P2 (Second passage) – calls that require the car to reverse once before serving.
- P3 (Third passage) – calls that require the car to reverse twice before serving.

Figure 3.1 shows how hall calls are allocated to the groups P1, P2 and P3. The car is currently stationary at floor 3, going up. Upward hall calls from floor 3 or above can be served without reversing. Downward hall calls require the car to reverse once (possibly having gone up first to reach it). Upward hall calls from below floor 3 require the car to reverse, travel down and reverse again. Car calls are allocated to the same groups, but are only placed in either P1 or P2 as they do not have an associated direction.

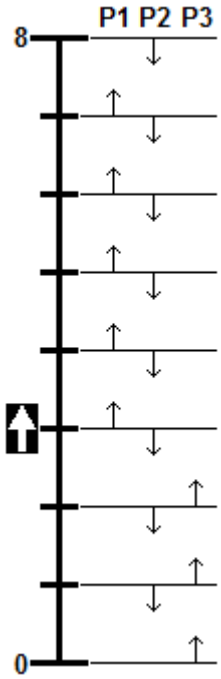

Figure 3.1: Grouping of Hall Calls to P1, P2 and P3

The behaviour of cars is governed precisely by the following three rules:

- If a passenger in an elevator car has requested to get off at a particular floor, the car must not pass that floor without stopping.
- If a hall call has been allocated to a car, the car must not pass the floor of the call in the direction of the call without stopping.

- If there are passengers in a car, the car must not change direction.

Control Logic

Since the decision was made to use a discrete event-based simulation, it was necessary to consider how the movement of the cars, which is inherently continuous, could be represented meaningfully in a discrete manner. While developing a dynamic programming approach to elevator scheduling, Nikovski and Brand [REF] present a discretised model of the state space of an elevator car in terms of its position in the shaft, which has been adapted for use in this simulator. If one considers the two parameters of the continuous state space to be current speed and vertical position in the shaft then, given that the car will have some pre-determined maximum speed and rates of acceleration and deceleration, it can be seen that the car will follow certain fixed paths around this state space.

Figure 3.2 gives an example of a possible state space for an upward-travelling car, where the red dots represent the states, the green lines the transitions between states, and the blue lines the paths of deceleration from each state to its associated floor. Each state can be thought of as a decision-point for its associated floor, at which the car decides whether to start slowing down in order to stop at the associated floor, or to carry on travelling upwards to the next decision point). In this example, the floors are spaced evenly, but one can imagine how the states might look otherwise. In any case, since each state lies on one of the blue deceleration paths, the parameters of a state can be determined from the floor with which it is associated, the movement speed associated with the state, and the specified rate of deceleration of the car. A similar diagram could be drawn for a downward-moving car.

As well as the floor, speed and direction, the state of the car contains two other parameters. These are an action and a Boolean representing whether or not the doors are open. While the latter is self-explanatory, the car action parameter requires some explanation. The ten possible actions are defined in an enumeration called CarAction, and are described below:

- Leaving – The car state has this action when the car is accelerating away from having stopped at one floor and moving towards the decision point for the next floor.
- Moving – This action is when the lift is passing a floor; it is moving, but it has not just left a floor, neither is it stopping at one.
- Stopping – This action is when the lift has passed a decision point and decided to stop at the associated floor.
- DoorsOpening – The car state has this action during the time that the car is opening its doors.
- Unloading – This action is used for the time when passengers are getting out of the car.
- Reversing – This action is used for the delay caused by a car configuring itself to move in the opposite direction. The time of this action can be set to zero if changing the direction of a car incurs no delay, but previous literature assumes some delay will be caused.
- Loading – Used during the time when passengers are boarding the car.

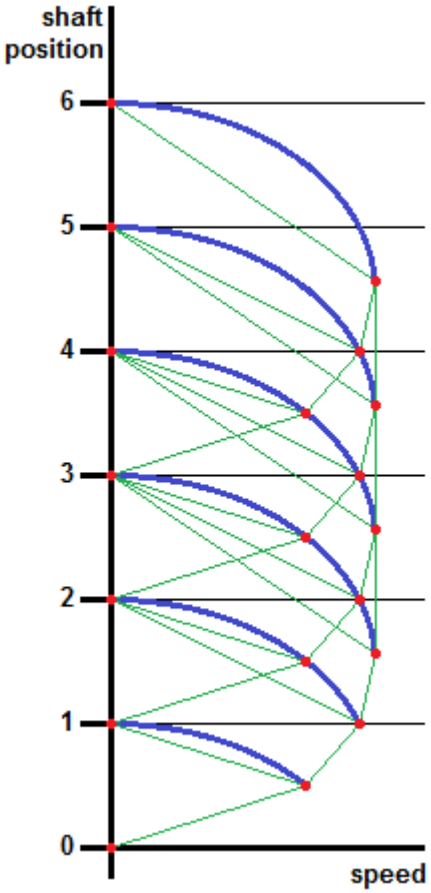

Figure 3.2: The discretised state space of an upward-travelling elevator car

- DoorsClosing – Used during the time when the car is closing its doors.
- Idle – None of the above actions are currently taking place.
- Stopped – This is something of a placeholder action, in that it does not represent any particular domain concept. However, it does hold the logic to decide which of the above actions to enter next (as shown in Figure 3.3).

Figure 3.3 demonstrates the relationship between the ten car action states. In this diagram, solid transition edges denote immediate action changes, while dotted lines denote state changes that are made via the agenda. For example, if the car is in the Stopped state, and determines that the next required action is to reverse, it will immediately change its state to Reversing. Since there must then be a delay before the car can do anything else (specified by the user as, for example, 1 second), it will add an event on the agenda for 1 second later. This agenda event will tell the car to go back into the Stopped state. Then it might determine that the next required action is to load passengers, so it will immediately change into the Loading state. It will then calculate the amount of time that loading will take (based on the number of passengers to load, and the user-specified loading time per passenger) and add an event onto the agenda for when the car should re-enter the Stopped state. Putting

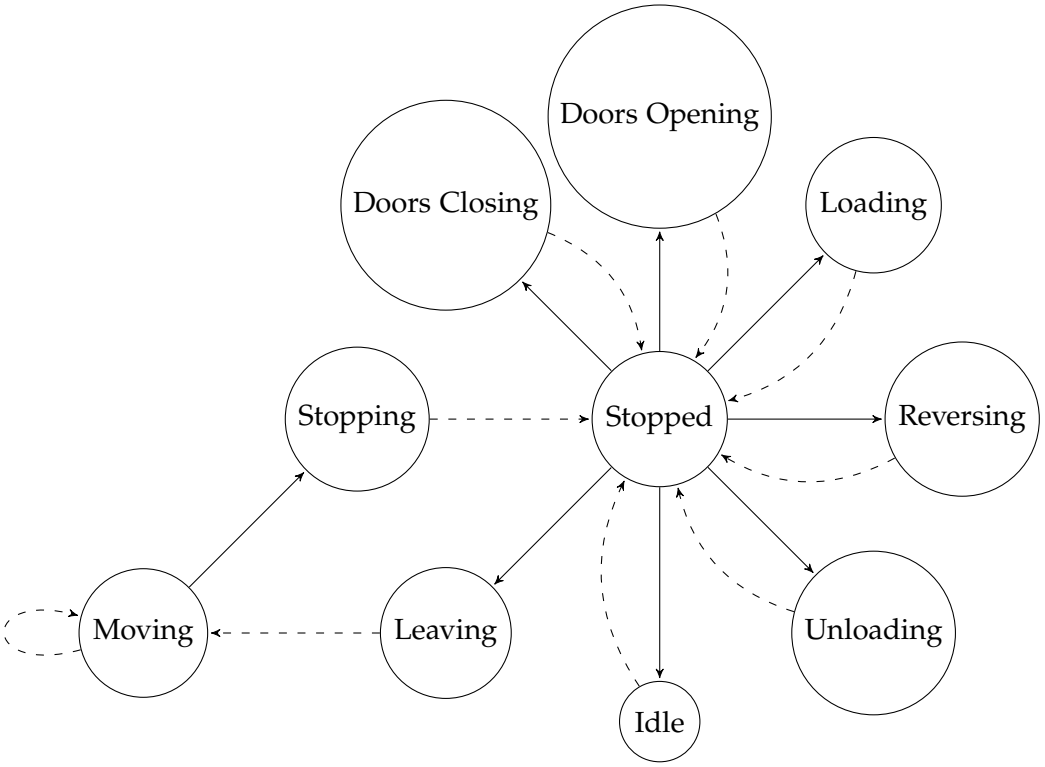

Figure 3.3: A transition diagram of the car action element of a car state

future events onto the agenda allows the immediate control to return to the main control loop, which can (if appropriate) serve Passenger Hall Call events and/or events for other cars in the mean time.

Movement Mathematics

When the car is in one of the Leaving, Moving and Stopping action states, the car must work out the parameters of its next state. If the current state is Leaving or Moving, then the next state will be a decision point for whether or not to stop at the next floor. If the current state is Stopping, then the next state will be Stopped at the next floor. In each case, we know all of the following information:

- U The current speed (ms^{-1})
- F The current floor
- G The next floor
- S The distance to the next floor (m)
- A The rate of acceleration of the car (ms^{-2})
- D The rate of deceleration of the car (*positive* ms^{-2})
- M The maximum speed of the car (ms^{-1})

We are required to find the following parameters, which define our next state:

- V The speed of the car when it reaches that state (ms^{-1})
- T The time taken to reach the state (s)

In each case, we use the following basic equations of motion:

$$v = u + at \tag{3.1}$$

$$v^2 = u^2 + 2as \tag{3.2}$$

$$s = ut + \frac{1}{2}at^2 \tag{3.3}$$

In these equations, the variables are generically defined as follows:

- t The duration of a time interval (s)
- s The distance travelled during the time interval (m)
- u The speed at the beginning of the time interval (ms^{-1})
- a The rate of acceleration (ms^{-2})
- v The speed at the end of the time interval (ms^{-1})

Once V and T have been found, the next car state will be placed on the agenda for T seconds from now.

Here, I will demonstrate the maths for upward movement. The maths for downward movement is analogous.

Moving state The Moving state applies when the car is moving, has reached a decision point, and has decided not to stop. The parameters of the next decision point must be calculated, and there are two scenarios that must be considered.

The first, and simplest, scenario is where the car will accelerate until it reaches the decision point for the next floor, then decide whether or not to start slowing for the floor. In this case, we simply need to find the point Q at which the acceleration curve from our current state crosses the deceleration curve for the next floor (Figure 3.4).

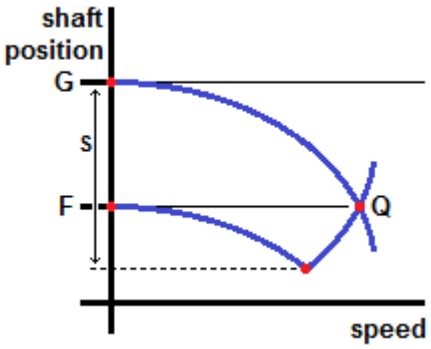

Figure 3.4: Simple case while moving

We first find the distance from here to point Q, which we will call Q_D. By two applications of equation 3.2 we find that:

$$Q_V^2 = U^2 + 2AQ_D \text{ (for the acceleration curve)}$$

$$\text{and } 0 = Q_V^2 - 2D(S - Q_D) \text{ (for the deceleration curve).}$$

Note that Q_V is the speed that we will have at point Q. Combining these two equations we find that

$$Q_D = \frac{2DS - U^2}{2(A + D)}$$

It is now trivial to find V and T as required, since

$$V^2 = Q_V^2 = U^2 + 2AQ_D \text{ from equation 3.2}$$

$$\text{and } T = Q_T = \frac{Q_V - U}{A} \text{ from equation 3.1.}$$

The second scenario is invoked when we find that Q_V exceeds the maximum speed M that our car can travel at. In this case we will accelerate to some point P on the acceleration curve (at which $P_V = M$), then move at a constant speed of M until we reach the decision point R, the latest point at which we can decelerate to the next floor from speed M (Figure 3.5).

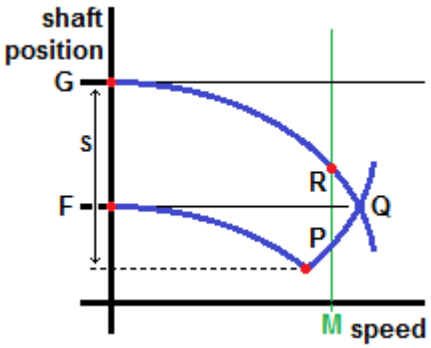

Figure 3.5: Complex case while moving

The parameters of the point P on our acceleration curve are trivial to find:

$$P_V = M \text{ by definition,}$$

$$P_T = \frac{P_V - U}{A} \text{ from equation 3.1}$$

$$\text{and } P_D = UP_T + \frac{1}{2}AP_T^2 \text{ from equation 3.3.}$$

Then, the distance from our initial point to point R, named R_D, can be found from the following equation

$$0 = R_V^2 - 2D(S - R_D) \text{ from equation 3.2,}$$

which rearranges to give

$$R_D = S - \frac{R_V^2}{2D}.$$

Then, since the movement from P to R is constant at speed M the required values V and T can be calculated as

$$T = R_T = P_T + \frac{R_D - P_D}{M}$$

$$\text{and } V = R_V = M.$$

Note that the above logic and equations also hold for the common case where the current speed of the car is already at the maximum speed. In such a scenario, it is correctly calculated that $P_T = P_D = 0$.

Leaving state The Leaving state applies when the car is stopped and about to start leaving. The parameters of the first decision point that will be encountered by the car must be calculated. The logic and equations for the Leaving state are the same as those for the Moving state, but if the car is in the Leaving state then $U = 0$. Figure 3.6 and Figure 3.7 demonstrate these scenarios graphically.

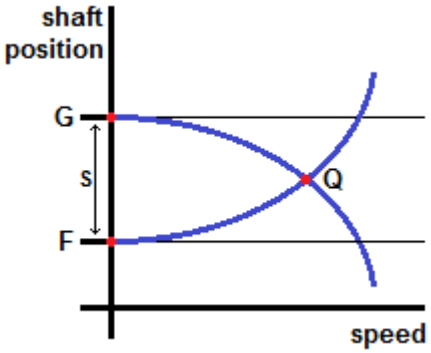

Figure 3.6: Simple case while leaving

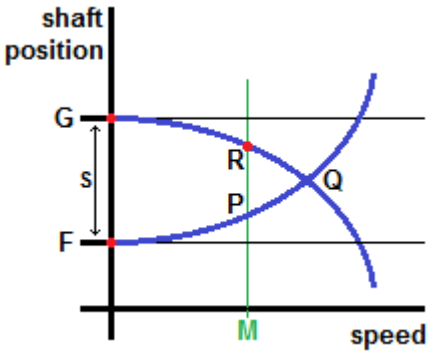

Figure 3.7: Complex case while leaving

Stopping state The Stopping state applies when the car is moving, has reached a decision point, and has decided to stop. If the car is in the Stopping state, the maths is very simple. The next state is reached by following the deceleration curve straight down to stopping at the floor. Therefore,

$$V = 0 \text{ by definition of Stopping}$$

$$\text{and } T = \frac{U}{D} \text{ from equation 3.1.}$$

3.6.2 Scheduler Component

The Scheduler component in the Core Elevator Simulator consists of an Interface class, named IScheduler, defining the methods that a Scheduler implemented by the user must include. In fact, the IScheduler interface is very simple and contains just one method which is called by the main control loop when a new PassengerGroup arrival is taken from the agenda:

```
interface IScheduler
{
        /// <summary>
        /// Allocates a given group of passengers to a specific car within
        /// the given building, based on the relevant allocation method
        /// </summary>
        /// <param name="group">The PassengerGroup to be allocated</param>
        /// <param name="building">The Building holding the domain model</param>
        void AllocateCall(PassengerGroup group, Building building);
}
```

There is also an enum class, SchedulerType, which defines the names of all of schedulers. Then, there is a static method which takes a specific SchedulerType enum value and returns an instance of the relevant Scheduler as implemented by the user.

In order to demonstrate the ease with which a new Scheduler can be added, I will include an example of a very simple scheduler, the ClosestCar scheduler, which simply allocates each call to whichever car is closest to the call at the time the call is made, ignoring the directions in which cars are moving.

Firstly, the user must create a new class, which implements IScheduler. As described, the AllocateCall method must be implemented, as follows:

```
class ClosestCarScheduler : IScheduler
{
        /// <summary>
        /// Allocates a given group of passengers to a specific car within
        /// the given building, based on the ClosestCar allocation method
        /// </summary>
        /// <param name="group">The PassengerGroup to be allocated</param>
        /// <param name="building">The Building holding the domain model</param>
        public void AllocateCall(PassengerGroup group, Building building)
        {
                // compile list of all cars
                List<ICar> cars = new List<ICar>();
                building.Shafts.ForEach(s => s.Cars.ForEach(c => cars.Add(c)));
                // get closest car
                var car = cars.OrderBy(c => Math.Abs(c.State.Floor - group.Origin)).First();
                // allocate call
                car.allocateHallCall(new HallCall(group));

                // update the PassengerGroup data with the time at which
                // the hall call was made
                group.changeState(PassengerState.Waiting, Simulation.agenda.getCurrentSimTime());
        }
}
```

Secondly, the name of the Scheduler must be added to the SchedulerType enum:

```
enum SchedulerType
{
        Manual,
        Random,
        ClosestCar
}
```

Finally, the new Scheduler must be added to the mapping method:

```
public static IScheduler getScheduler(SchedulerType scheduler)
{
        switch (scheduler)
        {
                case SchedulerType.Manual:
                        return new ManualScheduler.ManualScheduler();
                case SchedulerType.Random:
                        return new RandomScheduler.RandomScheduler();
                case SchedulerType.ClosestCar:
                        return new ClosestCarScheduler.ClosestCarScheduler();
        }
```

```
        return null;
}
```

Once the user has performed these three steps, the simulator can be recompiled and the new Scheduler can be used.

3.6.3 Passenger Distribution Generator

The purpose of the Passenger Distribution generator is to load an XML âĂŸspecificationâĂŹ file, which contains the parameters of the Poisson distributions for the various floors. Based on this, it outputs a second XML file containing a list of specific passenger groups, with their arrival times, sizes, origins and destinations. The arrival times of these passenger groups should follow the relevant Poisson distributions as taken from the specification file. An example specification file is shown below.

```
<ArrivalFloors>
        <ArrivalFloor Floor="0">
                <DestinationFloor Floor="1" ArrivalsPerMinuteMean="5.0" GroupSizeMean="2" />
                <DestinationFloor Floor="2" ArrivalsPerMinuteMean="3.0" GroupSizeMean="2" />
        </ArrivalFloor>
        <ArrivalFloor Floor="1">
                <DestinationFloor Floor="0" ArrivalsPerMinuteMean="0.5" GroupSizeMean="1" />
                <DestinationFloor Floor="2" ArrivalsPerMinuteMean="1.0" GroupSizeMean="1" />
        </ArrivalFloor>
        <ArrivalFloor Floor="2">
                <DestinationFloor Floor="0" ArrivalsPerMinuteMean=âĂİ0.3"␣GroupSizeMean="1"␣/>
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣<DestinationFloor␣Floor="1"␣ArrivalsPerMinuteMean="0.5"␣GroupSizeMean="1"␣/>
␣␣␣␣␣␣␣␣</ArrivalFloor>
</ArrivalFloors>
```

This example specification is for up-peak traffic between three floors. The first <Arrival-Floor> element specifies the mean number of arrivals per minute at floor 0 as 5 groups going to floor 1, and 3 groups going to floor 2. Similarly, the mean is 0.5 groups per minute going from floor 1 to 0, and 1 group per minute going from 1 to 2. 0.3 groups per minute go from floor 2 to 0, while 0.5 groups per minute go from 2 to 1. You will also see that the sizes of groups boarding at floor 0 are generally larger, with an average of 2 people as opposed to an average of 1 person per group for other movements.

The passenger distribution generator was run with the above specification file, and configured to create a schedule for a period of 10 minutes, from 13:00:00 to 13:10:00. An extract from the file is shown here, but the full schedule is available in the appendix.

```
<PassengerGroups>
        <PassengerGroup ArrivalTime="03/03/2014␣13:00:02" Size="1" Origin="0" Destination="2" />
        <PassengerGroup ArrivalTime="03/03/2014␣13:00:11" Size="2" Origin="0" Destination="1" />
        <PassengerGroup ArrivalTime="03/03/2014␣13:00:20" Size="1" Origin="1" Destination="2" />
        <PassengerGroup ArrivalTime="03/03/2014␣13:00:28" Size="2" Origin="0" Destination="1" />
        <PassengerGroup ArrivalTime="03/03/2014␣13:00:31" Size="2" Origin="0" Destination="1" />
        <PassengerGroup ArrivalTime="03/03/2014␣13:00:34" Size="2" Origin="0" Destination="1" />
        <PassengerGroup ArrivalTime="03/03/2014␣13:00:48" Size="2" Origin="1" Destination="2" />
        <PassengerGroup ArrivalTime="03/03/2014␣13:00:50" Size="1" Origin="0" Destination="1" />
        <PassengerGroup ArrivalTime="03/03/2014␣13:00:52" Size="3" Origin="0" Destination="2" />
        <PassengerGroup ArrivalTime="03/03/2014␣13:00:59" Size="1" Origin="0" Destination="2" />
        <PassengerGroup ArrivalTime="03/03/2014␣13:01:12" Size="1" Origin="2" Destination="1" />
        <PassengerGroup ArrivalTime="03/03/2014␣13:01:15" Size="4" Origin="0" Destination="1" />
        <PassengerGroup ArrivalTime="03/03/2014␣13:01:51" Size="2" Origin="0" Destination="1" />
        <PassengerGroup ArrivalTime="03/03/2014␣13:01:52" Size="1" Origin="0" Destination="1" />
        <PassengerGroup ArrivalTime="03/03/2014␣13:02:31" Size="3" Origin="0" Destination="2" />
        <PassengerGroup ArrivalTime="03/03/2014␣13:02:54" Size="2" Origin="0" Destination="2" />
        .
        .
        .
</PassengerGroups>
```

The format of this file is fairly straightforward; each Passenger Group in the schedule is shown as a single element in the file, with attributes storing its time of arrival, size, origin and destination. This file is loaded at the start of a simulation run and all of the Passenger Group arrivals are added to the agenda.

Generating values with a Poisson distribution

The schedule is generated by using Knuth's algorithm **[CITATION]** for Poisson distributions, which is described by the following pseudocode:

```
function PoissonDistributedRandomNumber(mean):
        L := e ^ (mean * -1)
        k := 0
        p := 1

        do while p > L:
                k := k + 1
                u := random[0, 1] // u is a random real number in the interval [0, 1]
                p := p * u

        return k - 1
```

When generating the schedule, we progress through time in steps specified by the user (in this example we will use 1 second time slices). At each time step, we calculate the mean number of passenger groups for each movement in that time step (for a 1 second step, we divide the mean per minute by 60), and then use Knuth's algorithm to determine, based on that mean, how many arrivals we will have for each movement. Then, for each arrival that we generate, we run Knuth's algorithm based on the mean group size for that movement to determine the size of the group. The user specifies a maximum group size; if the size of a group is not between 1 and the user-specified maximum, we generate another value using Knuth's algorithm.

3.6.4 Simulation Configuration

All parameters of a simulation run are specified in a single configuration file, so that this configuration can be easily stored and re-run at a later date if desired. A sample configuration file is shown in the appendix, but it includes the following parameters:

- **Scheduler** – the name of the scheduler that is to be used
- **Passenger Distribution parameters**
  - Whether to load an existing distribution file or create a new one based on a given specification file
  - Specification file path (if generating a new distribution)
  - Maximum Passenger Group size (if generating a new distribution)
  - Start time of distribution (if generating a new distribution)
  - End time of distribution (if generating a new distribution)
  - Resolution; the size of time steps (if generating a new distribution)
  - Distribution file path
- A selection of named sets of **Car Attributes**
  - Maximum capacity
  - Maximum speed
  - Acceleration
  - Deceleration
  - DoorsOpenTime – time taken to open doors

  - DoorsCloseTime – time taken to close doors
  - DirectionChangeTime – time taken to reverse
  - PassengerBoardTime
  - PassengerAlightTime

- The domain model **Building**, with its lowest and highest floors, and the distance between floors specified.
  - A selection of **Shafts**, within which
    * A selection of **Cars**, each of which is linked to a set of Car Attributes as defined above, and a CarType (always Single-decker in the Core Elevator Simulator) as well as the floor at which the car is to be initialised.

- The path to a **Log File** – the user can specify 'auto' here, in which case an automatic file name will be generated based on the current date and time.

3.7 Validation

All validation work was performed independently of Craig, as were the code corrections made in response.

The architecture of the Core Elevator Simulator allows each component to be validated separately, with only minimal verification on the ways in which they interact.

3.7.1 Domain Model Component

The most significant component to validate is the Domain Model component. In order to test it I designed a specific passenger arrival schedule, containing a total of 13 groups and 39 passengers arriving over the course of 2½minutes, that would result in each workflow within the Car domain logic being tested. The test modelled a building with just 3 floors, since this was enough to give confidence about the performance over a larger number without increasing the complexity of the hand-calculations. Since the cars do not interact with each other in the Core Elevator Simulator, I built a dummy scheduler that would assign all calls to car 0, meaning that all calls would be handled by the same car. **[These two points must be tested at least a little]**

Using Microsoft Excel to maintain a trace table, I worked through the way in which the car should behave as the different calls came in, performing all of the movement calculations and other mathematics by hand. I calculated that the simulation run with my designed passenger arrival schedule would take 4 minutes and 14 seconds, and noted down the specific boarding and alighting times of each Passenger Group. I also had a record of every state change that would take place in the car during the simulation run, which is shown in the appendix.

The test run included two scenarios where the car could not serve the calls due to lack of capacity, and it handled this correctly. It also included cases where hall calls came in as the doors were closing. The expected behaviour was for the car to reopen the doors and serve the hall calls before leaving.

There were times when the car would set off from floor 2, heading to floor 0 and then a hall call would come in at floor 1 just after it had left. In this case, the expected behaviour

was for the car to make an unexpected stop at floor 1, as long as the call came in before the car had passed the final decision point for floor 1.

Initially, when I ran the simulator for the first time on the specific passenger arrival schedule I was encouraged to see that for the first 45 seconds, the logs showed that the car was behaving exactly as I had predicted that it should in the trace table. Unfortunately, after that the timings became about half a second out, which was the indicator of some flawed logic either in the simulator itself or in my hand-calculated trace table. Having seen that the discrepancy came on the first occasion that the car was directed to pass a floor without stopping, I placed a breakpoint in that logic to find out if there was any problem.

In fact, there was a problem. I had erroneously implemented the formula for finding PD as follows

$$P_D = U + \frac{1}{2}AP_T^2,$$

when, in fact, the correct formula is slightly different, as follows

$$P_D = UP_T + \frac{1}{2}AP_T^2.$$

After I had corrected this implementation, I tried the simulation run again and this time found that everything was behaving as expected. All of the car state changes happened at the right time (within the 10-millisecond accuracy to which I had done my hand calculations), and all of the passenger boarding and arrival times were as expected too. I am now confident that the Domain Model component of the Core Elevator Simulator behaves correctly.

3.7.2 Scheduler Component

Since the Scheduler component will be re-implemented for each scheduling algorithm that is run on the simulator, it doesnâĂŹt really make sense to completely validate it at this stage. However, in order to test the general interactions between the Scheduler and the Domain Model component I have used the basic ClosestCar scheduler as described above. I then used a Manual scheduler, which asks the user to allocate calls to cars, to make the decisions in my head (in the way that I believe the ClosestCar scheduler should allocate them) and ensure that the behaviour of the simulator when using the ClosestCar scheduler matches the behaviour when I make the decisions myself.

After running the simulation in both scenarios (with the same passenger arrival schedule as used in the domain model component tests), I found that the logs were identical. Therefore, I am confident that the scheduler component is behaving as it should. It would not be useful to print the entire log here, but I will print the evaluation results which can be used for later verification if necessary:

```
Average waiting time:               4.92269230769231
Average squared waiting time:       126.226432384615
Average time to destination:        22.3131538461538
Average squared time to destination: 782.275027153846
Longest waiting time:               38.249
Longest time to destination:        66.667
```

When the simulator is used with new Scheduling algorithms, care should be taken to ensure that the new Schedulers have been implemented correctly, and tests should be performed to validate the implementations.

4 TCOS Elevator Simulator

Part IV

Algorithms for TCOS Elevator Systems

Bibliography