# Lift System – Z Modelling Report

Buster Major | bmaj406 | 619898106

## 1 Introduction

This report details the modelling process of a simple lift system in a formal specification language. The model has been built in the Z formal specification language, and it simulates some of the basic functionality like requesting lifts from floors, requesting floors from inside lifts, and lifts moving between floors. A set of requirements for the system have been defined which have beet translated into Z-based schemas. A set of theorems proving properties about the system have also been provided. Finally, I reflect on the project and its impact on my learning.

## 2 Requirements

This section details the requirements used to shape the lift system model. An overview of the model's functionality will be detailed, then requirements of specific model schemas will be described.

### 2.1 Lift System

The lift system modelled will be assumed to be able to cater for an arbitrary number of lifts and floors. It is assumed every lift will be able to access every floor, and that no special permissions like swipe card or keys are required to access certain floors.

Floors will have an arbitrary label associated which may not map to the physical height index of the floor (for example 'B', 'G', 'M', '2', etc). The floors are assumed to be stacked vertically, so it is possible to assign the lowest floor an index of 1, the next lowest an index of 2, etc. Floor indexes are assumed to map one-to-one with floor labels, i.e. floor labels must be unique. Furthermore, each lift has a set of lift buttons which may be labelled as any combination of 'up' and 'down'. The bottom-most floor of the lift system only has an 'up' button, and the top-most floor of the lift system only has a 'down' button. All lifts in between have both 'up' and 'down' buttons. If the bottom-most floor is the same as the top-most floor (i.e., the lift system only caters for one floor), the floor will have no buttons.

The lift will consist of a set of doors which can be either opened or closed, and a set of inner lift buttons which map one-to-one with every floor of the lift system by floor label. The lift will be able to move sequentially between floors, but it is only able to move if its doors are closed. It is assumed the doors only close on an explicit door close request from a user. When a lift reaches a floor, it is assumed its doors will automatically open.

Jobs are submitted to lifts through either floor buttons or lift buttons. A lift can either be completing an 'upwards' set of jobs or a 'downwards' set of jobs. This restricts a lift to a subset of floors it can possibly visit while its queue is populated to ensure efficiency in the lift system. If a lift is completing an 'upwards' set of jobs, it can only take on requests for floors that are above the lift's current position, to ensure passengers experience minimal ride times. Similarly, if a lift is completing a 'downwards' set of jobs it can only take on jobs below its current position. Each lift will have an ordered queue of jobs, which stores the index of the floors requested and ordered in ascending order if the lift is completing an upwards set of jobs and descending if the lift is completing a downwards set of jobs. The lift's next destination is at the head of its job queue. A lifts job directional status is considered idle if no jobs are in its job queue.

When a floor button is pressed, a set of rules determine which lift is assigned to going to the floor. The set of lifts considered are any lift that is idle, and any lift that is both : 1) completing a job sequence that is in the same direction as the pressed floor button, and 2) does not have to backtrack opposite to its job direction to pick up the passenger from the floor. For example, if a passenger presses an 'up' button on level 2, then a lift that is completing a set of upwards jobs but is on level 5 will not be considered, as it would have to backtrack downwards (the opposite direction to its job direction) to pick up the passenger.

$$lifts\ considered = idle\ lifts\ \cup (lifts\ going\ in\ same\ direction \cap lifts\ not\ needing\ to\ backtrack)$$

*Equation 1 - The composition of sets to build a set of lifts considered for job assigning*

From the set of lifts considered, the lift that is closest to the floor where the button was pressed is selected to travel to the floor and pick up the passenger, and the floor is assigned to the lift's jobs queue. Closeness is determined by the difference in the index of the lift's floor and the index of the floor the floor button was pressed on. In the case of a tie in lift closeness, an arbitrary lift from the equally close lifts is chosen. The case of an idle lift being chosen may allow for the scenario where the lift must travel in the opposite direction to the requested floor button to pick the passenger up.

In the case that there are no lifts available in the lifts considered set, the floor is added to a global waiting list observable by all lifts in the lift system. On completion of a job queue, i.e. a lift opens its door, removes the head job off the job queue and its remaining job queue is empty, the lift will check the waiting list for any jobs. If the waiting list is not empty, it will assign to its own job queue the head of the waiting list. The waiting list is a FIFO queue to prevent request starvation. The direction and level of the request for the floor are store, so when a lift picks a job off the queue it is able to determine not only the level it should travel to but also the job direction it should be in.

It is assumed all lifts move at an equal velocity. A lift can move when it has a non-empty job queue and its doors are closed. Upon reaching the floor first in the lift's job queue, the lift will stop moving and open its doors.

The objects of the system described are summarised in 2.2 and 2.3.

## 2.2 Lift

- A set of internal buttons mapping to the floor labels in the lift system
    - On press they add floor indexes to the lift's job queue depending on the lift's current job direction
- A lob list of floor indexes the lift must visit in the order they appear in the queue. The queue is ordered ascending of the lift is completing a set of upwards jobs, and descending if the lift is completing a set of downwards jobs
- A set of doors which can be either open or closed
- A current level index of which the lift is on
- A job movement direction, upwards if all jobs are above, downwards if all jobs are below, idle if no jobs.

## 2.3 Floor

- A set of buttons to request lifts.
    - Only a downwards button if there is no floor above the current floor, and only an upwards button if there is no floor below the current floor. All other floors have both

upwards and downwards buttons unless there is only one floor, in which case there are no buttons.

- A floor label separate from the floor's index which must be unique.

# 3 Z Model

The objects discussed in 2.2 and 2.3 map directly over to the Z model. These 2 objects are part of a state schema *LiftBank*, which encapsulates the entire lift system. The operation schemas for the system are *addLift* and *addFloor* for adding any number of lifts and floors, *pressLiftButton* which simulates a passenger pressing an internal button of a lift, *closeDoor* which simulates a user closing a lift door, *pressFloorButton* which simulates a user pressing an up or down button on a floor, and *moveAllLifts* which simulates all lifts moving according to their current job queues. All schemas mentioned are fairly complex and will be discussed further in depth.

## 3.1 State Schemas

### 3.1.1 Lift

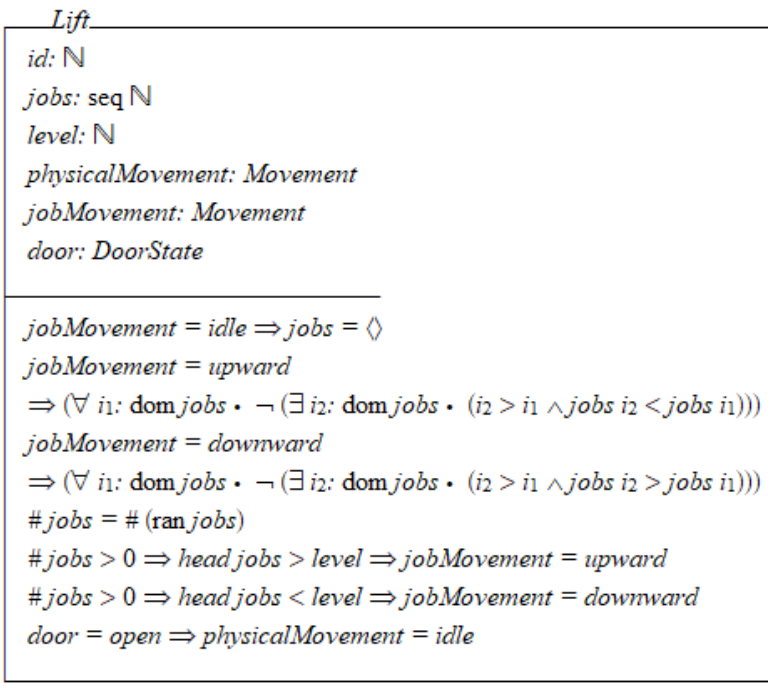


*Figure 1 - Z Lift state schema*

The lift schema contains the same attributes discussed in 2.2 in addition to some new attributes. An id is present for each ID which the purpose of distinguishing lifts, each lift ID is unique. This constraint is specified in *LiftBank*. For ease of modelling, both jobs and the lift's current level are integers, which map to indexes of floors in the *LiftBank* schema. In addition to the lift's job direction in *jobMovement*, each lift has a *physicalMovement* attribute which describes the lift's physical movement at ant time. This is necessary for reasons discussed in 2.1, where a lifts physical movement may not always be the same as its job movement direction. The door attribute can take the value of either open or closed.

In order, he first predicate invariants of the *Lift* state schema state that:

1) A lift's job queue must be empty if it's *jobMovement* is idle. (A *jobMovement* of idle means that the lift is not completing any jobs, hence it's job queue must be empty).
2) If the lift's job queue must be in ascending order if it's *jobMovement* is upward.
3) Vice-versa to 2).
4) Each job in the lift's job queue is unique.
5) If there are jobs in the lift's job queue, and if the head of the queue is above the current level then the *jobMovementi* of the lift must be upward. (This is helpful because if a lift has a job added it its job queue, then the lift's *jobMovement* attribute can be automatically, derived, and hence the order of the queue can be derived.)
6) Vice-versa to 5).

7) The lift must not be moving if the doors are open for safety reasons.

### 3.1.2 Floor

*Floor*

$label: FloorLabel$

$buttons: \mathbb{P}\ FloorButton$

*Figure 2- Z Floor state schema*

The floor schema simply contains a floor label and a set of buttons. There are no predicate invariants required because there is no relationship between the floor's label and its number of buttons. Constraints like floor label uniqueness across the lift system and the number of buttons depending on the floors index are described in the *LiftBank* schema, where floors are eventually assigned indexes and therefore a sense of order.

### 3.1.3 LiftBank

*LiftBank*

$lifts: \mathbb{P}\ Lift$

$floors: \text{seq}\ Floor$

$waitingList: \text{seq}\ (Floor \times FloorButton)$

---

$\#floors = \#(\text{ran}\ floors)$

$\forall f_1: \text{ran}\ floors \bullet \neg (\exists f_2: \text{ran}\ floors \bullet (f_2 \neq f_1 \land f_2.label = f_1.label))$

$\forall i_1: \text{dom}\ floors$

$\bullet\ (\neg (\exists i_2: \text{dom}\ floors \bullet i_2 > i_1) \land (\exists i_3: \text{dom}\ floors \bullet i_3 < i_1)$

$\Rightarrow (floors\ i_1).buttons = \{down\})$

$\land ((\exists i_2: \text{dom}\ floors \bullet i_2 > i_1) \land \neg (\exists i_3: \text{dom}\ floors \bullet i_3 < i_1)$

$\Rightarrow (floors\ i_1).buttons = \{up\})$

$\land ((\exists i_2: \text{dom}\ floors \bullet i_2 > i_1) \land (\exists i_3: \text{dom}\ floors \bullet i_3 < i_1)$

$\Rightarrow (floors\ i_1).buttons = \{up, down\})$

$\land (\neg (\exists i_2: \text{dom}\ floors \bullet i_2 > i_1) \land \neg (\exists i_3: \text{dom}\ floors \bullet i_3 < i_1)$

$\Rightarrow (floors\ i_1).buttons = \emptyset)$

$\forall l_1: lifts \bullet \neg (\exists l_2: lifts \bullet (l_2 \neq l_1 \land l_2.id = l_1.id))$

$\text{dom}\ (\text{ran}\ waitingList) \subseteq \text{ran}\ floors$

$waitingList \neq \langle\rangle \Rightarrow (\forall\ l: lifts \bullet l.jobMovement \neq idle)$

*Figure 3 - Z LiftBank state schema*

The purpose of the *LiftBank* schema is to unify the lift and floor schemas, declare constraints on the collections of these objects as well as provide a root state schema for operation schemas to act on. It contains a set of lift types, to model the set of lifts the lift system will have. No ordering of any kind is assumed, so a set is considered sufficient. In has an attribute of a sequence of floors, which models the ordered set of floors the building will have. The domain of the sequence maps to the index and therefore order or 'height' of the lift, so the first lift is the lowest at height 1, the second at height 2, etc. Finally, a sequence of *Floor* – *FloorButton* values are sored in a waiting list sequence, which acts in a FIFO manner as described in 2.1. The floor indicates the level of the new job for the lift, and the button indicates the *jobMovement* direction the left should set for itself, therefore restricting the floor buttons that user is able to press and ensuring other passengers do not unfairly intercept it.

In order, the predicate invariants of the *LiftBank* state schema state that:

1) All floors in the *floors* sequence must be unique.
2) There may be no 2 floors in the *floors* sequence with the same label.
3) All floors in the *floors* sequence adhere to the following button rules:
    a. If there is no floor above, the only button the floor has is down.
    b. If there is no floor below, the only button the floor has is up.
    c. If there is a floor above and a floor below, the buttons the floor has are up and down.
    d. If there is no floor above or below, the floor has no buttons.
4) All lifts in the lift set have a unique id.
5) Tall floors in the *waitingList* must be in the floor sequence
6) If the waitingList is populated then there cannot be a lift which is idel, else it would have taken a job off the *waitingList*.

## 3.2 Operation Schemas

For brevity, the *addLift* and *addFloor* schemas have been omitted from this section as they do not form part of the core functionality of the system, and they can be considered trivial. Furthermore, *closeDoor* has also been omitted due to its simplicity.

### 3.2.1 PressLiftButton

*PressLiftButton*

$$
\begin{aligned}
&\Delta LiftBank \\
&liftId?: \mathbb{N} \\
&floorLabel?: FloorLabel \\
&\hline \\
&\exists\, l_1 : lifts;\ level: \mathbb{N} \\
&\quad \bullet\ (floors\ level).label = floorLabel? \\
&\qquad \wedge\ liftId? = l_1.id \\
&\qquad \wedge\ (level \in \mathrm{ran}\ l_1.jobs \Rightarrow lifts' = lifts) \\
&\qquad \wedge\ (l_1.level = level \Rightarrow lifts' = lifts \setminus \{l_1\} \cup \{openLiftDoors\ l_1\}) \\
&\qquad \wedge\ (\neg\ (l_1.level = level \vee level \in \mathrm{ran}\ l_1.jobs) \\
&\qquad\quad \Rightarrow lifts' = lifts \setminus \{l_1\} \cup \{addJobToLift\ (l_1, level)\}) \\
&floors' = floors \\
&waitingList' = waitingList
\end{aligned}
$$

*Figure 4 - Z PressLiftButton Operation Schema*

The operation takes an integer id corresponding to a lift and a *FloorLabel* corresponding to some floor's label in the lift system. The operation should be interpreted as 'press the inner lift button corresponding to *floorLabel* in the lift with id *liftId*'. The operation will add the floor to the lift's job queue only if the floor is in the direction of the lift's current job movement, satisfying requirements described in 2.1. If the floor is not in this direction the operation cannot be run. If the floor pressed is already in the lift's job queue, nothing will happen. If the floor pressed if the floor which the lift is currently on, the lift's doors will open if they aren't already, and the job will not be added to the lift's job queue. The exact way in which the job is added to the queue is not specified in the *addJobToLift* axiom (which is further discussed in 3.3, as it is left to the predicate invariants describe in the *Lift* schema to determine job queue ordering.

The predicates of note in the *PressLiftButton* operation schema are as follows:

1) The *liftId* input must correspond to some lift's id in the *lifts* set, and the *floorLabel* input must correspond to some floor in the *floors* sequence's label. We know this lift is unique because all floor labels are unique.
    a. If the level is equal to the lift's current level, open the doors.
    b. If the level is already in the range of the lift's jobs, do nothing.
    c. If a. and b. are not true, add the job to the lift's job queue.

### 3.2.2 PressFloorButton

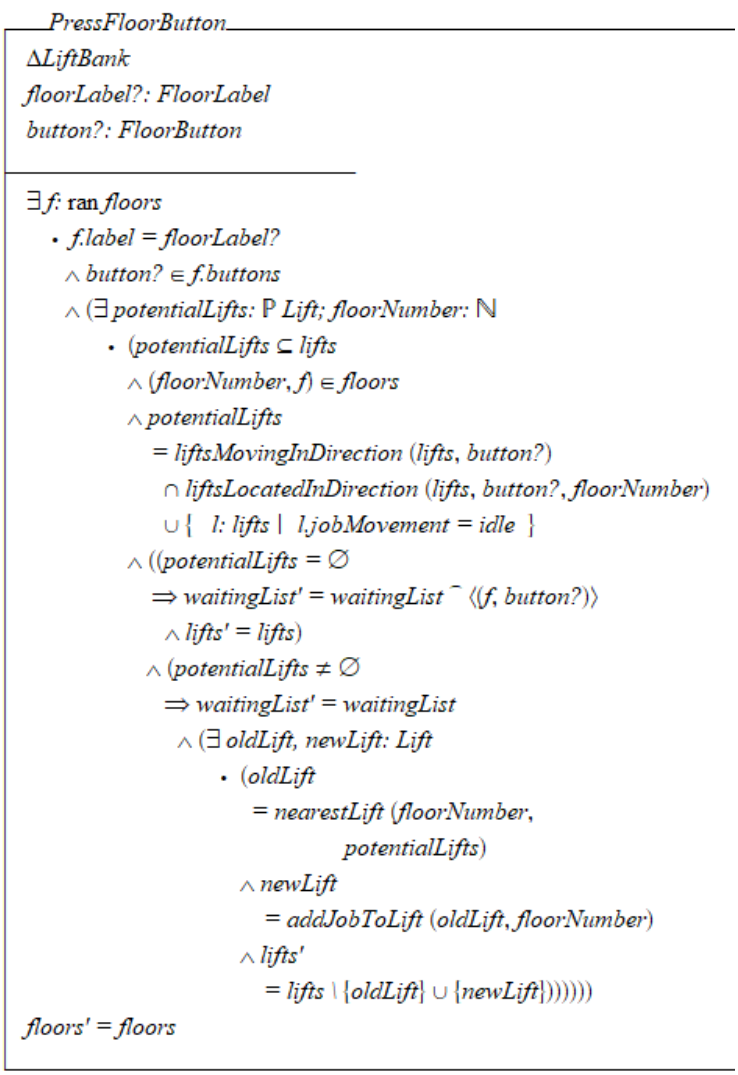


*Figure 5 - Z PressFloorButton operation schema*

The *PressFloorButoon* operation schema simulates the pressing of a button on a certain floor of the lift system. It will send a job to any available lift. The inputs to the schema are a floor label corresponding to the label of a floor in the lift system, and a button that exists on that floor according to the rules defined in 2.1. The operation considers the floor and the direction of the button press, and selects a set of lifts from a set of 1) lifts that are both going in the same direction as the request and can get to the floor in question in that direction from their location, and 2) and idle lifts. The closest lift of this set is sent the job to append to its job queue. If no lifts that meet these requirements are met, then the floor and the direction of the button press will be appended to the back of the global *waitingList*.

The predicates of note in the *PressFloorButton* operation schema are as follows:

1) The floor and button inputs must each correspond to a floor and a button on that floor respectively. A set of potential lifts for sending the request to must be a subset of *lifts*. This set of potential lifts is the intersection of lifts moving in the correct direction and the lifts which have the floor in the direction of which they are moving, all with the union of with the set of idle lifts. If this set is empty the *waitingList* has the floor and direction appended to the end. If the set of potential lifts is not empty a new lift is defined with all same properties as the nearest lift out of the set of potential lifts, and this new defined lift replaces its old counterpart in the set of *lifts*.

### 3.2.3 MoveAllLifts

*MoveAllLifts*

$\Delta LiftBank$

---

$\#\, lifts' = \#\, lifts$

$\forall\, l_1 : lifts$

$\bullet\ \exists\, l_2 : Lift;\ nonZeroLiftDesiredMovements : \mathbb{P}\, \mathbb{N};\ levelsToMove : \mathbb{N}$

$\bullet\ nonZeroLiftDesiredMovements = getLiftJobDifferences\ lifts \setminus \{0\}$

$\wedge\ (nonZeroLiftDesiredMovements = \varnothing$

$\Rightarrow lifts' = lifts \wedge waitingList' = waitingList)$

$\wedge\ (nonZeroLiftDesiredMovements \neq \varnothing$

$\Rightarrow levelsToMove = min\ nonZeroLiftDesiredMovements$

$\wedge\ l_2 = moveLift\ (l_1, levelsToMove)$

$\wedge\ (\exists\, level : \mathbb{N};\ floor : Floor;\ direction : FloorButton$

$\bullet\ ((l_2.jobs = \langle\rangle \wedge waitingList \neq \langle\rangle$

$\Rightarrow floor \in \mathrm{dom}\ \{(head\ waitingList)\}$

$\wedge\ direction \in \mathrm{ran}\ \{(head\ waitingList)\}$

$\wedge\ (level, floor) \in floors$

$\wedge\ l_2.jobs = \langle level \rangle$

$\wedge\ (direction = up \Rightarrow l_2.jobMovement = upward)$

$\wedge\ (direction = down$

$\Rightarrow l_2.jobMovement = downward)$

$\wedge\ waitingList' = tail\ waitingList)$

$\wedge\ \neg\, (l_2.jobs = \langle\rangle \wedge waitingList \neq \langle\rangle)$

$\Rightarrow waitingList' = waitingList))$

$\wedge\ l_2 \in lifts')$

$floors' = floors$

$waitingList' = waitingList$

*Figure 6 - Z MoveAllLifts operation schema*

Perhaps the most complex of the operation schemas, *MoveAllLifts* can be thought of as a large scale 'tick' of varying length, where lifts are able to move and therefore change level. The lifts which move are those which have a non-empty job queue and have closed doors. Any lift that finishes the tick on the level that corresponds to the first level on its job queue will stop motion, have its doors open and remove the level from its job queue, simulating a lift reaching its floor. The 'length' of the 'tick' corresponds to the amount of levels the lifts will travel. For example, if the length of the tick is three, then all lifts that can move will move up to 3 levels. The length of the tick depends on the **minimum non-zero distance any movable lift has to the floor on the head of its job queue**. For example, imagine a scenario with three lifts. Lift 1 has its doors open, is on level 3 and a next job of level 4, lift 2 has its doors closed, is on level 3 and has its next job of level 5, and lift 3 has its doors closed, is on level 3 and has its next job of level 6. The *minimal non-zero distance the non-movable lifts have to their destination* is 2 from lift 2, as lift 3 has 3 levels to travel from level 3 to level 5 and lift 1 cannot move as its doors are open. When the *moveAllLifts* operation is therefore applied, lift 1 will still be on level 3, lift 2 will be on level 5 with its doors open and level 5 removed from its job queue, and lift 3 will be on level 5 with its doors closed, and still have level 6 on its job queue. It is worth noting that this mechanic is merely a simulation of a model of a real-life lift system. Obviously lifts in real life do not move in discrete ticks, but for the purposes of this model the mechanic is sufficient.

The predicates of note in the *MoveAllLifts* operation schema are as follows:

1) The number of lifts in *lifts* remains the same after the operation (this is important because we construct a new set of lifts from the old set, and without this predicate the set become unbounded.)
2) For every lift in *lifts* before the operation, the *minimum non-zero distance any movable lift has to the floor on the head of its job queue* is calculated, and if it is the empty set then the lift set remains unchanged, as this tells us that there are no candidate lifts to move. If it is not the empty set, then move the lift in question the same number of levels as the minimum value of the set

of movable lift distances. If this lift is now at its destination level, open its doors and remove the head job from its queue. If the lift's queue is empty and the global *waitingList* is nonempty at this point, remove its head and assign the floor to the lift's job list, and set the job movement of the lift accordingly. Note the lines $floor \in dom\{(head\ waitingList)\}$ and $direction \in ran\{(head\ waitingList)\}$ will only ever evaluate to a single value each because the set will only ever have one element, the head of the *waitingList*. Finally, this deduced new lift is said to be in the *lifts'* set after the operation, and because this operation is done to every single lift in the pre operation set as well as the cardinalities of the pre-set and post-set being defined as the same, we know that the pre and post lift set contain the same lifts with only potentially altered doors, job queues and levels.

## 3.3 Axioms of note

### 3.3.1 AddJobToLift

$$addJobToLift: Lift \times \mathbb{N} \rightarrow Lift$$

---

$$\begin{aligned}
&\forall\ l_1: Lift;\ l_2: Lift;\ level: \mathbb{N} \\
&\quad \bullet\ addJobToLift\ (l_1, level) = l_2 \\
&\quad \Leftrightarrow l_1.id = l_2.id \\
&\qquad \wedge\ l_1.level = l_2.level \\
&\qquad \wedge\ level \notin \text{ran}\ l_1.jobs \\
&\qquad \wedge\ l_1.physicalMovement = l_2.physicalMovement \\
&\qquad \wedge\ l_1.door = l_2.door \\
&\qquad \wedge\ level \in \text{ran}\ l_2.jobs \\
&\qquad \wedge\ \#\ (l_2.jobs \rhd \{level\}) = 1 \\
&\qquad \wedge\ \text{ran}\ l_1.jobs = \text{ran}\ (l_2.jobs \rhd \{level\})
\end{aligned}$$

*Figure 7 - Z addJobToLift axiom*

This axiom takes a tuple of a lift and an integer signifying a floor index (level) to add to the lift's job queue, that isn't already in the lift's job queue. The axiom is worth discussing because it illustrates the declarative nature of the lift job queue ordering. The axiom works by deducing another variable of type lift with all same attributes as the input lift except for the job queue, which is defined as follows. The level input to the axiom **must** be in the sequence of the new deduced lift, the number of times this level appears in the job queue must be only once, and the levels stored in the queues of the old and newly deduced lifts, apart from the newly introduced level are the same. These constraints mean that the output lift's job queue must contain the level input to the axiom, but that will be the only new element with no other elements removed. Because of this loose definition, the lift schema itself will determine the exact ordering of the job queue as discussed in 3.1.1, and there is no knowledge/procedure describing queue insertion here.

# 4 Theorems

Several theorems have been included in the model for verification purposes. Some of the more important theorems will be discussed here. It is important to note not all theorems written as part of the model are mentioned, as they are either too simple or not important enough.

## 4.1 Lift must be idle for doors to open

Perhaps the most important property of the model to verify, it is crucial that the doors of the lift remain closed when the lift is moving. It is an extreme health and safety hazard for a user if this property is not met. The theorem states that is a lift is moving, then its doors must be closed.

$$\begin{aligned}
&\textbf{theorem}\ LiftsMustBeIdleToOpenDoors \\
&\forall\ LiftBank \\
&\quad \bullet\ MoveAllLifts \\
&\quad \Rightarrow (\forall\ lift: lifts' \bullet\ lift.physicalMovement \neq idle \Rightarrow lift.door = closed)
\end{aligned}$$

*Figure 8 - Theorem stating doors must be closed if lift is moving*

## 4.2 Lift must only accept valid jobs

For the purposes of user experience and efficiency of the system, a lift should only add a floor to its job queue if the job meets a few rules. These rules state that the floor requested must be reachable by the lift in its same direction of job movement, relative from where the lift is. In other words, the lift should not backtrack to a job. It is important to test the *PressFloorButton* and *PressLiftButton* specifications in this case, because they deal with assigning lifts jobs. The theorem states that a lift cannot have a job in its queue less than its level if it is moving upward, or a job higher than its level if it is moving downward. The theorem is checked upon *PressLiftButton* and *PressFloorButton* as they deal with adding jobs to lifts.

**theorem** *LiftMustNotHaveJobsInOppositeDirectionToJobMovement*
$\forall$ *LiftBank*; $fl_1$: *FloorLabel*; $fl_2$: *FloorLabel*; *fb*: *FloorButton*; *fid*: $\mathbb{N}$
• *PressLiftButton*[*liftId*? := *lid*, *floorLabel*? := $fl_1$]
$\wedge$ *PressFloorButton*[*floorLabel*? := $fl_2$, *button*? := *fb*]
$\Rightarrow$ ($\forall$ *lift*: *lifts*'
• (*lift.jobMovement* = *upward*
$\Rightarrow \neg$ ($\exists$ *n*: ran *lift.jobs* • *n* < *lift.level*))
$\wedge$ *lift.jobMovement* = *downward*
$\Rightarrow \neg$ ($\exists$ *n*: ran *lift.jobs* • *n* > *lift.level*))

*Figure 9 - Theorem stating only valid jobs may be added to a lift*

## 4.3 Lift job queues must be ordered accordingly

Depending on the direction of jobs for a lift the ordering of the queue is very important. Jobs are taken off the head of the queue, and the closest floor should be chosen to visit first, to ensure a positive passenger experience, and to ensure the lifts run efficiently. If the lift has a job movement of downwards, the job queue should be ordered descending, and vice-versa. This theorem states these two constraints for each scenario.

**theorem** *LiftsJobQueuesAreOrderedCorrectly*
$\forall$ *LiftBank*; $fl_1$: *FloorLabel*; $fl_2$: *FloorLabel*; *fb*: *FloorButton*; *lid*: $\mathbb{N}$
• *PressLiftButton*[*liftId*? := *lid*, *floorLabel*? := $fl_1$]
$\vee$ *PressFloorButton*[*floorLabel*? := $fl_2$, *button*? := *fb*]
$\vee$ *MoveAllLifts*
$\Rightarrow$ ($\forall$ *lift*: *lifts*'
• (*lift.jobMovement* = *upward*
$\Rightarrow$ ($\forall$ $i_1$: dom *lift.jobs*
• $\neg$ ($\exists$ $i_2$: dom *lift.jobs*
• ($i_2 > i_1 \wedge$ *lift.jobs* $i_2$ < *lift.jobs* $i_1$))))
$\wedge$ (*liftMovement* = *downward*
$\Rightarrow$ ($\forall$ $i_1$: dom *lift.jobs*
• $\neg$ ($\exists$ $i_2$: dom *lift.jobs*
• ($i_2 > i_1$
$\wedge$ *lift.jobs* $i_2$ > *lift.jobs* $i_1$)))))

*Figure 10 - Theorem stating a lift's job queue must be ordered according to its job direction*

## 4.4 A lift will move if it is able to

For the lift system it is crucial that the lifts move when they are expected to. If a lift has its doors closed and a non-empty queue, it should be moving from level to level, getting closer to its target destination. This ensures a positive user experience. The theorem states that if a lift has a non-empty job queue and its doors are closed (it is expected to move) then there exists a second lift in the post state schema with the same id, and its level has changed in the direction expected.

$$
\begin{aligned}
&\textbf{theorem } LiftsWithNonEmptyQueuesMove \\
&\forall\ LiftBank \\
&\quad \bullet\ MoveAllLifts \\
&\quad \Rightarrow (\forall\ l_1 : lifts \\
&\qquad \bullet\ l_1.jobs \neq \langle\rangle \land l_1.door = closed \\
&\qquad \Rightarrow (\exists\ l_2 : lifts' \\
&\qquad\quad \bullet\ l_1.id = l_2.id \\
&\qquad\quad \land (l_1.jobMovement = upward \Rightarrow l_2.level > l_1.level) \\
&\qquad\quad \land (l_1.jobMovement = downward \\
&\qquad\quad\ \Rightarrow l_2.level < l_1.level)))
\end{aligned}
$$

*Figure 11 - Theorem stating when a lift is expected to move it will move in the expected direction*

## 4.5 Only lift doors will open if a lift job is the same floor it's on

This theorem covers 2 scenarios, a lift is on level 3 with its doors closed, and someone on level three requests a job valid for the lift. In this case the lifts doors should open, and the job should not be added to its job queue, as it is already on the level. Conversely, if a button press for level 3 comes from inside the lift, the same behaviour should apply. This ensures unnecessary queue operations are prevented, and lift doors open when expected. Furthermore, it allows for a floor button to act as an immediate door opener if the lift is idle. The theorem acts by checking that all lifts in a pre-state schema valid for this behaviour have a corresponding lift in the post state schema, whose doors are open and job queue remains the same.

$$
\begin{aligned}
&\textbf{theorem } OnlyLiftDoorsOpenWhenSameLiftLevelButtonPressed \\
&\forall\ LiftBank; lid : \mathbb{N}; fl : FloorLabel \\
&\quad \bullet\ PressLiftButton[liftId? := fid, floorLabel? := fl] \\
&\quad \Rightarrow (\forall\ l_1 : lifts \\
&\qquad \bullet\ (\exists\ floor : \mathbf{ran}\ floors; index : \mathbb{N} \\
&\qquad\quad \bullet\ floor.label = fl \\
&\qquad\quad \land (index, floor) \in floors \\
&\qquad\quad \land index = l_1.level \\
&\qquad\quad \Rightarrow (\exists\ l_2 : lifts' \\
&\qquad\qquad \bullet\ l_1.id = l_2.id \\
&\qquad\qquad \land l_2.door = open \\
&\qquad\qquad \land l_2.jobs = l_1.jobs)))
\end{aligned}
$$

*Figure 12 - Theorem stating a lifts doors should just open when a job equal to the lifts level is submitted*

# 5 Lessons Learned

Translating a set of requirements to a declarative formal specification has proven to be a challenging yet rewarding task. With learning a new language there is bound to be teething problems, and this assignment has been no exception. The predominant issue I faced in the writing of the model was a lack of understanding in structuring the attributes of a schema, and the consequences this would have on operations to those schemas.

An example of this naivety showing in the model is the structuring of the lifts in the *liftBank* schema. The lifts are stored as a set, but on reflection I believe they would have been better stored as a function of lift id to lift, as this would have allowed simpler syntax in the operation schemas. Namely, whenever an operation schema has input a lift id, and the lift itself must be read or altered, then the lift must be existentially quantified and said to have the same id as the lift id input before use. This is far more verbose than the simple function syntax that would have been required otherwise.

An important lesson I learned late into the project is the importance breaking schemas down to compositions of axioms. For example, the *moveLift* axiom is rather large and complex by itself, yet this predicate was once included in the *MoveAllLifts* operation schema, making it nearly twice as long and far harder to read. In fact, I believe readability is a fundamentally important attribute of documentation, of which this model is a type of. Another good example of the importance of 'axiom encapsulation' is the *PressFloorButton* axiom, which needs to be able to deduce the lift sets according to those defined in Equation 1. Without the axioms deducing the lifts moving in a certain direction, the lifts located correctly relative to their movement or the nearest lift out of these, the axiom was far longer and far harder to read. Finally, the use of axioms allows for a degree of reuse in the model, for example the *abs* (absolute value) function and the *addJobToLift* function.

# 6 Conclusion

The lift system has been modelled in the Z formal modelling language. The model has been structured with a root state schema called a *LiftBank*, which is composed of a set of lifts, a sequence of floors and a sequence of jobs on a waiting list. The model can simulate the act of adding floors and lifts, pressing a directional button on a floor to request a lift, pressing a floor button inside a lift to request a floor, closing a lift door, and a discrete movement operation where all lifts in the system move the minimum amount for a certain lift to reach its destination. While functions like manually closing lift doors and a coarse grained 'tick' like operation are assumptions about the lift systems functionality, they are adequate for out purposes here. Furthermore, all these functions have been developed from a set of requirements for the system. Therefore, in a sense, this has been a translation exercise. Finally, a set of theorems have been included in the model for verification of the system's most important properties, like doors closing on lift movement and the ordering in which a lift will visit floors. I have gained a lot of learning from doing this assignment, namely that of understanding how to effectively model using Z in a readable and concise fashion.