The Ups and Downs of Formal Methods

Michael R. Clarkson
Ann E. Kelley Sobel
Toni E. Lehmkuhl
Stephanie R. Taylor
Bryce D. Williams

Department of Systems Analysis
Miami University
Oxford, OH 45056

June 21, 1999

Abstract

A solution to the elevator scheduling problem is developed in this article that utilizes a formal method involving only first-order logic. From the English requirements, a mathematical specification of the problem is developed, and code is derived directly from the specification. The solution is compared with other (formal and non-formal) solutions.

Keywords: Formal specification, Program derivation, Verification, Software engineering

The Ups and Downs of Formal Methods

1 Introduction

Technology has the potential to simplify life, but the details behind it are rarely simple. The average person often takes those details for granted. For instance, a conversation between office workers waiting for an elevator would rarely involve the algorithm used to schedule the arrival of that elevator, and if such a conversation were to take place, it would no doubt be met by quizzical looks from those nearby. Yet that algorithm is of great importance to all involved, as an elevator that traveled randomly between floors and caused long delays would be found unacceptable.

The goal of this article is to completely demonstrate the use of formal methods of software engineering in solving a problem that is relevant to everyday life, reasonably intuitive, and yet challenging. The elevator scheduling problem will be used to introduce a formal method involving only first-order logic. The requirements to satisfy are those of a typical elevator. From the requirements, a mathematical specification of the problem will be developed, and code will be derived directly from this specification. This approach allows the definition of the elevator scheduling problem in a manner that will demonstrate the use of formal methods in software specification and design, as well as lead to a concise and precise solution to the problem.

2 Problem

The informal definition of the elevator scheduling problem was taken from the call for papers of the 1987 IEEE International Workshop on Software Specification and Design, where it was explicitly proposed as a "challenging exercise" [IEE87]. This problem involves an elevator system that schedules the processing of requests. The informal requirements were adapted as follows:

1. Maintain a set of buttons that have been pushed inside the elevator (requested floor number)

2. Maintain a set of buttons that have been pushed outside the elevator (floor number in which passenger is waiting and requested direction)

3. Halt when elevator has no more requests to service

4. Service all outside requests eventually with all floors given equal priority

5. Service all inside requests eventually with all floors being serviced sequentially in the direction of travel.

A simplistic solution to this problem would be to utilize a first-in, first-out queue, that is, enqueue all requests, and service each request sequentially in the order received. This solution was disallowed, however, due to its inconsistency with real-life elevator systems. Therefore, a qualitative goal of efficiency was introduced in the design of the scheduling algorithm. It was decided the algorithm should ensure that requests are processed in a timely manner, and that the direction of elevator movement and stops occur in a logical manner that maximizes the use of the elevator.

3 Design

The initial design of the system was an object oriented design (OOD) using unified modeling language (UML). The primary classes discovered were *Elevator* and *ElevatorSystem*. The majority of the methods in these classes needed little in the way of algorithmic design, for example, list operations. The scheduling algorithm itself was encapsulated in the *UpdateElevator* method of class *Elevator*. Formal methods were used to design *UpdateElevator*. It was specified using first-order logic, and derived into guarded command code using the weakest precondition predicate transformer (wp) [Dij76]. In the OOD there exists, for each object of class *Elevator*, a set containing the numbers of the buttons that are illuminated inside the elevator. There also exists a set containing the number of the floor in which a request is waiting and the direction, up or down, of the request. This fulfills the existence, but not the maintenance, stipulation of the first two requirements. Since maintenance is a function of the scheduling algorithm, it was included in the formal specification. The third requirement, since it related to movement of the elevator, was also included in the formal specification.

The fourth requirement posed a difficulty due to the conflict between the requirements of equal priority and efficiency. A scheduler that gives an equal, non-deterministic priority to all floors could interrupt its current requests to service a floor far away. This, again, would be inconsistent with real-life elevator systems. The equal priority requirement was therefore interpreted as follows: If a halted elevator received two new service requests at the same time, no bias would exist towards choosing which new request to fulfill.

The sequential component of the fifth requirement was fulfilled in the OOD. A simulation framework that encapsulated the elevator system was included in the design. This framework changes the current floor of an elevator by at most one floor in the direction chosen by *UpdateElevator*, and calls *UpdateElevator* for every floor visited by the elevator, as well as at certain time intervals for halted elevators.

Finally, the liveness properties of the fourth and fifth requirements (eventual service of requests) are considered in a latter section.

4 Specification

4.1 Outline

Design of the scheduling algorithm, *UpdateElevator*, was accomplished through formal specification. The specification language used was first-order logic. The algorithm had to meet certain components of the requirements by loading and unloading passengers, and determining direction of movement. The specification can be found below.

The parameters required by *UpdateElevator* are:

- $CurFloor$, an integer representing the floor where the elevator is currently located
- $CurDir$, a value in the set $\{UP, DOWN, HALT\}$ that represents the direction in which the elevator is currently moving
- $InsideReqSet$, a set of integers representing the floor numbers associated with the buttons currently illuminated inside the elevator
- *OutsideReqSet*, a set of *OutsideReq* values (pairs consisting of a floor number and direction value) that represent the set of buttons currently illuminated outside the elevator. Elements are added to this set by a process external to the elevator scheduler.

The precondition is necessary for saving the initial values of *CurDir, InsideReqSet*, and *OutsideReqSet* since these values can change during the execution of *UpdateElevator*, and both the initial and final values must be referenced in the postcondition. Thus, references to CD, I, or O in the postcondition are actually references to the initial values of $CurDir, InsideReqSet$, and *OutsideReqSet*.

The postcondition was separated into one main portion, two predicates and one function. These predicates, whose definitions follow, are textual substitutions for frequently repeated pieces of the postcondition and are used to improve readability.

The only function, $Rev(Dir)$, returns the reverse of the direction Dir. For example, $Rev(UP)$ would return the direction value $DOWN$.

$ReasonToGo(Dir)$ is a predicate that is *true* if there is a reason to go in the direction Dir. For example, $ReasonToGo(UP)$ is *true* either if someone inside the elevator has pushed a button requesting to go to a floor higher than the floor the elevator is currently on, or if someone on a higher floor outside the elevator has pushed a button to request the elevator.

PickupDropoff(Dir) is a predicate that specifies the changes that occur in the inside and outside request sets as passengers are picked up and dropped off at different floors throughout the building. When conditions permit the fulfillment of a request, this predicate specifies that the request will no longer exist in the appropriate request list (though it may be added to the other list) after execution of the algorithm. This fulfills the maintenance component of the first two English requirements.

In the main postcondition, *ReasonToGo* is used to realize the liveness properties of the fourth and fifth requirements. As long as requests exist this predicate will be **true**, and thus the elevator will not stop moving and servicing requests, although no formal proof of this has yet been devised. When *ReasonToGo* becomes **false** for all directions, the main postcondition chooses *HALT* as the new elevator direction, thus fulfilling the third requirement. Other directions are chosen based upon *ReasonToGo*. The guiding principle behind the specification is that of *least work*; in particular, the elevator does not change direction until there is no reason to continue in the current direction. This is in order to meet the efficiency goals of the requirements. By continually travelling in one direction for a maximal distance and only taking on passengers travelling in that direction, inside requests can be serviced in a timely manner.

The original draft of the specification required approximately five hours to write. Revisions of the specification were required as errors were discovered through a process of test case analysis, as well as during the proving of the associated guarded command code. The errors were generally due to the specification being incomplete. For example, one error resulted from not specifying how the elevator should resume motion after halting. This error was discovered when considering a test case. Another error was discovered when the proof led to the generation of an **abort** statement. Revising the specification to account for errors required approximately ten additional hours, some of which was used for generating key portions of the proof. All errors in the specification were discovered before implementing the code - that is, the specification never required revision due to errors discovered during actual code testing.

4.2 Notation

The specification format of Cohen [Coh90] was founded on the Hoare triple.

$$\{Q\} \ \ \mathrm{S} \ \ \{R\}$$

> The execution of S begun with the precondition Q satisfied is guaranteed to terminate with R satisfied.

Cohen uses both the pre- and postcondition as defined in the Hoare triple when writing a specification. Any variables that are used either in the pre- or postcondition must be declared.

For example, given the following program:

> Set q and r to the quotient and remainder of the integer division of $x \ \geq 0$ by $y \ > 0$.

the specification is written as follows:

$$\mathbf{var}\ x,y,q,r:\ int\ \{x \geq 0 \wedge y > 0\}\ ;\ R$$

The specification consists of two parts, the precondition and the postcondition, separated by a semicolon. The precondition consists of variable declarations and predicates, surrounded by braces, over the variables. Multiple declarations are separated by semicolons, and each declaration may be followed by its own predicates. The postcondition contains a list of the declared variables that are modified in the postcondition, followed by a predicate that is not in braces. The modified variables and predicate are separated by a colon.

Returning to the above example, the complete specification follows:

$$\textbf{var}\ x,y,q,r :\ int\ \{x \geq 0 \wedge y > 0\}\ ;\ q,r :\ q*y+r = x \wedge 0 \leq r \wedge r < y$$

Notice that the values of x and y are left unchanged by the specification and therefore are not mentioned as retaining their initial values in the postcondition.

It is desirable to separate the declarations along with their corresponding clauses of the precondition on separate lines in order to enhance readability.

$\textbf{var}\ x :\ int\ \{x \geq 0\}$
$;\textbf{var}\ y :\ int\ \{y > 0\}$
$;\textbf{var}\ q,r :\ int$
$;q,r :\ q*y+r = x\ \wedge\ 0 \geq r\ \wedge\ r < y$

4.3 *UpdateElevator*

Labels identify individual clauses in the specification, and in general, apply until the next label. Exceptions to this rule are bounded by brackets. The labels were introduced to enhance readability of the derivation of guarded command code.

$\textbf{var}\ CurFloor : int$
$;\textbf{var}\ CurDir : direction\ \{Q_0 : CD = CurDir\}$
$;\textbf{var}\ InsideReqSet : set\,of\,int\ \{Q_1 : I = InsideReqSet\}$
$;\textbf{var}\ OutsideReqSet : set\,of\,OutsideReq\ \{Q_2 : O = OutsideReqSet\}$
$;CurDir, InsideReqSet, OutsideReqSet : \big(B_0 : CD = HALT$
$\quad \wedge R_0 : \big[B_{00} : \big(ReasonToGo(UP)$
$\qquad \wedge R_{00} : CurDir = UP \wedge PickupDropoff(UP)\big)$
$\qquad \vee \big(B_{01} : ReasonToGo(DOWN)$
$\qquad \wedge R_{01} : CurDir = DOWN \wedge PickupDropoff(DOWN)\big)$
$\qquad \vee\big(B_{02} : \neg\big(ReasonToGo(UP) \vee ReasonToGo(DOWN)\big)$
$\qquad \wedge R_{02} : CurDir = HALT\big)\big]\big)$
$\quad \vee\big(B_1 : CD \neq HALT \wedge R_1 : \big[(B_{10} : ReasonToGo(CD)$
$\qquad \wedge R_{10} : CurDir = CD \wedge PickupDropoff(CD)\big)$
$\qquad \vee\big(B_{11} : \neg ReasonToGo(CD) \wedge ReasonToGo(Rev(CD))$
$\qquad \wedge R_{11} : CurDir = Rev(CD) \wedge PickupDropoff(Rev(CD))\big)$
$\qquad \vee\big(B_{12} : \neg\big(ReasonToGo(CD) \vee ReasonToGo(Rev(CD))\big)$
$\qquad \wedge R_{12} : CurDir = HALT \wedge PickupDropoff(HALT))\big]\big)$

where

$Rev(Dir) \equiv \{UP \quad if\, Dir = DOWN,$
$\quad DOWN \quad if\, Dir = UP,$
$\quad HALT \quad if\, Dir = HALT\}$
$ReasonToGo(Dir) \equiv \big(\exists i | 0 \le i < \#I : \big(I.i > CurFloor \wedge Dir = UP\big)$
$\quad \vee\big(I.i < CurFloor \wedge Dir = DOWN\big)\big)\big)$
$\vee\big(\exists o | 0 \le o < \#O : \big(O.o.floor > CurFloor \wedge Dir = UP\big)\big)$
$\quad \vee\big(O.o.floor < CurFloor \wedge Dir = DOWN\big)$
$\quad \vee\big(O.o.floor = CurFloor \wedge Dir = O.o.Dir\big)\big)$
$PickupDropoff(Dir) \equiv \big(\ C_0 : (CurFloor, Dir) \notin O$
$\quad \wedge U_0 : \big(C_{00} : CurFloor \notin I \vee \big(C_{01} : CurFloor \in I$
$\quad \wedge U_{01} : InsideReqSet = I - CurFloor\big)\big)\big)$
$\vee\big(C_1 : (CurFloor, Dir) \in O \wedge U_1 : \big[\big(C_{10} : CurFloor \notin I$
$\quad \wedge U_{10} : InsideReqSet = I + GetDests()$
$\quad \wedge OutsideReqSet = O - (CurFloor, Dir)\big)$
$\quad \vee\big(C_{11} : CurFloor \in I \wedge U_{11} : InsideReqSet = I + GetDests()$
$\quad -CurFloor, OutsideReqSet = O - (CurFloor, Dir))\big]\big)$
$Q = Q_0 \ \wedge \ Q_1 \ \wedge \ Q_2$
$R = (B_0 \wedge R_0) \vee (B_1 \wedge R_1)$
$U = (C_0 \wedge U_0) \vee (C_1 \wedge U_1) = PickupDropoff(Dir)$
$Direction = \{UP, DOWN, HALT\}$
$OutsideReq = (floor : int,\ dir : Direction)$
$GetDests()$ is a call to the user interface. This function returns the set of buttons just pushed inside the elevator as passengers enter. The equivalent gathering of buttons pushed outside the elevator occurs external to this scheduling algorithm.

5 Derivation

As in the specification, labels were introduced to enhance readibility. Labels apply to either the predicate that immediately follows, or the largest complete program statement that immediately follows. The derivations of $UpdateElevator()$, $ProgPD$, and a lemma are given below.

5.1 $UpdateElevator$

5.1.1 Guarded Command Code

$\{Q\}$
S :**if** $B_0 : CD = HALT \ \rightarrow$
$\quad S_0$: **if** $B_{00} : ReasonToGo(UP) \rightarrow$
$\quad\quad S_{00} : CurDir := UP; ProgPD(UP)$
$\quad\quad [] \, B_{01} :\ ReasonToGo(DOWN) \rightarrow$
$\quad\quad S_{01} : CurDir := DOWN; ProgPD(DOWN)$
$\quad\quad [] \, B_{02} :\ \neg\big(ReasonToGo(UP) \vee ReasonToGo(DOWN)\big) \rightarrow$
$\quad\quad S_{02}$: **skip**

$$
\begin{array}{l}
\qquad\qquad \textbf{fi} \\
\quad [\,]\, B_1 : CD \neq HALT \; \rightarrow \\
\qquad\qquad S_1 : \textbf{if}\, B_{10} : ReasonToGo(CD) \rightarrow \\
\qquad\qquad\quad S_{10} : ProgPD(CD) \\
\qquad\qquad [\,]\, B_{11} : \neg ReasonToGo(CD) \wedge ReasonToGo(Rev(CD)) \rightarrow \\
\qquad\qquad\quad S_{11} : CurDir := Rev(CD); ProgPD(Rev(CD)) \\
\qquad\qquad [\,]\, B_{12} : \neg\big(ReasonToGo(CD) \vee ReasonToGo(Rev(CD))\big) \rightarrow \\
\qquad\qquad\quad S_{12} : CurDir := HALT; ProgPD(HALT) \\
\qquad\qquad \textbf{fi} \\
\quad \textbf{fi} \\
\{R\}
\end{array}
$$

5.1.2 Proof

The proof proceeds by assembling progressively larger pieces of the program. The smallest portions of code, S_{ij} are proven first, followed by S_i, and finally S itself.

(1) $\{B_{00}\}\, S_{00}\, \{R_0\}$

$$
\begin{array}{l}
\quad wp.S_{00}.R_0 \\
\equiv\!<\text{Assume } B_{00} \equiv ReasonToGo(UP) > \\
\quad wp.S_{00}.\,(R_{00} \vee (B_{01} \wedge R_{01})) \\
\equiv\!< p \Rightarrow p \vee q > \\
\quad wp.S_{00}.R_{00} \\
\equiv\!<\text{Substituting for } R_{00} > \\
\quad wp.S_{00}.CurDir = UP \wedge PickupDropoff(UP) \\
\equiv\!< S_{00} = CurDir := UP; ProgPD(UP) > \\
\quad wp.CurDir := UP; ProgPD(UP).CurDir = UP \wedge \\
\qquad\quad PickupDropoff(UP) \\
\equiv\!<\text{Definition of composition}> \\
\quad wp.CurDir := UP.(wp.ProgPD(UP).CurDir = UP \wedge \\
\qquad\quad PickupDropoff(UP)) \\
\equiv\!<\text{Conjunctivity of } wp > \\
\quad wp.CurDir := UP.(wp.ProgPD(UP).CurDir = UP \wedge \\
\qquad wp.ProgPD(UP).PickupDropoff(UP)) \\
\equiv\!<\text{Proof of } ProgPD, \text{identity of } \wedge > \\
\quad wp.CurDir := UP.(wp.ProgPD(UP).CurDir = UP) \\
\equiv\!<\text{Lemma 1 (see section 5.3)}> \\
\quad wp.CurDir := UP.CurDir = UP \\
\equiv\!<\text{Definition of} :=,\; x = x > \\
\quad \textbf{true}
\end{array}
$$

(2) $\{B_{01}\}\, S_{01}\, \{R_0\}$

Derivation is exactly the same as derivation of $\{B_{00}\}\, S_{00}\, \{R_0\}$ with all occurrences of UP replaced by $DOWN$.

(3) $\{B_{02}\}\, S_{02}\, \{R_0\}$
$\quad wp.\mathbf{skip}.R_0$
$\equiv$< Definition of **skip** >
$\quad R_0$
$\equiv$< Assume $B_{02} \equiv \neg\,(ReasonToGo(UP) \vee ReasonToGo(DOWN))$ >
$\quad CurDir = HALT$
$\equiv$< $B_0 \equiv CD = HALT$ from outer **if**, $CD = CurDir$ from Q >
$\quad$ **true**

(4) $\{B_{10}\}\, S_{10}\, \{R_1\}$
$\quad wp.S_{10}.R_1$
$\equiv$< Assume $B_{10} \equiv ReasonToGo(CD)$ >
$\quad wp.S_{10}.CurDir = CD \wedge PickupDropoff(CD)$
$\equiv$< $S_{10} = ProgPD(CD)$ >
$\quad wp.ProgPD(CD).CurDir = CD \wedge PickupDropoff(CD)$
$\equiv$< Conjunctivity of wp >
$\quad wp.ProgPD(CD).CurDir = CD \wedge$
$\qquad\quad wp.ProgPD(CD).PickupDropoff(CD))$
$\equiv$< Proof of $ProgPD$, Identity $\wedge$ >
$\quad wp.ProgPD(CD).CurDir = CD$
$\equiv$< Lemma 1 (see section 5.3) >
$\quad CurDir = CD$
$\equiv$< $B_1 \equiv CurDir = CD$ from outer **if** >
$\quad$ **true**

(5) $\{B_{11}\}\, S_{11}\, \{R_1\}$
$\quad wp.S_{11}.R_1$
$\equiv$< Assume $B_{11} \equiv \neg ReasonToGo(CD) \wedge ReasonToGo(Rev(CD))$ >
$\quad wp.S_{11}.CurDir = Rev(CD) \wedge PickupDropoff(Rev(CD))$
$\equiv$< $S_{11} = CurDir := Rev(CD); ProgPD(Rev(CD))$ >
$\quad wp.CurDir := Rev(CD); ProgPD(Rev(CD)).CurDir = Rev(CD) \wedge$
$\qquad\quad PickupDropoff(Rev(CD))$
$\equiv$< Definition of composition >
$\quad wp.CurDir := Rev(CD).(wp.ProgPD(Rev(CD)).CurDir =$
$\qquad\quad Rev(CD) \wedge PickupDropoff(Rev(CD)))$
$\equiv$< Conjunctivity of wp >
$\quad wp.CurDir := Rev(CD).(wp.ProgPD(Rev(CD)).CurDir =$
$\qquad\quad Rev(CD) \wedge wp.ProgPD(Rev(CD)).PickupDropoff(Rev(CD)))$
$\equiv$< Proof of $ProgPD$, identity of $\wedge$ >
$\quad wp.CurDir := Rev(CD).(wp.ProgPD(Rev(CD)).CurDir =$
$\qquad\quad Rev(CD))$
$\equiv$< Lemma 1 (see section 5.3) >

$wp.CurDir := Rev(CD).CurDir = Rev(CD)$
$\equiv <$Definition of $:=$, $x = x >$
true

(6) $\{B_{12}\}\, S_{12}\, \{R_1\}$
$wp.S_{12}.R_1$
$\equiv <$Assume $B_{12} \equiv \neg\,(ReasonToGo(CD) \vee ReasonToGo(Rev(CD))) >$
$wp.S_{12}.CurDir = HALT \;\wedge\; PickupDropoff(HALT)$
$\equiv < S_{12} = CurDir := HALT; ProgPD(HALT) >$
$wp.CurDir := HALT; ProgPD(HALT).CurDir = HALT \;\wedge$
$PickupDropoff(HALT)$
The rest of the proof is identical to the derivation of S_{11}, with Rev(CD) replaced by *HALT*.

(7) Disjunction of B_{0j}
$B_{00} \vee B_{01} \vee B_{02}$
$\equiv < B_{02} = \neg\,(B_{00} \wedge B_{01}) >$
$B_{00} \vee B_{01} \vee \neg\,(B_{00} \vee B_{01})$
$\equiv < p \vee \neg p >$
true

(8) $\{B_0\}S_0\{R\}$
$wp.S_0.R$
$\equiv <$Assume $B_0 >$
$wp.S_0.R_0$
$\equiv <$Definition of **if**$>$
$(B_{00} \;\vee\; B_{01} \;\vee\; B_{02}) \;\wedge\; \{B_{00}\}\, S_{00}\, \{R_0\} \;\wedge\; \{B_{01}\}\, S_{01}\, \{R_0\} \wedge$
$\{B_{02}\}\, S_{02}\, \{R_0\}$
$\equiv <$(7), (1), (2), (3), assuming $B_0 >$
true

(9) Disjunction of B_{1j}
$B_{10} \vee B_{11} \vee B_{12}$
$\equiv < B_{12} = \neg\,(B_{10} \;\wedge\; B_{11}) >$
$B_{10} \vee B_{11} \vee \neg\,(B_{10} \vee B_{11})$
$\equiv < p \;\vee\; \neg p >$
true

(10) $\{B_1\}S_1\{R\}$
$wp.S_1.R$
$\equiv <$Assume $B_1 >$
$wp.S_1.R_1$
$\equiv <$Definition of **if**$>$

$(B_{10} \vee B_{11} \vee B_{12}) \wedge \{B_{10}\}\ S_{10}\,\{R_1\}\ \wedge\ \{B_{11}\}\,S_{11}\,\{R_1\}\wedge$
$\{B_{12}\}\,S_{12}\,\{R_1\}$
$\equiv$<(9), (4), (5), (6), assuming B_1 >
true

(11) Disjunction of B_i
$CD = HALT\ \vee\ CD \neq HALT$
$\equiv< X \vee \neg X >$
true

(12) $\{Q\}\,S\,\{R\}$
$wp.S.R$
$\equiv$<Definition of **if**>
$(B_0\ \vee\ B_1)\ \wedge\ \{B_0\}\,S_0\,\{R\}\ \wedge\ \{B_1\}\,S_1\,\{R\}$
$\equiv$<(11), (8), (10)>
true

5.2 $ProgPD$

5.2.1 Specification

$\{Q : \textbf{true}\}\,ProgPD(Dir)\,\{PickupDropoff(Dir)\}$

5.2.2 Guarded command code

$\{P : \textbf{true}\}$
if $C_0 : (CurFloor, Dir) \notin O \rightarrow$
T_0 : **if**$C_{00} : CurFloor \notin I\ \rightarrow\ T_{00}$: **skip**
[] $C_{01} : CurFloor\ \in I\ \ \rightarrow T_{01} : InsideReqSet := I - CurFloor$
fi
[] $C_1 : (CurFloor, Dir) \in O\ \rightarrow$
T_1 : **if** $C_{10} : CurFloor \notin I\ \rightarrow$
$T_{10} : InsideReqSet, OutsideReqSet := I + GetDests(),$
$O - (CurFloor, Dir)$
[] $C_{11} : CurFloor \in I\ \rightarrow$
$T_{11} : InsideReqSet, OutsideReqSet := I + GetDests() -$
$CurFloor, O - (CurFloor, Dir)$
fi
fi
$\{U : PickupDropoff(Dir)\}$

5.2.3 Proof

(1) Disjunction of guards

$C_0 \vee C_1$

$\equiv <$Substitution$>$

$(CurFloor, Dir) \notin O \vee (CurFloor, Dir) \in O$

$\equiv <$Definition of $\notin >$

$\neg\,((CurFloor, Dir) \in O) \vee (CurFloor, Dir) \in O$

$\equiv < \neg p \vee p >$

true

$C_{00} \vee C_{01} \equiv C_{10} \vee C_{11} \equiv$ **true** by similar logic.

(2) $\{C_{00}\}\,\mathrm{T}_{00}\,\{U_0\}$

$wp.\mathbf{skip}.U_0$

$\equiv <$Definition of **skip**$>$

U_0

$\equiv <$Assume $C_{00} >$

true

(3) $\{C_{01}\}\,\mathrm{T}_{01}\,\{U_0\}$

$wp.InsideReqSet := I - CurFloor.U_0$

$\equiv <$Assume $C_{01} >$

$wp.InsideReqSet := I - CurFloor.InsideReqSet = I - CurFloor$

$\equiv <$Definition of $:=$, $x = x >$

true

Proofs of T_{10} and T_{11} are identical in shape, with appropriate subscripts.

(4) $\{C_0\}\,\mathrm{T}_0\,\{U\}$

$wp.T_0.U$

$\equiv <$Definition of **if**$>$

$(C_{00}\ \vee\ C_{01})\ \wedge\ \{C_{00}\}\,T_{00}\,\{U_0\} \wedge \{C_{01}\}\,T_{01}\,\{U_0\}$

$\equiv <$ (1), (2), (3) $>$

true

Proof of T_1 proceeds indentically, using appropriate subscripts.

(5) $\{P\}\,ProgPD\,\{U\}$

$wp.ProgPD.U$

$\equiv <$Definition of **if**, Assume appropriate guards$>$

$(C_0\ \vee\ C_1) \wedge \{C_0\}\,T_0\,\{U_0\} \wedge \{C_1\}\,T_1\,\{U_1\}$

$\equiv <$(1), (4)$>$

true

5.3 Proof of Lemma 1

Lemma 1: $(wp.ProgPD(D).CurDir = D) \equiv CurDir = D$

$wp.ProgPD(D).CurDir = D$
$\equiv$<Substitution for $ProgPD$, definition of **if**>
$(B_0 \lor B_1) \land \{B_0\}\, S_0\, \{CurDir = D\} \land \{B_1\}\, S_1\, \{CurDir = D\}$
$\equiv$<(1) from proof of $ProgPD$, identity of $\land$ >
$\{B_0\}\, S_0\, \{CurDir = D\} \land \{B_1\}\, S_1\, \{CurDir = D\}$
$\equiv$<Substitution for S_0 and S_1, definition of **if**>
$(B_{00} \lor B_{01}) \land \{B_{00}\}\, S_{00}\, \{CurDir = D\} \land$
$\{B_{01}\}\, S_{01}\, \{CurDir = D\} \land (B_{10} \lor B_{11}) \land$
$\{B_{10}\}\, S_{10}\, \{CurDir = D\} \land \{B_{11}\}\, S_{11}\, \{CurDir = D\}$
$\equiv$<(1) from proof of $ProgPD$, identity of $\land$ >
$\{B_{00}\}\, S_{00}\, \{CurDir = D\} \land \{B_{01}\}\, S_{01}\, \{CurDir = D\} \land$
$\{B_{10}\}\, S_{10}\, \{CurDir = D\} \land \{B_{11}\}\, S_{11}\, \{CurDir = D\}$
$\equiv$<Substitution for S_{xy}, definition of Hoare triple>
$(B_{00} \Rightarrow wp.\mathbf{skip}.CurDir = D) \land$
$(B_{01} \Rightarrow wp.InsideReqSet := I - CurFloor.CurDir = D) \land$
$(B_{10} \Rightarrow wp.InsideReqSet, OutsideReqSet := I + GetDests(),$
$O - (CurFloor, Dir).CurDir = D) \land$
$(B_{11} \Rightarrow wp.InsideReqSet, OutsideReqSet := I +$
$GetDests() - CurFloor, O - (CurFloor, Dir).CurDir = D)$
$\equiv$<Definition of **skip**, :=>
$(B_{00} \Rightarrow CurDir = D) \land (B_{01} \Rightarrow CurDir = D) \land$
$(B_{10} \Rightarrow CurDir = D) \land (B_{11} \Rightarrow CurDir = D)$
$\equiv$< $(p \Rightarrow q) \land (r \Rightarrow q) \equiv (p \lor r \Rightarrow q)$ >
$(B_{00} \lor B_{01} \Rightarrow CurDir = D) \land (B_{10} \lor B_{11} \Rightarrow CurDir = D)$
$\equiv$<(1) from proof of $ProgPD$, left identity of $\Rightarrow$>
$CurDir = D \land CurDir = D$
$\equiv$<idempotency of $\land$ >
$CurDir = D$

6 Implementation

The guarded command code resulting from the above derivations was used to generate code for a simulation to demonstrate the operation of the elevator system. The simulation was coded in Microsoft Visual C++ using the Microsoft Foundation Classes. As discussed in the design above, most of the code was not written with the aid of formal methods. For those procedures which code was derived, the code was translated into C++ in a very direct manner. Predicates were translated as function calls, multiple assignments were changed into sequences of single assignments, and non-determinism in **if** statements was resolved in the order which the guards appeared. The resulting code is as follows:

```
void CElevatorSystem::ProgPD(int CurFloor, int Dir)
{
   if (!InOutReqSet(CurFloor, Dir)) {
      if (InInReqSet(CurFloor)) {
         while (m_Elevator.FindStop(CurFloor)) {
            m_Elevator.RemoveStop(CurFloor);
         }
      }
   } else if (!InInReqSet(CurFloor)) {
         GetDests(CurFloor, Dir);
         RemoveRequests(CurFloor, Dir);
      } else {
         while (m_Elevator.FindStop(CurFloor)) {
            m_Elevator.RemoveStop(CurFloor);
         }
         GetDests(CurFloor, Dir);
         RemoveRequests(CurFloor, Dir);
   }
}

int CElevatorSystem::Rev(int Dir)
{
   if (Dir == UP) {
      return DOWN;
   } else if (Dir == DOWN) {
      return UP;
   } else
      return HALT;
}

int CElevatorSystem::UpdateElevator()
{
   int CD, CurFloor;

   CD = m_Elevator.GetCurrentDirection();
   CurFloor = m_Elevator.GetCurrentFloor();

   if (CD == HALT) {
      if (ReasonToGo(UP)) {
         m_Elevator.SetCurrentDirection(UP);
         ProgPD(CurFloor, UP);
      } else if (ReasonToGo(DOWN)) {
         m_Elevator.SetCurrentDirection(DOWN);
         ProgPD(CurFloor, DOWN);
      }
   } else if (ReasonToGo(CD)) {
```

```
        ProgPD(CurFloor, CD);
      } else if (ReasonToGo(Rev(CD))) {
        m_Elevator.SetCurrentDirection(Rev(CD));
        ProgPD(CurFloor, Rev(CD));
      } else {
        m_Elevator.SetCurrentDirection(HALT);
        ProgPD(CurFloor, HALT);
      }
    }
    return m_Elevator.GetCurrentDirection();
}
```

The formal methods approach of code derivation proved beneficial in implementing the code for this algorithm. The first run of the program was not error-free, but the errors encountered were related exclusively to the code not derived from a specification. *ReasonToGo* was the only procedure involved in the scheduling algorithm itself that contained an error in logic, and it was the only procedure in the algorithm that was not formally derived. Further testing of the code showed the sections generated by the application of formal methods to be correct.

7 Liveness Properties of the Elevator

There are two liveness requirements listed in section 2: requirement 4 and 5. The fifth requirement, service all inside requests eventually with all floors being serviced sequentially in the direction of travel, is the basis for the definition of *ReasonToGo*. The elevator will not change direction until there is no reason to continue in the current direction; namely, either an outside request from or an inside request for a floor that can be reached by maintaining the current direction of the elevator. Since the elevator doesn't change direction, all inside requests in the $CurDir$ direction will be satisfied. Assuming a finite number of floors, $ReasonToGo(CurDir)$ must eventually become **false**, at which time $ReasonToGo(Rev(CD))$ becomes **true**, and will service all inside requests in the opposite direction.

For the fourth requirement, service all outside requests eventually with all floors given equal priority, four situations must be examined.

1. External request from the j^{th} floor with the elevator on the i^{th} floor, $i < j$, and $CurDir$ is UP.

2. External request from the i^{th} floor with the elevator on the j^{th} floor, $i < j$, and $CurDir$ is $DOWN$.

   In these two cases, $ReasonToGo(CurDir)$ is **true** and therefore the elevator continues its movement in the direction of $CurDir$; eventually encountering the requesting floor.

3. External request from the i^{th} floor with the elevator on the j^{th} floor, $i < j$, and $CurDir$ is UP.

4. External request from the j^{th} floor with the elevator on the i^{th} floor, $i < j$, and $CurDir$ is $DOWN$.

   In these two cases, the elevator is traveling in the opposite direction of the request which establishes $ReasonToGo(Rev(CurDir))$. $ReasonToGo(CurDir)$ must eventually become **false** since there exists a finite number of floors between the elevator's current floor and the end floor. At this time, the elevator will change direction and continue movement to the requesting floor since $ReasonToGo(CurDir)$ will at least remain **true** until the elevator encounters the requesting floor.

8 Comparisons to Presented Elevator Specifications

Many solutions were offered to the elevator scheduling problem as presented at the IEEE Specification & Design workshop. Other solutions to the problem dealt primarily with the physical requirements of the elevator. One concentrated on elevator lights (on/off), doors (open/closed), and buttons (press/release) [WKS87, p. 266]. Another included the operation of the emergency button in the design [SD87, p. 23]. All of these aspects are important in physical operation of an elevator, but ignore the essence of designing an algorithm to control the movement of the elevator itself. Furthermore, these solutions did not include in their specifications the maintenance of request sets for inside and outside the elevator. Tracking such information is crucial in designing an elevator that will minimize waiting time for passengers.

One published solution to this problem gives priority to the passengers in the elevator and those waiting on floors that the elevator passes by, not allowing the elevator to check for requests on other floors. If no one in the elevator ever requests a floor where people are waiting, and the elevator never passes that floor, then those people will never be picked up. This is not likely to happen; however, it ignores a serious part of the functionality of an elevator by not guaranteeing that all requests are eventually serviced [SD87, p. 26].

The approach to the elevator scheduling problem presented is more logical in nature, focusing on the actual algorithm that determines the direction the elevator should travel dependent upon the requests of people waiting for and riding inside the elevator. This quintessential part of the problem was captured mathematically in a manner that permits proof of the correctness of the code derived from it. Implementation greatly benefits from code that is correct from the onset. Furthermore, the solution is much more complete, detailed, and exhaustive than others presented at the IEEE Workshop.

This problem was also given to two classes of third-semester programming students at Miami University to solve. One class had received two semesters of training in formal methods application, while the other class had none. In general, the teams of students who applied formal methods in solving the problem created code that was shorter, contained fewer errors, and used less "case-based" analysis.

9 Conclusions

Formal methods are effective tools in software design, as is demonstrated by the solution presented above. Not only do formal methods provide a clear notation for problem specification, but they also provide the means to prove the correctness of the code that is derived from the specification. In a time when performance is key, and so many mission critical systems depend on the performance of computer software, the ability to know undoubtedly that software meets the criteria of its design is imperative. Formal methods has the potential to change software design from a practice of translating vague designs into partially matching code to a science that is explicit in design and exact in implementation.

References

[Coh90] E. Cohen. *Programming in the 1990s: An Introduction to the Calculation of Programs.* Springer–Verlag, 1990.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[IEE87] IEEE. *IEEE Workshop on Software Specification and Design*, 1987.

[SD87] M. D. Schwartz and N. M. Delisle. Specifying a lift control system with CSP. In *IEEE Workshop on Software Specification and Design* [IEE87].

[WKS87] J. C. P. Woodcock, S. King, and I. H. Sorensen. Mathematics for specification and design: The problem with lifts... In *IEEE Workshop on Software Specification and Design* [IEE87].