Technical Report

CMU/SEI-89-TR-24
ESD-89-TR-32

Temporal Logic Case Study

William G. Wood

August 1989

Technical Report
CMU/SEI-89-TR-24
ESD-89-TR-32
August 1989

Temporal Logic Case Study

William G. Wood

Software Methods Program

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Temporal Logic Case Study

Abstract. This report is a case study applying temporal logic to specify the operation of a bank of identical elevators servicing a number of floors in a building. The goal of the study was to understand the application of temporal logic in a problem domain that is appropriate for the method, and to determine some of the strengths and weaknesses of temporal logic in this domain. The case study uses a finite state machine language to build a model of the system specification, and verifies that the temporal logic specifications are consistent using this model. The specification aspires to be complete, consistent, and unambiguous.

1. Introduction

Software is consuming larger and larger portions of both the development and maintenance budgets for large defense systems. One major reason for this is that the system specification and the derived software specifications are often incomplete, inconsistent, and ambiguous. The cost of detecting and repairing a flaw in the specification at test, integration, or installation is very high. Not only are there executable objects to be changed; user and maintenance manuals, and many layers of specification, design, and test documents must be changed as well. To make the specification more complete and flexible, the specification of the system and the software must be improved in a number of ways. Some of these improvements are administrative and affect the acquisition and management of both the development process and the product development life cycle. Specifications may also be improved by using more systematic and well-documented methods such as Hatley/Pirbhai, Ward/Mellor, JSD, Statecharts, and other such approaches; most of these are automated and supported by one or more commercial tool sets. In addition to the above, there is a large effort underway in the development of software engineering environments to support both the engineers using these tool sets and the management personnel associated with the development effort.

Most of these efforts are still based on experience; they rely on the software engineer to craft his specification carefully, and provide little of the assistance from scientific and mathematical theory that the other engineering fields enjoy. This is beginning to change, however, as the mathematical theories supporting software specification of both sequential and reactive systems are now emerging. From these so-called formal methods, the software engineer can, in the future, hope to produce specifications which are verifiably consistent. There are many such formal specification techniques being proposed, and this report will not describe and compare them. There are few instances of application engineers using these methods to specify a large complex system and formally verify that the specification is consistent. This report specifies the reactive part of a relatively large and complex system. At some time in the future, this specification can be used to determine to what extent verification is possible. Even if verification cannot be realized, such specifications are still more precise than English language descriptions, and should contain fewer flaws. Therefore, they can be used to specify systems and software in a more rigorous manner than the current state of the art, until formal verification is achievable.

This report uses temporal logic to specify the operation of a bank of elevators servicing a number of floors. We describe the elevator only in terms of the user of the elevator, describing the requirements in terms of actions he takes, and indicators that the system displays to him. Hence the users push summons buttons on each floor to get an elevator to move to the floor, and when it arrives and opens its doors, they step inside and push destination buttons to get it to move to their desired destination floor. When they depress the push buttons, they light up, and become unlit in an appropriate manner when the elevator travels between floors. The specification is produced at two levels. The first level is that of the end-users describing how they wish the elevator system to operate, and the second level is that of the system engineer adding behavior to provide equitability and efficiency in the performance.

The system specification is defined from the point of view of an applications engineer, rather than the point of view of an expert in temporal logic. For this reason, the chosen elevator problem contains most of the signals and indicators seen in modern elevators, since the author was interested in the problem of specifying a complex problem using temporal logic. In addition to the temporal logic specification, a model of the specification was also built, using the State Machine Language (SML) developed by Ed Clarke and others at CMU, and described in [Clarke 87]. The SML model was then used to verify the consistency of the formulas, using a toolset called EMC, also described in the previous reference.

1.1. The Elevator Problem

There is no single elevator problem statement from which different solutions have been derived, and the elevator problem chosen here, once again, is somewhat different from these others. The characteristics of the elevator problem to be specified in detail in this report are sketched below. Later sections detail the operational specification.

1. There are many elevators serving multiple floors.
2. On board each elevator is a set of destination push buttons, one for each floor, which backlight when depressed, and remain lit until arrival at the selected floor.
3. On board each elevator are two directional signal lights, one for going up, and the other for going down.
4. On board each elevator is a set of lights, one for each floor. One of these lights is always lit, indicating the elevator is at that floor.
5. On each floor there are two summons push buttons, one for summoning the elevator to go up, and the other to go down. These backlight when pushed, and remain lit until an elevator arrives that will go in the selected direction. The top and bottom floors each have only a single summons push button.
6. On each floor, beside each elevator are two floor directional lights, one showing the direction the elevator will take. When an elevator arrives at the floor, the appropriate light shows the direction the elevator will take when leaving the floor. The top and bottom floors have only a single directional light each.

7. Each elevator has doors which are either closed or not closed. Opening, closing, or emergency stops are not considered. On each floor, there are doors for each elevator. Both the elevator doors and the floor doors have to be open for people to enter or leave the elevator at a floor.
8. The specification is not concerned with what happens under failure conditions.

1.2. The Scheduler

To describe the elevator system from the user's viewpoint, the user's expectations on equality of service must be included in some way. This is an optimization problem, and depends on many factors, such as statistics of usage and current requests for service. The solution often contains more than one component. For example, at morning rush hour, the elevators may "home" to the ground floor, while at the end of the lunch hour, they may home to the cafeteria floor. All of these problems have to be addressed, but their solutions have to be abstracted, so that the control of the elevators can be defined independently of the final scheduling strategies (which may vary with the time of day or the level of service, and may change over relatively short time periods, as the tenants of the building, or their elevator usage habits change). The approach taken in this paper is to treat all of the elevators as autonomous, with a few interfaces to a scheduler. The details of the scheduler are undefined except for its interfaces, and the constraints on the fairness of that behavior. Hence each elevator will continue to provide its simple-minded service unless the scheduler issues a command to override this service to improve the efficiency of operation. One additional advantage to such an approach is that it is appropriate for distributed fault-tolerant operation. If the scheduler is not in service, or a processor loses communications with the scheduler, service will continue to be provided in a predictable, though sub-optimal, manner.

1.3. Review of Other Elevator Problem Specifications

The elevator subsystem was chosen for three reasons.

- There are a number of researchers who have previously chosen to describe specifications of elevators. Comments on some of these specifications are included below. Each such specification is for a different type of elevator system, making comparisons among different specifications awkward.
- Everyone intuitively understands an elevator's operation, so that little explanation of the problem domain is needed.
- The elevator is quite a complicated reactive system, with many identical components (buttons, lights) which require similar behavior from the elevator.

Barringer [Barringer 87] describes a multiple elevator system (using temporal logic) that is somewhat simpler than the one used here. He has no lights indicating on-board direction, or floor arrival lights, and the lights in his definition are extinguished when the doors are opening, rather than in advance of arrival. He does, however, include a four-state door (closed, opening, open, and closing), and the logic to cope with "foot in the door" and other emergency conditions.

Woodcock and others [Woodcock 87] use the theory of Communicating Sequential Processes (CSP) from [Hoare 85] to describe the operation of a single elevator system at a single floor. They use the trace capabilities included in that approach to define such an operation. The operation considers a single summons button at the floor, a two-state door, and assumes that the light is extinguished when the doors are opened. There are no considerations of elevators moving between floors, or signals within elevators.

Schwartz and others [Schwartz 87] also use CSP to specify the elevator system for multiple lifts at many floors, with up and down summons buttons at each floor, an emergency button and a "back in service" consideration. There is an arrival condition specified, which corresponds closely to the floor arrival lights in each direction. They do not, however, have directional lights on the elevator. They use the trace capability of CSP to specify the operation of the elevator in terms of queues for service going up and down, where queue entries are made by depressing the push buttons, and removal from the queue occurs when the elevator visits a floor.

In the above two cases (Barringer and Schwartz), where the elevator problem includes onboard signals and considers elevator motion, the authors considered primitive scheduling and coordination necessary to the definition of the problem.

2. Terminology

This chapter describes the terminology used throughout the document. The chapter introduces the notations used for temporal operators, describes the sets and sequences used, introduces some special notations to make the formula more concise and readable, and defines the variables used throughout the document. In addition, the safety conditions are described, since they are mostly restrictions on the values of the variables.

2.1. Temporal Logic

Temporal logic is a well-developed branch of modal logic [Hughes 68], and is thoroughly described in [Rescher 71]. Temporal logic has been proposed as applying both to the specification and verification of program behavior, and to the specification of system behavior. The first significant proposals for using temporal logic to describe program behavior came from Pnueli, and his later paper [Pnueli 85] is a good survey of the field. Other good general descriptions of the applicability of temporal logic are [Lamport 86] and [Lamport 83]. There have been numerous papers from Clarke and others demonstrating this applicability; [Clarke 86] and [Clarke 87] describe the development of tools to automate the process. A more complete description of the specific temporal logic model used in this report is given in Appendix 1.

The specification of a system in temporal logic is usually divided into *safety conditions, liveness conditions, and fairness conditions*; and we shall follow this breakdown.

1. The *safety* conditions are those which must not occur in operation of the system. In the elevator specification, for example, the up directional light and the down directional light cannot both be lit simultaneously. Another example of a safety condition is that a backlit push button must not come on without the button having been depressed.
2. The *liveness* conditions specify what the system must do. For example, when a push button is depressed, the backlight must come on and stay on until an elevator is present at the selected floor.
3. The *fairness* conditions describe how nondeterministic specifications are to be resolved. For example, if the whole elevator system is dormant (no requests), and two destination buttons are depressed simultaneously, we can specify the action to be taken nondeterministically. The fairness condition could express that we do not select the up direction every time such a race condition occurs.

Temporal logic allows the system operation to be defined nondeterministically, and to include fairness conditions such that one course of action does not necessarily dominate. The users section of this report describes the system deterministically, and the systems section demonstrates how one can specify either deterministically or otherwise. In the case of the elevator, the difference is not significant. However, in general, it is best to specify a system nondeterministically, since this is the most general specification and postpones the operational details until later in the development process. Often the advantage in thus postponing such a decision is that there may be a distinct benefit of one way over an other, which does not become obvious until later in the life cycle.

2.2. Temporal Operators

The expected logical operators used throughout are **and** (∧), **or** (∨), **not** (¬), **equivalence** (≡), and **implies** (→).

The temporal operators to be used are listed below, and the bold word in the description of the operator is the word to be visualized when reading the formula.

1. ◊ **a** : This means that **eventually a** will be true.
2. []**a** : This means that **henceforth a** is true.
3. °**a** : This means that at the **next** state (instant in time) **a** will be true.
4. •**a** : This means that at the **previous** state (instant in time) **a** was true.
5. **a** ⇒ **b** : This is read as **strictly implies**, and it means that henceforth, if **a** is true, then **b** is true. **(a ⇒ b)** $=_{df}$ []**(a → b)**.
6. **a U b** : This means that **a** is true **until** the state (instant in time) when **b** occurs. The strong **until** implies that **b** will eventually occur. In addition, **a** must always be true in the present.
7. **a = b**: This means that **a** is strictly equivalent to **b**. **(a = b)** $=_{df}$ [] **(a ≡ b)**.

2.3. Sets and Sequences

The sets in the system are:

1. Set of elevators : **I**; set of directions : **J = {up, down}**.
2. Sequences of floors : **M = [1,2, ..., |M|]**; M^-**[1,2, ..., |M-1|]**; M_- **= [2, ..., |M|]**.
3. Set of destination push buttons : **IM** ≡ **I*M**.
4. Set of on-floor summons buttons : **MJ** ≡ **M*J (1, down) - (|M|, up)**.
5. Set of on-floor arrival lights : **IMJ** ≡ **I*M*J - (**∀ **i** ∈ **I, (i,1,down)) - (**∀ **i** ∈ **I, (i,|M|,up))**.

2.4. Special Notation

1. It is convenient to define a condition at a floor in the current direction the elevator is traveling beyond the current floor (for example if the current floor is 4, and the current direction is up, then any floor number greater than 4).

   ∃ **m** $\oplus^{j}$ **m1** ≡ **(m,m1)** ∈ **M*M, ((m > m1)** ∧ **(j = up))** ∨ **((m < m1)** ∧ **(j = down))**.
2. It is useful to indicate the concept of "in the opposite direction".

   ¬**j** ∧**(j=up)** ≡**down**

   ¬**j** ∧**(j = down)** ≡ **up**

2.5. Variable Definitions

The indicators and signals visible to the end user are each described below, and the variable definitions for each are given.

The on-board variables are defined below.

1. For each elevator **i** there is a light for each floor **m**, indicating that the elevator is in the vicinity of that floor. $\mathbf{el\_position}_{\mathbf{i,m}}$
2. For each elevator there is a light signaling that the elevator is traveling up, and another that it is traveling down. $\mathbf{el\_direction}_{\mathbf{i,j}}$ where **j= {up, down}**
3. Each elevator's doors are either opened or closed. $\mathbf{el\_doors\_closed}_{\mathbf{i}}$
4. For each elevator, there is a set of destination push buttons, one for each floor **m**. When a push button is depressed, it indicates that the traveler wishes the elevator to stop the elevator at the selected floor. $\mathbf{el\_dest\_depr}_{\mathbf{i,m}}$
5. Each destination push button is backlit to indicate that a request for service has been made, and is yet to be honored. The button is also backlit while it is depressed, until it is released. $\mathbf{el\_dest\_lit}_{\mathbf{i,m}}$

The on-floor variables are enumerated below.

1. On each floor there is a summons button in each direction, with the obvious exception of the first and last floors. They are depressed to indicate that the traveler wishes an elevator to come to floor **m**, and take him in the chosen direction **j**. $\mathbf{fl\_summ\_depr}_{\mathbf{m,j}}$
2. Each summons push button is backlit to indicate that it has been depressed. $\mathbf{fl\_summ\_lit}_{\mathbf{m,j}}$
3. On each floor, there are two indicators next to each elevator. These indicate that an elevator is stopping or is stopped at the floor. These are defined for all floors; though the indicators for the top floor going up, and the bottom floor going down are not ever set or reset, they are defined for consistency of terminology. $\mathbf{fl\_direction}_{\mathbf{i,m,j}}$
4. Each elevator has a set of doors on each floor. $\mathbf{fl\_doors\_closed}_{\mathbf{i,m}}$

2.6. Scheduler Interfaces

The elevators must have a scheduler deciding which elevators stop to service summons buttons, or move to service summons button demands when some of the elevators are dormant. The functionality of the scheduler itself does not concern us here, since this belongs to the realm of optimization theory, but the scheduler's interfaces are important. The signals described below represent the results of the scheduler. The scheduler could obviously disturb operations—for example, it could issue a command to each elevator to ignore the requests for a summons button. To evaluate the operation of the system, we must put restrictions on the expected behavior of the scheduler. These are expressed as fairness conditions in a later section. Each elevator will work autonomously, stopping at a floor if the appropriate conditions are set. These conditions are described in a later section. The scheduler can override some of these conditions in the interest of more efficient operation of the elevator system.

1. The scheduler can override the decision of an elevator **i** to stop at a floor **m** in some cases. The precise description of the cases will be given later. $\mathbf{sch\_stop\_override_{i,m,j}}$
2. The scheduler can prevent an elevator **i** from leaving a floor **m** in the direction **j**, in response to a summons request. $\mathbf{sch\_start\_override_{i,m,j}}$
3. The scheduler can cause the elevator **i** traveling in direction **j** to stop at a floor **m**. $\mathbf{sch\_stop_{i,m,j}}$

3. Safety Conditions

The safety conditions describe conditions of the operation which are disallowed. Most of the safety conditions in this context are necessary to describe conditions that are mutually inconsistent, such as the fact that the elevator cannot be going up and down at the same time. There are a few, however, forbidding absurd operations such as lights coming on of their own volition.

3.1. On-Board Safety Considerations

1. The elevator must only indicate movement in one of the two directions, or no direction. It cannot simultaneously show movement in both directions. If both indicators are unlit (false), then this means that the elevator is stopped.

   $\forall\, \mathbf{i} \in \mathbf{I},\ [\,]\ \neg\ (\mathbf{el\_direction}_{\mathbf{i,\ up}} \wedge \mathbf{el\_direction}_{\mathbf{i,down}})$
2. The elevator can only be shown to be at one floor at any time.

   $\forall\, \mathbf{i} \in \mathbf{I},\ \exists\, \mathbf{m1} \in \mathbf{M}\ ,\ \mathbf{el\_position}_{\mathbf{i,m1}} \Rightarrow (\forall\, \mathbf{m} \in (\mathbf{M - m1}),\ \neg\ \mathbf{el\_position}_{\mathbf{i,m}})$
3. When an elevator destination push button is lit, it must previously been depressed or lit. This prohibits the light from coming on autonomously.

   $\forall\, (\mathbf{i,m}) \in \mathbf{IM},\ \mathbf{el\_dest\_lit}_{\mathbf{i,m}} \Rightarrow \bullet\ (\mathbf{el\_dest\_lit}_{\mathbf{i,m}} \vee \mathbf{el\_dest\_depr}_{\mathbf{i,m}})$

3.2. On-Floor Safety Considerations

1. For each elevator only one arrival light at one floor can be on at any time. Having no arrival light on is always valid. The light can only be on when the elevator is in the vicinity of the floor.

   $\forall\, (\mathbf{i,m1,j1}) \in \mathbf{IMJ},\ \mathbf{fl\_direction}_{\mathbf{i,m1,j1}} \Rightarrow \mathbf{el\_position}_{\mathbf{i,m1}} \wedge (\forall\, (\mathbf{m,j}) \in (\mathbf{MJ - (m1,j1)}),\ \neg\ \mathbf{fl\_direction}_{\mathbf{i,m,j}})$
2. In addition, there are two variables which are always false.

   $\forall\, \mathbf{i} \in \mathbf{I},\ [\,]\ (\neg\ \mathbf{fl\_direction}_{\mathbf{i,1,down}} \wedge \neg\ \mathbf{fl\_direction}_{\mathbf{i,M,up}})$
3. The floor summons lights must not be lit, unless at the previous state they were lit or the push button was depressed.

   $\forall\, (\mathbf{m,j}) \in \mathbf{MJ},\ \mathbf{fl\_summ\_lit}_{\mathbf{m,j}} \Rightarrow \bullet\ (\mathbf{fl\_summ\_lit}_{\mathbf{m,j}} \vee \mathbf{fl\_summ\_depr}_{\mathbf{m,j}})$
4. The floor doors can be open only if the elevator doors are open and the elevator is at the floor.

   $\forall\, (\mathbf{i,m}) \in \mathbf{IM},\ [\,]\ \neg\ \mathbf{fl\_doors\_closed}_{\mathbf{i,m}} \equiv \neg\ \mathbf{el\_doors\_closed}_{\mathbf{i}} \wedge \mathbf{el\_position}_{\mathbf{i,m}}$

Note that this prohibits more than one door being open at one time for any elevator, since only one of the $\mathbf{el\_position}_{\mathbf{i,m}}$ values can be true at any instant.

3.3. Scheduler Safety Considerations

1. It would be ridiculous to have a scheduled stop at the same time as as override in the same direction.

   $\forall$ **(i,m,j)** $\in$ **IMJ,** [] $\neg$ **(**$\mathbf{sch\_stop\_override_{i,m,j}} \wedge \mathbf{sch\_stop_{i,m,j}}$**)**
2. In addition, the elevator must always stop at the top and bottom floors.

   $\forall$ **i** $\in$ **I,** [] **(**$\mathbf{sch\_stop_{i,M,up}} \wedge \mathbf{sch\_stop_{i,M,down}} \wedge \mathbf{sch\_stop_{i,1,down}} \wedge \mathbf{sch\_stop_{i,1,up}}$**)**
3. The scheduler will never have all elevators with a start override in the same direction at the same time.

   $\forall$ **j** $\in$ **J,** [] $\neg$ **(**$\forall$ **i** $\in$ **I,** $\mathbf{sch\_start\_override_{i,m,j}}$**)**

4. User's Viewpoint

We start by describing the user's view of how the elevator should operate. This view is restricted to what the user can do and see. This allows for many different implementations of elevator operation.

1. When a summons push button is depressed at floor **m** for direction **j**, it is also backlit, and will remain backlit until a floor directional light shows that any elevator **i** is stopping at floor **m** in the direction **j**. The button is always lit while depressed, indicating to the user that the system is responding to the request for service even if there is an elevator already at the floor with its arrival light set in the requested direction.

   $\forall$ **(m,j)** $\in$ **MJ,** $\mathbf{fl\_summ\_depr_{m,j}} \Rightarrow \mathbf{fl\_summ\_lit_{m,j}}$ **U (** $\exists$ **i** $\in$ **I,** $\mathbf{fl\_direction_{i,m,j}}$**)**

   $\forall$ **(m,j)** $\in$ **MJ, (**$\neg$ $\mathbf{fl\_summ\_depr_{m,j}} \wedge$ **(** $\exists$ **i** $\in$ **I,** $\mathbf{fl\_direction_{i,m,j}}$**)** $\Rightarrow \neg$ $\mathbf{fl\_summ\_lit_{m,j}}$
2. When the floor directional light is on, the elevator is at the floor, and eventually its doors will open. The floor directional light comes on shortly before the doors open, to give the user time to move to the appropriate elevator. This formula explicitly states that after turning a floor directional light on, the elevator cannot change floors and the floor directional light stays on until the doors have opened.

   $\forall$ **(i,m,j)** $\in$ **IMJ,** $\mathbf{fl\_direction_{i,m,j}} \Rightarrow$**(** $\mathbf{fl\_direction_{i,m,j}} \wedge \mathbf{el\_position_{i,m}}$**) U (**$\bullet$ $\neg$ $\mathbf{fl\_doors\_closed_{i,m}} \wedge \mathbf{fl\_doors\_closed_{i,m}}$**)**
3. When the doors eventually close, the floor directional indicator lights are also extinguished.

   $\forall$ **(i,m,j)** $\in$ **IMJ, (**$\bullet$ $\neg$ $\mathbf{fl\_doors\_closed_{i,m}} \wedge \mathbf{fl\_doors\_closed_{i,m}}$**)** $\Rightarrow$ $\neg\mathbf{fl\_direction_{i,m,j}}$
4. When the user enters an elevator, with its floor directional light set, he expects the directional light in the elevator to be in the same direction as that indicated by the external floor directional light.

   $\forall$ **(i,m,j)** $\in$ **IMJ,** $\mathbf{fl\_direction_{i,m,j}} \Rightarrow \mathbf{el\_direction_{i,j}}$
5. When an on-board destination push button is depressed, it is also backlit, and will remain backlit until an associated floor directional light is lit.

   $\forall$ **(i,m)** $\in$ **IM,** $\mathbf{el\_dest\_depr_{i,m}} \Rightarrow \mathbf{el\_dest\_lit_{i,m}}$ **U (** $\exists$ **j** $\in$ **J,** $\mathbf{fl\_direction_{i,m,j}}$**)**

   $\forall$ **(i,m)** $\in$ **IM, (**$\neg$ $\mathbf{el\_dest\_depr_{i,m}} \wedge$ **(** $\exists$ **j** $\in$ **J,** $\mathbf{fl\_direction_{i,m,j}}$**)** $\Rightarrow \neg$ $\mathbf{el\_dest\_lit_{i,m}}$

6. If a destination light is backlit, and if it is in the direction indicated by the directional light, the elevator will proceed to this selected floor without turning around.

   $\forall\ (i,m1) \in IM,\ (\exists\ m \oplus^{j} m1,\ el\_dest\_lit_{i,m} \wedge el\_direction_{i,j} \wedge el\_position_{i,m1}) \Rightarrow el\_direction_{i,j}\ U\ el\_position_{i,m}$

7. As the elevator moves, and passes floors, the on-board directional indicators are updated. That is, if the elevator is at floor **m**, and going up, it must remain at floor **m**, until either it reaches floor **m+1**, or its directional light is extinguished.

   $\forall\ (i,m) \in I{*}M^{-},\ el\_position_{i,m} \wedge el\_direction_{i,up} \Rightarrow (el\_position_{i,m}\ U\ ((el\_position_{i,m+1} \wedge el\_direction_{i,up}) \vee (\neg\ el\_direction_{i,up} \wedge el\_position_{i,m})))$

   $\forall\ (i,m) \in I{*}M_{-},\ el\_position_{i,m} \wedge el\_direction_{i,down} \Rightarrow (el\_position_{i,m}\ U\ ((el\_position_{i,m-1} \wedge el\_direction_{i,down}) \vee (\neg\ el\_direction_{i,down} \wedge el\_position_{i,m})))$

8. The user also wishes that the service be equitable and efficient.

   a. Equitable service dictates that each request is given roughly equal priority, and that service to one user should not be much worse than service to other users.

   b. Efficient service has two components. The first component is related to the service provided; that the elevator system should respond as quickly as possible to each request (within the equitable constraints). The second component has to do with the cost of service (this is of more concern to the systems engineers than the users, but is put here for completeness).

5. System Engineer's Viewpoint

The user's viewpoint holds under all conditions, but there are many refinements to be added to make the system work effectively. For example, having every elevator going from bottom to top and back continuously, stopping at every floor, would satisfy the user conditions described in the previous chapter, with the exception of equitable and efficient service. One could even provide equitable service, by maintaining constant "headways" between the elevators as they move up and down. This, however, would be an inefficient way to operate the elevator subsystem. The systems engineer has to add formulas to further refine the operation of the system. This chapter considers some of these refinements. The introduction of some logical functions makes the description of operation much simpler. I then tackle the problem of refining the user's specifications further to clear up some ambiguities.

5.1. Useful Compositions into Functions

There are many conditions which keep arising in the specification; rather than describing such conditions repeatedly, they have been gathered into some definitions here. Their usefulness will become apparent later. The functions are enumerated below as compositions of the atomic propositions, and the interactions between the functions are described later.

1. The elevator is referred to as being dormant if certain conditions hold, indicating that the elevator has nothing to do just now. An elevator **i**, dormant at floor **m** is represented by the variable $\mathbf{dormant}_{i,m}$. The dormant conditions indicate that the elevator is at rest at a floor, with its doors closed, no destination push buttons backlit, and no summons buttons at that floor backlit.

   $\forall$ **(i,m)** $\in$ **IM,** $\mathbf{dormant}_{i,m} =$
   $\mathbf{el\_doors\_closed}_{i} \wedge \mathbf{el\_position}_{i,m} \wedge (\forall\ \mathbf{j} \in \mathbf{J}, \neg\ \mathbf{el\_direction}_{i,j} \wedge \neg\ \mathbf{fl\_direction}_{i,m,j})$

2. A **strong_continue** for an elevator in a given direction exists if there is an on-board destination push button backlit for a floor in the direction being considered. It is designated as a **strong_continue**, since, if the elevator already has the directional indicator set in direction **j**, it must continue to move in that direction.

   $\forall$ **(i,m1,j)** $\in$ **IMJ,** $\mathbf{strong\_continue}_{i,m1,j} = (\exists\ \mathbf{m} \oplus^{\mathbf{j}} \mathbf{m1}, \mathbf{el\_dest\_lit}_{i,m})$

   $\forall$ **i** $\in$ **I,** $[]\ (\neg\ \mathbf{strong\_continue}_{i,1,down} \wedge \neg\ \mathbf{strong\_continue}_{i,|M|,up})$

3. A **weak_continue** condition in a given direction indicates that there is a summons button backlit at a floor in the direction being considered. It is called a weak continue, since the scheduler can override this condition to provide more efficient and equitable service.

   $\forall$ **(m1,j1)** $\in$ **MJ,** $\mathbf{weak\_continue}_{m1,j1} = (\exists\ \mathbf{m} \oplus^{\mathbf{j1}} \mathbf{m1}, \mathbf{fl\_summ\_lit}_{m,j})$

   $[]\ (\neg\ \mathbf{weak\_continue}_{1,down} \wedge \neg\ \mathbf{weak\_continue}_{|M|,up})$

4. A continue condition in a given direction indicates that there is either a strong_continue condition, or a weak_continue without a sch_start_override.

$$\forall\ \mathbf{(i,m1,j1)} \in \mathbf{IMJ,\ continue_{i,m,j}} = \mathbf{strong\_continue_{i,m,j}} \vee \mathbf{(weak\_continue_{i,m,j}} \wedge \neg\ \mathbf{sch\_start\_override_{i,m,j})}$$

Note that, by definition of both strong and weak continues, the condition below is true.

$$\forall\ \mathbf{(i)} \in \mathbf{I,}\ [\,]\ \mathbf{(}\neg\ \mathbf{continue_{i,1,down}} \wedge \neg\ \mathbf{continue_{i,|M|,up})}$$

5. The function **stop_at**$_{\mathbf{i,m,j}}$ indicates that the elevator **i** has reason to stop at floor **m**, and indicates that it will travel in direction **j**. Note that at the same floor, the functions in both directions can each be true simultaneously.

$$\forall\ \mathbf{(i,m,j)} \in \mathbf{IMJ,\ stop\_at_{i,m,j}} = \mathbf{(el\_position_{i,m}} \wedge \mathbf{(el\_dest\_lit_{i,m}} \vee \mathbf{(fl\_summ\_lit_{m,j}} \wedge \neg\ \mathbf{sch\_stop\_override_{i,m,j})} \vee \mathbf{sch\_stop_{i,m,j})}$$

6. In addition, we define that the elevator must stop at the top and bottom floors.

$$\forall\ \mathbf{i} \in \mathbf{I,}\ [\,]\ \mathbf{(stop\_at_{i,1,down}} \wedge \mathbf{stop\_at_{i,M,up})}$$

5.2. Becoming Dormant

When the doors have just closed, if the elevator has no need to continue in the indicated direction, then the on-board directional indicator will become extinguished. Since the doors are closed, and the floor and elevator directional lights are extinguished, this places the elevator in a *dormant* condition.

$$\forall\ \mathbf{(i,m,j)} \in \mathbf{IMJ,\ (\bullet}\ \neg\ \mathbf{el\_doors\_closed_i} \wedge \mathbf{el\_doors\_closed_i} \wedge \mathbf{el\_position_{i,m}} \wedge \mathbf{el\_direction_{i,j}} \wedge \neg\ \mathbf{continue_{i,m,j})} \Rightarrow\ ^{\circ}\ \neg\ \mathbf{el\_direction_{i,j}}$$

5.3. Dormant Conditions

When the elevator is dormant, there are some conditions which will cause it to become non-dormant, and these are described below. These are expressed nondeterministically, and it is assumed that a fair choice will be made, as is described in the section labeled fairness.

1. When the elevator **i** is dormant at a floor **m**, and a user pushes the on-board destination button for the floor **m**, the light should only be on while the button is depressed, and the doors should simply re-open with no directional lights set. The door's reopening does not conflict with any of the directional signals.

$$\forall\ \mathbf{(i,m)} \in \mathbf{IM,\ (dormant_{i,m}} \wedge \mathbf{el\_dest\_lit_{i,m})} \Rightarrow\ ^{\circ}\ \neg\ \mathbf{el\_doors\_closed_i}$$

2. If the elevator is dormant at a floor, and a summons button at that floor is lit, the doors will reopen, with the floor indicator reset to the new direction. If only one summons button is lit, then the directional light comes on in that direction. If the up and down buttons are pushed simultaneously, we specify a nondeterministic choice as to which directional indicator will be lit.

$\forall$ **(i,m,j)** $\in$ **IMJ, (dormant**$_{i,m} \wedge$ **fl_summ_lit**$_{m,j} \wedge \neg$ **fl_summ_lit**$_{m,\neg j}$**)** $\Rightarrow$
$\circ$ **(fl_direction**$_{i,m,j} \wedge \neg$ **el_doors_closed)**

$\forall$ **(i,m)** $\in$ **IM, (dormant**$_{i,m} \wedge$ **fl_summ_lit**$_{m,down} \wedge$ **fl_summ_lit**$_{m,up}$**)** $\Rightarrow$
$\circ$**((fl_direction**$_{i,m,down} \vee$**fl_direction**$_{i,m,up}$**)**$\wedge\neg$**el_doors_closed)**

3. When the elevator is dormant at a floor, and a condition to cause it to go to another floor has arisen, it will switch on the on-board directional indicator in the appropriate direction. The up direction is given preference in this case.

   $\forall$ **(i,m)** $\in$ **IM, dormant**$_{i,m} \wedge$ **continue**$_{i,m,up} \wedge \neg \circ$ **fl_direction**$_{i,m,down} \Rightarrow \circ$ **el_direction**$_{i,up}$

   $\forall$ **(i,m)** $\in$ **IM, dormant**$_{i,m} \wedge$ **continue**$_{i,m,down} \wedge \neg$ **continue**$_{i,m,up} \wedge \neg \circ$ **fl_direction**$_{i,m,up} \Rightarrow \circ$ **el_direction**$_{i,down}$

The scheduler could always intervene to cause the elevators to react in the manner it desired. For example, with all elevators dormant at the same floor, we would expect the scheduler to prohibit all but one from responding in each direction.

5.4. Floor Indicators

The user's view described most of the interactions in terms of floor directional indicators. It did not, however, specifically address how these were set. The resetting was defined, by stating that when the doors are open, the elevator remains at the floor until they close, and all the floor indicators are reset. Since we use the strong **until**, this indicates that such a condition must eventually occur. The setting of the indicators is more complicated, as is detailed below.

1. For the in-between floors, the indicator is set to the direction of motion if the following conditions occur. These conditions are represented by the function **stop_at**$_{i,m,j}$.
   a. If the elevator stops at a floor due to a destination push button being lit for the floor, and the continue condition is set.
   b. If the elevator is to stop for a lit summons button in the direction of motion, with no scheduler override to prevent stopping.
   c. A guard on this activity is that the floor summons light in the opposite direction has not yet been set. This is needed to prevent the summons in the opposite direction from being set before the conditions to cause a continue in the same direction occur.

   $\forall$ **(i,m,j)** $\in$ **IMJ (el_direction**$_{i,j} \wedge$ **stop_at**$_{i,m,j} \wedge$ **continue**$_{i,m,j} \wedge \neg$ **fl_direction**$_{i,m,\neg j}$**)** $\Rightarrow$ **fl_direction**$_{i,m,j}$

   Note that this will not cause conflicts with the floor directional indicator in the direction of motion to be set at the top and bottom floors, since $\mathbf{continue}_{\mathbf{i,1,down}}$ and $\mathbf{continue}_{\mathbf{i,M,up}}$ are defined as always false. Hence, this will not violate the safety conditions previously given in section 2.4.2.

2. For the in-between floors, the indicators are set opposite to the direction of motion if the following conditions occur. Note that this assignment includes a *next*(°) operator to allow time for the floor directional light in the current direction to be determined unambiguously. The on-board directional lights will also change.

    a. There is no need to continue in the current direction.

    b. The floor directional signal in the current direction is not set.

    c. The stop_at function in the opposite direction is true.

    $$\forall\ \mathbf{(i,m,j)} \in \mathbf{IMJ}\ (\mathbf{el\_direction}_{\mathbf{i,j}} \wedge \neg\ \mathbf{continue}_{\mathbf{i,m,j}} \wedge \neg\ \mathbf{fl\_direction}_{\mathbf{i,m,j}} \wedge \mathbf{stop\_at}_{\mathbf{i,m,\neg j}}) \wedge \mathbf{continue}_{\mathbf{i,m,\neg j}} \Rightarrow {}^{\circ}\ \mathbf{fl\_direction}_{\mathbf{i,m,\neg j}}$$

    Please note that this condition always sets the floor directional light correctly at the top and bottom floors.

Note that in all of the above formula, when $\mathbf{fl\_direction}_{\mathbf{i,m,j}}$ is *true*, then by the previous formula in the user's specification, $\mathbf{el\_direction}_{\mathbf{i,j}}$ must also be *true*, and from the safety conditions formula then both $\mathbf{fl\_direction}_{\mathbf{i,m,\neg j}}$ and $\mathbf{el\_direction}_{\mathbf{i,\neg j}}$ are *false*. This assumes that there are no inconsistencies between these invoked formulas and any other formula.

5.5. Other Stopping Conditions

The systems engineer has now specified almost everything in his domain, however, a close inspection of all of the above formulas indicate that once an elevator is moving in a direction, it will continue in that direction until it reaches the last floor in that direction unless a specific condition exists at an in-between floor. This could be left to the scheduler to clear up, but since it is not difficult, it may as well be included as part of the controller. We can specify that if the elevator is at a floor, and there is no reason to stop and set a floor directional light, nor any reason to continue, then it should switch off the on-board directional light, thus effectively stopping the elevator at that floor in a dormant condition, without causing the doors to open.

$$\forall\ \mathbf{(i,m,j)} \in \mathbf{IMJ}\ (\mathbf{el\_position}_{\mathbf{i,m}} \wedge \mathbf{el\_direction}_{\mathbf{i,j}} \wedge \neg\ \mathbf{continue}_{\mathbf{i,m,j}} \wedge \neg\ \mathbf{fl\_direction}_{\mathbf{i,m,j}} \Rightarrow {}^{\circ}\ \neg\ \mathbf{el\_direction}_{\mathbf{i,j}}$$

This formula will also cause the elevator to become dormant if it reaches the top or bottom floor with no on-board buttons lit, and no summons button lit, due to the definition of the continue functions being false at the extremities.

6. Fairness Conditions

The fairness conditions represent restrictions to the manner in which nondeterministic conditions are handled. The fairness conditions assert that in such cases, the same path will not always be taken. In this specification, there is only one such applicable condition, namely when the system is dormant, and the up and down summons push buttons are backlit simultaneously, the system should not always select one of the directions. The condition of interest is repeated below from section 4.3, paragraph 2.

$\forall$ **(i,m)** $\in$ **IM, (dormant**$_{i,m}$ $\wedge$ **fl_summ_lit**$_{m,down}$ $\wedge$ **fl_summ_lit**$_{m,up}$**)** $\Rightarrow$ $\circ$**(fl_direction**$_{i,m,down}$ $\vee$**fl_direction**$_{i,m,up}$**)**

As is obvious from the above, the choice of direction is arbitrary and nondeterministic. The fairness condition must ensure that each time this condition arises, the same direction is not always chosen.

$\forall$ **(i,m)** $\in$ **IM,** [] $\Diamond$ **(dormant**$_{i,m}$ $\wedge$ **fl_summ_lit**$_{m,down}$ $\wedge$ **fl_summ_lit**$_{m,up}$**)** $\rightarrow$ $\circ$ **fl_direction**$_{i,m,down}$

$\forall$ **(i,m)** $\in$ **IM,** [] $\Diamond$ **(dormant**$_{i,m}$ $\wedge$ **fl_summ_lit**$_{m,down}$ $\wedge$ **fl_summ_lit**$_{m,up}$**)** $\rightarrow$ $\circ$ **fl_direction**$_{i,m,up}$

If these fairness conditions are not imposed, for example by always preferring the up direction, then the single elevator can be prevented from turning on the floor directional light down, by pushing the up summons button as the doors close, causing the up directional light to come on, and the doors to re-open. Obviously this can continue forever.

7. Verification

One of the benefits of using Temporal Logic to specify the operation of a system is that the system can be verified to be consistent, and some conclusions can be reached about completeness and ambiguity.

7.1. Example 1

This example shows how the logic can be interpreted as a state machine, and illustrates some of the strengths and weaknesses of this approach. The reader unfamiliar with the state machine representation used may refer to Appendix 1. The example is for the trivial case of a single floor destination push button and its associated directional indicator light in a system with a single elevator, ignoring all of the other considerations. The English statement of the problem is given below.

When the floor summons push button is depressed, it is also backlit, and will remain backlit until the associated floor directional light is lit.

The definition of the three atomic propositions (variables) are:

1. **fl_summ_depr**: when this is true, the push button is depressed, and when it is false, the push button is released.
2. **fl_summ_lit**: when this is true, the push button is backlit, and when it is false, the light is unlit.
3. **fl_direction**: when this is true, the floor directional light is lit, and when it is false, it is unlit.

Two formulas which jointly describe the behavior of the push button are listed below.

1. **fl_summ__depr $\Rightarrow$ fl_summ_lit U fl_direction**
2. **($\neg$ fl_summ_depr $\wedge$ fl_direction) $\Rightarrow$ $\neg$ fl_summ_lit**

The system can also be represented as a state machine, with eight states, as can be seen from Table 1. The relationship R is defined below in Table 2.

state number	1	2	3	4	5	6	7	8
fl_summ_depr	F	T	F	T	F	T	F	T
fl_summ_lit	F	T	T	T	F	F	T	F
fl_direction	F	F	F	T	T	T	T	F
Formula 1	t	t	t	t	t	f	t	f
Formula 2	t	t	t	t	t	t	f	t

Table 1

state	1	2	3	4	5
1		x	x	x	x
2			x	x	x
3		x		x	x
4	x				x
5	x			x	

Table 2 (R)

state	1	2	3	4	5
1		x		x	x
2			x	x	x
3		x		x	x
4	x				x
5	x			x	

Table 3 (R^*)

path number	1	2	3	4	5	6	7
path	1 2 3 5	1 2 4	1 2 5	1 4	1 5	1 2 3 4	1 3 5
fl_summ_depr	F T F F	F T T	F T F	F T	F F	F T F T	F F F
fl_summ_lit	F T T F	F T T	F T F	F T	F F	F T T T	F T F
fl_direction	F F F T	F F T	F F T	F T	F T	F F F T	F F T

Table 4.

In Table 1 the line labeled Formula 1 shows the truth value of the first of the two formulas specifying the system, and the line labeled Formula 2 shows the truth value of the second specification formula. As can clearly be seen, one or the other formula is invalid for three states (6,7,8) in the truth table. States 6 and 8 are invalid since the destination button is depressed, and the summons button is unlit, clearly violating the requirements of Formula 1. State 7 is invalid since it allows the summons light to be on when the directional light is on and the push button is not depressed, hence violating Formula 2. Thus the two formulas are only true over this restricted set of states S = (1,2,3,4,5) and are thus *consistent* with each other with respect to this set of states. Then Table 1 defines the mapping function P of each state in S on to the values of the atomic propositions in that state. The relationship R between the states is shown in Table 2. Thus a complete state machine SM =(S,R,P) has been defined for the system.

A list of some (7) of the valid paths of states, starting at state 1, is shown in Table 4; the paths are only shown until they satisfy Formula 1. As can be seen from the table, in each path whenever the **fl_summ_depr** condition is true, then so is the **fl_summ_lit**, and it remains true until the **fl_direction** becomes true. This means that it satisfies Formula 1, and the path is valid. When **fl_summ_depr** is false, then Formula 1 does not apply.

However, path 7, which satisfies the formula, is clearly absurd in context, since the light represented by **fl_summ_lit** comes on without any push button depression. In this case, we must accept that the specification is incomplete, since it produces an undesirable behavior. We must now add the further formula shown below to disallow this condition.

fl_summ_lit ⇒ •(fl_summ_lit ∨ fl_summ_depr)

This states that if the light is lit, then in the previous state, either the light was lit or the push button was depressed. This formula is clearly invalid against SM. We must now change the relationship R to R^* shown as Table 3, which invalidates the relation between states 1 and 3, making path 7 in R impossible in R^*. This now defines a new state machine SM^*=(S, R^*, P), against which all three formulas are valid.

7.2. Example 2

This example verifies that the elevator motion between two intermediate floors is consistent. This section considers only one elevator, and drops the quantifier **i** from the original equations.

The relevant atomic propositions are :

1. $\mathbf{el\_position_m}$
2. $\mathbf{el\_direction_j}$

The formulas to be considered are listed below (note that these differ somewhat from the ones in the body of the document).

1. $\forall\ \mathbf{m} \in \mathbf{M}\ []\ \mathbf{el\_position_m} \wedge \mathbf{el\_direction_{up}} \Rightarrow \mathbf{(el\_position_m\ U\ (el\_position_{m+1}} \vee \mathbf{(}\neg\ \mathbf{el\_direction_{up}} \wedge \mathbf{el\_position_m)))}$
2. $\forall\ \mathbf{(m)} \in \mathbf{M}\ []\ \mathbf{el\_position_m} \wedge \mathbf{el\_direction_{down}} \Rightarrow \mathbf{(el\_position_m\ U\ (el\_position_{m-1}} \vee \mathbf{(}\neg\ \mathbf{el\_direction_{down}} \wedge \mathbf{el\_position_m))}$
3. $\forall\ \mathbf{m1} \in \mathbf{M}\ []\ \mathbf{el\_position_{m1}} \Rightarrow \mathbf{(}\ \forall\ \mathbf{m} \in \mathbf{(M\text{-}m1)}\ \neg\ \mathbf{el\_position_m)}$
4. $\forall\ \mathbf{j} \in \mathbf{J}\ []\ \mathbf{el\_direction_j} \Rightarrow \neg\ \mathbf{el\_direction_{\neg j}}$

The 12 relevant states are given below in Table 2.1.

states	1	2	3	4	5	6	7	8	9	10	11	12
$el\_position_{m-1}$	T	F	F	T	F	F	T	F	F	F	F	F
$el\_position_{m+0}$	F	T	F	F	T	F	F	T	F	F	F	F
$el\_position_{m+1}$	F	F	T	F	F	T	F	F	T	F	F	F
$el\_direction_{up}$	T	T	T	F	F	F	F	F	F	T	F	F
$el\_direction_{down}$	F	F	F	F	F	F	T	T	T	F	T	F

Table 2.1

It would be tedious to list all of the unacceptable states; suffice it to note there are 20 states that are disallowed by the safety conditions, that only one of the three values **el_position** and only one of the two values **el_direction** can be true at any time.

The relationship R, between the 12 states, is shown below in tabular form.

	1	2	3	4	5	6	7	8	9	10	11	12
1		x		x	x		x	x				
2			x		x	x		x	x	x		
3						x			x	x		
4	x						x					
5		x						x				
6			x						x			
7				x							x	
8					x		x					
9						x		x				
10	x										x	x
11									x	x		x
12										x	x	

Table 2.2

It can be easily checked that the valid paths from the above defined state machine, starting from state 2, are listed below. Note that a single state change is only necessary in each case to provide the desired behavior.

path	1		2		3		4	
states	2	3	2	5	2	6	2	8
$el\_position_{m-1}$	F	F	F	F	F	F	F	F
$el\_position_{m}$	T	F	T	T	T	F	T	T
$el\_position_{m+1}$	F	T	F	F	F	T	F	F
$el\_direction_{up}$	T	T	T	F	T	F	T	F
$el\_direction_{down}$	F	F	F	F	F	F	F	T

Table 2.3

Unfortunately, inspection of the paths show that path 3 is undesirable, since it allows the elevator to change floors, while allowing the directional light to go off immediately. If we wish to maintain the directional indicator across floor changes, then we must state this explicitly. This can be done by extending Formula 1 as indicated below. The formula below is not consistent with the old relationship defined in table 2.1, and we need to redo this relationship by dropping the transition for (2,6) and the similar conditions for the other floors and the other direction. Only the formula for the elevator to move up is written.

$$\forall\ m \in M\ []\ el\_position_m \wedge el\_direction_{up} \Rightarrow (el\_position_m\ U$$
$$(el\_position_{m+1} \wedge el\_direction_{up}) \vee (\neg\ el\_direction_{up} \wedge el\_position_m))$$

8. The SML Model

The elevator model was built using the State Machine Language (SML) developed by Ed Clarke and associates, and described in [Clarke 87] and other publications. The details of the model programs are given in Appendix 2.

The model consists of three major parts.

1. A first floor, which turns the elevator around, and causes it to wait for a request at the second or third floor.
2. A second floor, which is modeled completely, allowing the elevator to bypass it in either direction if there is no reason to stop, or to stop in response to a summons button or destination button.
3. A third floor, which turns the elevator around, and causes it to wait for a request at the second or first floor.

The SML model of the elevator is a subset of the temporal logic descriptions due to the limitations of the tools. Various models were built during the model development stage to respond to these difficulties:

- Modeling a system which was small enough for the compiler.
- Learning the SML language, and how best to use this language to express the problem.
- Attempting to achieve as large a system as possible.

8.1. Model Overview

The model could have been done in two different ways:

1. The control of the elevator moving from floor to floor could have been implemented procedurally by a main program; this is referred to as the *centralized* approach.
2. The control of the elevator moving from floor to floor could be distributed over the system, with each module forever in a loop awaiting its activation condition to occur; this is referred to as the *distributed* approach.

I selected the second approach, since it seems more appropriate for distributed concurrent systems. I conducted experiments once or twice during the development of the model to check whether the distributed control philosophy was causing larger state machines to be generated than would have been the case with centralized control. This was done at the early stages, on a small (16) state machine, and in this case the underlying state machines were identical. At a later stage of development, I took a model using the distributed approach and converted it to a centralized control program. Once again the same size state machine was created (238 states) in each case, and the same set of tests were passed by each approach. I took this as evidence that both representations were equivalent, and stuck with the more appealing (to me) distributed approach.

The layout of the modules is shown in Figure 8-1. Each of the six modules is shown on the graph, and the control flow between modules is represented by the lines connecting the modules. A short description of the control flow is given below. Before reading this description, it should be understood that each module is in a forever loop, waiting for specific conditions to arise to cause the module to start executing the body of the loop. When a module satisfies its outer loop condition, and proceeds to execute the body, it is described in the text below as being active. While awaiting the initiating condition, it is described as being inactive.

1. When the elevator is below the middle floor, then **at_below** is active, while all other modules are inactive. When an appropriate push button is depressed, it will change the variables, and cause the outside wait condition in routine at_floor_up to be satisfied, and become active. At the same time, the at_below module becomes inactive.
2. When the **at_floor_up** module becomes active, it checks whether there are any conditions requiring the elevator to stop. If there are it sets a floor directional light on, thus causing the stopping_at_floor module to become active, otherwise it changes the elevator position to above, thus causing the at_above module to become active. Under both conditions, the at_floor_up module becomes inactive.
3. When the **stopping_at_floor** module becomes active, it goes through the stopping procedure. When the doors are closed, if there is no reason to move up or down, the elevator becomes dormant, and the at_dormant module is activated. In this case, the stopping_at_floor module remains active in case the condition arises that the elevator is to leave the floor without re-opening its doors, due to a push button request to visit another floor.
4. When the **at_dormant** module becomes active, it awaits a push button request, and sets the appropriate variables depending on the button depressed. In all cases, it expects the stopping_at_floor module to take care of the further actions required (such as leaving the floor), and itself becomes inactive.
5. The **at_above** and **at_floor_dn** modules are similar to the at_below and at_floor_up modules.
6. There is an additional module for the buttons and lights that interacts with all other modules; to reduce clutter, it is not shown on the figure.

Figure 8-1: Elevator Control Modules

8.2. Model Characteristics

Before describing the model, some of the limitations, assumptions, and difficulties should be enumerated.

1. There are problems of scale, and I was continually trying to change the model design to allow for a larger scale system to be evaluated.
2. The model is not derived automatically from the temporal logic statements, but is programmed separately. This gives the modeler freedom of choice in developing the model. The model described in the report is reasonably close to the temporal logic descriptions (in some places at least), but some earlier ones were not so close. In a later section, two different versions modeling the same temporal formulas will be shown and discussed.
3. I eventually decided to use parallelism to as great an extent as was reasonable (and could be handled within the scale available in the modeling tool).
4. The model is somewhat constructive. That is, some groups of temporal logic statements can be identified as processes in the model.

8.3. Variants of the At_dormant Model

There were two different models built for the at_dormant module. Both models were included in the configuration at different stages of the specification development, and passed various tests during the stages where they were used. In the final stages of the code development, the model described in Appendix 2.f was used, and the variant shown in Appendix 2.h was used at earlier stages. The two models are described below, then compared in a later subsection.

8.3.1. Directly Derived Model

The first of the models is a direct mapping from the temporal logic expressions. It specifies each temporal logic condition as a branch in a parallel statement, and waits for one or more of the conditions to arise. Guards against simultaneous occurrences are included. The extra conditions are inserted to explicitly represent the inherent nondeterminism when the up and down floor summons requests are registered simultaneously. This is done by simply introducing a switch, which allocates the direction differently on each invocation.

8.3.2. Action Oriented

The second model is somewhat different, and is based on the fact that one of four conditions arise to conclude the elevators dormancy:

1. It opens its doors with the directional indicators set **up**.
2. It opens its doors with the directional indicators set **down**.
3. It sets the elevator **up** directional indicator, and sets out for the floor above.
4. It sets the elevator **down** directional indicator, and sets out for the floor below.

The model is built by waiting for any of the initiating events to occur, then choosing which action is appropriate. This, of course, does not bear a direct correlation to the temporal logic formulas. The temporal formulas could be re-expressed to read almost exactly like this model, but there was no reason to do this.

8.3.3. Comparisons

The advantages of the second model are that it uses less parallelism. It is probably closer to an implementation and it produces a significantly smaller state machine (roughly 500 states as opposed to roughly 1000 states for the first model). The disadvantage of the second model is the obvious one, that its derivation from the temporal logic formulas is not obvious or straightforward. On the other hand, the first model is easily derived (and reviewed) from the temporal logic formulas, but seems awkward for developing an implementation. I chose to use the first model in most of the work, because it is straightforward and easy to interpret.

8.4. Model Validation

Obviously the model cannot validate the complete generality of the temporal logic statements, since it could not handle multiple elevators and floors. It did, however, model the complete operation of a single floor, and its interfaces to the adjacent floors. In validating the model, I had to transform the temporal logic statements in the document somewhat before proceeding with the validation. There were some changes in syntax (for example the operators are all expressed as special text character sets rather than symbols), but these are relatively unimportant and are ignored here. The major transformations are detailed below.

1. The temporal logic expressions in the report are given with no concept of time. In the modeling toolset, time is represented as a fixed interval between ticks of a clock. More to the point, any change in state occurring during one interval cannot cause another change in state until (at least) the following interval. This means that some of the formulas expressed in the body of the report required an additional **next** operator. For example, the formula describing a push button depression was transformed as seen below.

   $\forall$ **(m,j)** $\in$ **MJ, fl_summ_depr**$_{m,j}$ $\Rightarrow$ $^{\circ}$**(fl_summ_lit**$_{m,j}$ **U (** $\exists$ **i** $\in$ **I, fl_direction**$_{i,m,j}$**))**
2. The toolset does not include the **previous** temporal operator. This means that we must re-express those formula using this operator, and requires us to use **next** operations. One example is shown below. for the "becoming dormant" condition.

   $\forall$ **(i,m,j)** $\in$ **IMJ, (**$\neg$ **el_doors_closed**$_i$ $\wedge$ $^{\circ}$ **(el_doors_closed**$_i$ $\wedge$ **el_position**$_{i,m}$ $\wedge$ **el_direction**$_{i,j}$ $\wedge$ $\neg$ **continue**$_{i,m,j}$**))** $\Rightarrow$ $^{\circ\circ}$ $\neg$ **el_direction**$_{i,j}$
3. The toolset includes the complete branching time temporal logic capabilities (Appendix 1), whereas the logic in this paper did not require the existential search through paths.
4. The toolset set is based on propositional calculus, not predicate calculus, and hence contains no quantifiers over a number of similar objects. The temporal expressions in the report merely use quantifiers as a notational convenience, (as opposed to using them to count such items as number of occurrences), and can easily, if inconveniently, be replaced by a number of individual formulas. Since it was only possible to model a very small system, this was not a problem. If, however, a larger system were to be modeled, it would be a problem.

9. Discussion

The discussion of the findings is organized around some previous work in the classification and evaluation of software development methods described in [Firth 87] and [Wood 88]. This considers a method to consist of representations of objects, ways of deriving these representations, and the means of examining the objects for desirable characteristics. In addition to those characteristics which can be clearly demonstrated in the elevator problem, I also include some others that are important in most specification domains, but are not present in the chosen elevator problem. However, I will begin with a general overview.

The temporal logic specification was produced by a specifier (the author), and reviewed by a single reviewer. This report, of course, has had other reviewers. But in this chapter, all references to the reviewer are to the one individual who reviewed the specification through many cycles over a period of months.

The temporal logic specification proved to be a good communication vehicle between the specifier and the reviewer, even though, in this case, both people were somewhat unfamiliar with temporal logic, and were struggling with the underlying mathematics as well as the elevator specification. The specification was written and went through about three review and rewrite cycles. The first review changed the organization of the report as well as some of the formulas. The remaining reviews concentrated on the formulas and their consistency. Since we were still in the throes of understanding the temporal logic, this, of course, cleared up some of the foggy understanding of the logic, and reinforced my belief that it is easier to learn a technique by applying it to a small problem. By the last review, the reviewer was able to point out some rather esoteric problems with the specification, only to find that the specifier could justify the existing formulas by cross-references to other portions of the specification, and convince the reviewer of the consistency of the formulas as they existed. This convinced both the specifier and the reviewer that the temporal logic was extremely appropriate for this problem, and improved our understanding of the problem. More importantly, it assisted our ability to communicate. Not only could we argue over details of the specification, but we were able to resolve these areas of disagreement precisely. I should also add that the reviewer always felt that there was insufficient explanatory text, while the specifier was uninterested in expanding the English descriptions.

9.1. Representations

The representations are the objects which specify the system, and the discussion below is targeted at specific characteristics of systems that the representations should have.

1. The elevator specification is a behavioral specification. The only portion of the specification which needs to be functional is the scheduler, and it was not considered. Since temporal logic is a powerful method for describing behavior, and was chosen as the specification technique for that reason, it comes as no surprise that it is a good specification technique for the problem. The style of the specification, is declarative, rather than operational. The SML model is an operational model of the system's behavior.

2. The temporal logic as used here exhibits no particular notion of concurrency, communications, processes, or modularity. Each formula is expressed in a stand-alone manner, and each will operate concurrently with the other formulas. Hence the structure is rather flat, and assumes that communication occurs by the global sharing of data.
3. The ability to express the problem nondeterministically was useful in only a few places, and could easily have been avoided altogether, so this specification does not rely on temporal logic's ability to conveniently specify things nondeterministically. Since nondeterminism is little needed, then fairness does not play a major role in this specification. The fairness of the scheduler inputs is by far the most important of the fairness issues, but, due to the inability to scale up the model to more than one elevator, these fairness issues could not be properly considered.
4. There are no explicit representations of time required in the specification, so the inability of temporal logic to conveniently deal with such issues as hard real-time deadlines was not a problem.
5. There are no data abstraction capabilities in SML. As a result, many of the procedures were repeated with slight changes using the editor. This was preferable to passing many parameters to more general modules, which would have thoroughly confused the issues.

9.2. Derivation

In this section we will discuss only the derivation of the temporal logic expressions, since the building of the model should be treated under the heading of examination.

1. I not only lumped all of the safety formula into a single early chapter in the report, but actually expressed most of these conditions early in the derivation process, thus simplifying the expression of the liveness conditions. All of the safety conditions were not written down on a single pass before the liveness conditions were expressed, but were derived and changed iteratively. In general, however, the safety formula emerged early in the derivation process, and was changed very little in the later stages of correcting the liveness conditions.
2. The liveness conditions were not originally derived as reported in the paper. I initially organized the report to specify the operation by organizing the specification formulas around the different device types (for example, by writing all formulas associated with the elevator directional lights). This organization turned out to be cumbersome to review and keep consistent. I then decided to take the approach of specifying only what the end-user touched and saw. This caused an additional difficulty of removing some atomic formulas which had been conveniently introduced (for example, a logical variable indicating if the elevator was bypassing the floor or stopping at the floor). When such conditions were removed (and in some cases replaced by equivalences for convenience) the specification solidified quickly. A significant early part of the problem was to remove my knowledge (or guess) of how the system should work.
3. This allowed me to then refine the user's specification and derive the system specification by resolving some open specification issues. At one point I considered these to be an overworking of the problem, which I had been tempted

into by my recent conversion to using temporal logic. My opinion during this time period was to simply model rather than waste time and paper including issues explicitly in the specification. However, building a model to satisfy these conditions consistently was a challenging endeavor, so in the end I was convinced that the further detail was not only desirable, but absolutely necessary for success.

4. I did not elaborate on the specification by introducing more detail, such as the manner in which a controller would interact with the elevator devices, actuators, and sensors to produce the desired behavior, and from this derive a software specification in temporal logic. This, of course, is important future work.

9.3. Examination

The examinations conducted were of three different types, namely:

- Review of the temporal logic for consistency and applicability.
- Handcrafting of small state machines to verify some small scale local consistency of the temporal logic.
- Building the SML model to check the consistency of the temporal logic.

The handcrafting of the state machine would have been unnecessary, had I started earlier to use the modeling languages. However, at least one important point arose from this effort, so it will be reported.

1. As has been stated too often already, the temporal logic specification was a splendid vehicle for communicating precisely the behavior of the elevator system to a reviewer, and to accommodate the changes that arose from the review. The success of the review process is further emphasized by the fact that the formulas were consistent with the model.
2. The model building in SML relied on the liveness conditions. The safety conditions were largely used for model validation. The SML model has been discussed in detail in a previous chapter.
3. The formulas were checked against the limited scale model, and were found to be consistent. This was a psychological "shot in the arm" for the specifier, and greatly increased my confidence not only in the elevator specification, but in the usefulness and applicability of this approach. Two other ramifications of consistency are listed below.

    a. The early handcrafting of a state machine revealed some underspecification, which would not have been found in the modeling of the system. Care has to be taken that such underspecification is not simply ignored, but that the specifier (and reviewers) seek out conditions that have not been specified.

    b. There has been no systematic attempt to look into the concept of overspecification, which can be defined as a specification formula whose truth or falsity can be derived directly from the other formulas in the specification. This could be a useful concept, if the test specification is derived from the temporal logic formula. If a formula were determined to be an overspecification of the system, then tests are not necessary to cover this formula, thus reducing the test burden.

Some other important issues, such as maintainability and usability were not specifically addressed in the case study. However I feel comfortable speculating that the specification, especially with modeling support, is eminently maintainable. Changes to the specification can be made, the model can then be changed to accommodate them, and checks for consistency can finally be made. This can all be accomplished without referring to the actual implementation.

10. Conclusion

The specification of the elevator problem, the building of models describing its behavior, and the checking of the consistency of the specification, all led me to believe in the usefulness of this approach, and convinced me not only that such systems can be specified, but also that they should be specified. Such specifications allow for consistent and thorough reviewing of the details of how the system is to operate, and the modeling allows for consistency checking in fine detail. The availability of a supporting toolset allowed me to gain confidence in my ability to apply the technique, and to find flaws in the logic and the model.

The specification was understandable to a knowledgeable reviewer, who approached the review with energy and enthusiasm. To successfully use this technique, it would be necessary to educate the people who would read such a specification to some minimal level. Most readers, such as quality control people deriving test specifications, would not have to understand the temporal logic to the extent of demonstrating completeness, or of deriving the formula, but would have to be sufficiently knowledgeable to develop test requirements from the specifications. The actual end-users could not be expected to understand such formulas, but their engineering support people, who develop the requirements, would have to understand the specification.

Appendix 1: Temporal Logic

Temporal logic is a modal logic and each temporal formula is interpreted as true or false against a model of the world. I chose to check the validity of a formula against a state machine in a manner similar to that of (Clarke 84). The state machine SM =(S,R,P) is defined over a set of atomic propositions AP (logical variables), and has the elements described below.

1. **S** is a finite set of states;
2. **R** is a binary relationship on S, defining the possible transitions between states; the arc $(s_i, s_j) \in R$ indicates that there is a direct path between the states s_i and s_j. Obviously if there is no such relationship in **R**, then there is no arc connecting the two states. Every state must be connected to at least one other state.
3. **P** assigns to each state the set of atomic propositions true in that state.

A path $\alpha(s_i)$ is an infinite sequence of states starting at state s_i; $\alpha(s_0) =_{df} (s_0, s_1, s_2, ...)$, such that each adjacent tuple of states (s_i, s_{i+1}) is in the relationship **R**. $\forall i \ni i \geq 0, (s_i, s_{i+1}) \in R$.

1.a. Branching Time Temporal Logic

The standard notation for stating that the formula f holds at state s_i in the structure SM is: **SM,** $s_i \models f$. When the structure SM is understood, $s_i \models f$ will suffice. The operators used within the body of the paper are from Branching Time Temporal Logic, and are enumerated below.

1. $s_i \models p$ iff $p \in P(s_i)$
2. $s_i \models \neg p$ iff $p \notin P(s_i)$
3. $s_i \models f1 \vee f2$ iff $(s_i \models f1) \vee (s_i \models f2)$
4. $s_i \models \circ f$ iff $\forall j, (s_i, s_j) \in R, s_j \models f$
5. $s_i \models \bullet f$iff $\forall j, (s_j, s_i) \in R, s_j \models f$
6. $s_0 \models \Diamond p$ iff $\forall$ paths $\alpha(s_0)$ $(\exists i \ni i \geq 0, (s_i \models p)$
7. $s_0 \models [\,]\, p$ iff $\forall$ paths $\alpha(s_0)$ $(\forall i \ni i \geq 0, (s_i \models p)$
8. $s_0 \models p\ U\ q$ iff $\forall$ paths $\alpha(s_0)$ $(\exists i \ni i \geq 0, (s_i \models q) \wedge (\forall j \ni 0 \leq j \leq i, (s_j \models p)))$

Appendix 2: Model details

The overview of the model was given in the body of the report, but details of the program modules are presented here. Each model is described separately as if it were a stand-alone module, since this makes the presentation more understandable. In actual fact all modules comprise a single source file, and are compiled as such. The program variable declarations and the main program are described first, followed by each of the other modules in turn. The program variables relate directly to the names used in the temporal logic formulas, according to the rules listed below.

- Since we are only modeling the middle floor completely, and only have one elevator, the floor and elevator subscripts are simply dropped.
- El_above and el_below are used to represent the position at the upper and lower floors (rather than el_position_3 and el_position_1.
- The floor doors were not included in the model.

Some comments here should make the programs more readable.

- Each assignment statement takes a single clock tick to execute. If many statements are to be executed during the same clock tick, they are enclosed in a compress_endcompress statement.
- Each of the statement groups within a *parallel* statement (groups are separated by the || symbols) are active simultaneously within the same clock tick. The *break* statement in one group causes all groups to exit from the parallel statement. The *exit* statement within a parallel grouping causes exit from the innermost loop, thus also exiting from the parallel statement.
- The *delay 3* statement causes a delay of 3 clock ticks.
- The *wait*(expression) causes the software to wait until the expression becomes true.
- The *raise* and *lower* statements cause the variables to become *true* and *false*, respectively.
- The symbol "!" denotes logicalnot $\neg$.
- The symbol "|" denotes logicalor $\vee$.
- The symbol "&" denotes logicaland $\wedge$.

2.a. Main Program

```
program fl1_el1_7e;

input  el_dest_depr;                           -- destination button is depressed
output el_dest_lit;                            -- destination button is lit

input  fl_summ_depr_up, fl_summ_depr_dn;   -- summons button is depressed
output fl_summ_lit_up,  fl_summ_lit_dn;    __ summons button is lit

input  fl_summ_depr_adn, fl_summ_depr_bup; --summons at above and below floors
output continue_up, continue_dn;           -- continues derived from buttons

output fl_direction_up, fl_direction_dn;   -- floor directional lights
output el_direction_up, el_direction_dn;   -- elevator directional lights

output el_position;                        -- elevator at middle floor
output el_above, el_below;                 -- elevator above, below

output el_doors_closed;                     -- doors closed
output dormant;                            -- dormant condition

internal non_determ;                       -- flag to mimic nondeterminism

procedure wait(exp)          -- waits for the expression to become true
while !(exp) do loop skip endloop
endproc;

compress                     -- set the initial conditions
   raise(el_below);          -- the elevator is below the middle floor
   lower(el_direction_up);   -- its not going up
   raise(el_direction_dn);   -- it is going down
   lower(fl_direction_up);   -- the middle floor direction light up is reset
   lower(fl_direction_dn);   -- the middle floor direction light down is reset
   lower(el_above);          -- the elevator is not above the middle floor
   raise(el_doors_closed);    -- the doors are closed
endcompress;
parallel
   button_lights;
||
   at_floor_up;
||
   at_floor_dn;
||
   at_above;
||
   at_below;
||
   stopping_at_floor;
||
   at_dormant;
endparallel;
endprog
```

Table II.1

2.b. Buttons and Lights

It is appropriate to start with these models, since they are straightforward, and will help introduce the terminology. Each button/light combination is treated as its own state machine, running in parallel with each other and with the rest of the system. The SML programs can be seen in Table II.2. Each is an instantiation of a single procedure butt_light, with the passed parameters giving the important details. The procedure button_lights is invoked by the main program. Five buttons and lights are modeled: the full set of three for the middle floor, and one each for the below and above floors. The buttons and lights at the above and below floors are there to emulate the continue functions, which are combinations of the requests of buttons and lights in a more extensive system.

The procedure butt_light is forever in a loop. Within the loop it awaits the depression of the push button. Then it puts the light on and waits for the condition where the floor directional light comes on with the push button not depressed, at which point it turns off the light. Note that depressing the button while it is lit has no effect.

```
procedure butt_light(depress, fl_direction, light)
-- when a push button is depressed, it backlights the button associated with it
-- then waits until the arrival condition to clear the light occurs.
-- Note that this guarantees the light to be on for at least one cycle
loop
   wait(depress);                    -- wait for button
   raise(light);                     -- put the light on
   wait( (fl_direction) & !depress); --wait till elevator stopping
   lower(light);                     -- put light off
endloop;
endproc;

procedure button_lights
parallel
   butt_light(el_dest_depr, fl_direction_up | fl_direction_dn, el_dest_lit);
||
   butt_light(fl_summ_depr_up, fl_direction_up, fl_summ_lit_up);
||
   butt_light(fl_summ_depr_dn, fl_direction_dn, fl_summ_lit_dn);
||
   butt_light(fl_summ_depr_adn, el_above, continue_up);
||
   butt_light(fl_summ_depr_bup, el_below, continue_dn);
endparallel;
endproc;
```

Table II.2

2.c. At_floor_up

This module waits for the elevator arriving at the middle floor from below, and can be seen in Table II.3. It models the specification in the *floor indicators* and *other stopping conditions* sections, and includes the stop_at function. It is somewhat less general than the temporal logic specification. If there is no reason to continue, the specification requires the elevator to stop at the floor by turning off the elevator directional light. Close inspection of the SML model shows that, in this case, the model can turn on the floor directional light in the opposite direction while it is stopping. This does not violate any of the TL specification.

```
procedure at_floor_up
loop
   wait(el_below);                        -- wait until the elevator is below
   wait(el_position & el_direction_up);   -- wait till its coming to the floor
   if (el_dest_lit | fl_summ_lit_up | !continue_up) -- stopping condition
   then                                             -- stop
      if(fl_summ_lit_up | continue_up)    -- indicate going up condition
      then  raise(fl_direction_up)        -- indicate going up
      else  compress                      -- change to indicate going down
              raise(fl_direction_dn);
              lower(el_direction_up);
              raise(el_direction_dn);
            endcompress;
      endif;
    else
      compress                                      -- pass the floor by
         raise(el_above);
         lower(el_position);
      endcompress;
    endif;
endloop;
endproc;
```

Table II.3

2.d. At-floor_down

This module waits on the elevator arriving at the middle floor from above, and can be seen in Table II.4. It models the specification in the *floor indicators* and *other stopping conditions* sections, and includes the stop_at function. It is somewhat less general than the temporal logic specification. If there is no reason to continue, the specification requires the elevator to stop at the floor, by turning off the elevator directional light. This is very similar to the at_floor_up procedure.

```
procedure at_floor_dn
loop
   wait(el_above);
   wait(el_position & el_direction_dn);
   if (el_dest_lit | fl_summ_lit_dn | !continue_dn)
   then
      if(fl_summ_lit_dn | continue_dn)
      then  raise(fl_direction_dn)
      else  compress
              raise(fl_direction_up);
              lower(el_direction_dn);
              raise(el_direction_up);
            endcompress;
      endif;
    else
      compress
         raise(el_below);
         lower(el_position);
      endcompress;
    endif;
endloop;
endproc;
```

Table II.4

2.e. Stopping_at _floor

This procedure models the actions of stopping the elevator at the floor, and also causes the elevator to leave the floor when the required conditions arise. This is shown in Table II.5.

1. The procedure waits until the doors are closed and one of the floor directional lights are on. The decision to turn one of these lights on or off is made in the at_floor_up or at_floor_down procedures.
2. The module then delays to model the elevator travel time to the floor, and opens the doors.
3. The module then starts a loop, with a parallel statement with three separate branches.

    a. The first parallel condition represents how the elevator reacts when the doors open. Entry into this condition can occur not only from above, but from actions within the procedure dormant, since one of the results of the dormancy condition is to open the doors and turn on a directional light. This branch delays, then closes the doors and turns off the floor directional indicators. If the conditions for the elevator to move are not present, the elevator directional indicators are turned off, and the dormancy condition is raised. This loop is exited only when the elevator leaves the floor because of one of two other branches.

    b. The second parallel branch represents waiting for the condition to continue up to arise. This condition can occur immediately after the doors are closed, or can occur while the elevator is dormant. When the up condition arises, the elevator leaves for the floor above.

    c. The third parallel branch represents waiting for the condition to continue down to arise. This condition can occur immediately after the doors are closed, or can occur while the elevator is dormant. When the down condition arises, the elevator leaves for the floor below.

```
procedure stopping_at_floor
loop
   wait( el_doors_closed & (fl_direction_up | fl_direction_dn ));
   delay 1;                   -- wait for arrival to occur
   lower(el_doors_closed);     -- open the doors
   loop
     parallel
        loop
          wait(!el_doors_closed & el_position);
          delay 1;
          compress
            raise(el_doors_closed);     -- close the doors
            lower(fl_direction_up);    -- clear the floor directional lights
            lower(fl_direction_dn);    -- clear the floor directional lights
          endcompress;
          if(!(continue_up & el_direction_up)&!(continue_dn & el_direction_dn))
          then
            compress                   -- make the elevator dormant
                lower(el_direction_up);
                lower(el_direction_dn);
                raise(dormant);
            endcompress;
          endif;
         endloop;
      ||  wait( el_position & continue_up & el_direction_up & el_doors_closed);
          compress                     -- elevator goes up
             raise (el_above);
             lower( el_position);
          endcompress;
          exit;
      ||  wait( el_position & continue_dn & el_direction_dn & el_doors_closed);
          compress
              raise (el_below);        -- elevator goes down
              lower( el_position);
          endcompress;
          exit;
      endparallel;
   endloop;
endloop;
endproc;
```

Table II.5

2.f. Dormant

The dormant module is a straightforward mapping from the temporal logic statements. The additional statements were added to include fairness explicitly in the model. This is done by alternately assigning an up or down floor directional signal when simultaneous requests in both directions are raised. There is no action taken to move the elevator in this module. The conditions raised cause the correct action to be taken by the stopping_at_floor module.

```
procedure at_dormant
loop
    wait(dormant);
    parallel
        wait(el_dest_lit);
        compress
           lower(dormant); lower(el_doors_closed);
        endcompress;
        break;
    ||  wait(fl_summ_lit_up & !fl_summ_lit_dn);
        compress
           raise(fl_direction_up); raise(el_direction_up);
           lower(dormant); lower(el_doors_closed);
        endcompress;
        break;
    ||  wait(fl_summ_lit_dn & !fl_summ_lit_up);
        compress
           raise(fl_direction_dn); raise(el_direction_dn);
           lower(dormant); lower(el_doors_closed);
        endcompress;
        break;
    ||  wait(fl_summ_lit_up & fl_summ_lit_dn & non_determ);
        compress
           raise(fl_direction_up); raise(el_direction_up);
           lower(dormant); lower(non_determ); lower(el_doors_closed);
        endcompress;
        break;
    ||  wait(fl_summ_lit_up & fl_summ_lit_dn & !non_determ);
        compress
           raise(fl_direction_dn); raise(el_direction_dn);
           lower(dormant); raise(non_determ); lower(el_doors_closed);
        endcompress;
        break;
    ||  wait(continue_up  & !( fl_summ_lit_up | fl_summ_lit_dn ));
        compress
           raise(el_direction_up);
           lower(dormant);
        endcompress;
        break;
    ||  wait(continue_dn  & !(continue_up | fl_summ_lit_up | fl_summ_lit_dn ));
        compress
           raise(el_direction_dn);
           lower(dormant);
        endcompress;
        break;
   endparallel;
endloop;
endproc;
```

Table II.6

2.g. At_above and At_below

The first procedure (Table II.7) models the elevator being above the completely modeled middle floor. When the elevator comes here, it delays for one second, thus allowing some time for it to arrive. Then it waits for a condition to cause it to turn around and go down, and finally it returns the elevator to the lower floor.

The second procedure (Table II.8) is similar, and models the elevator being on the below floor.

```
procedure at_above
loop
   wait(el_above & el_direction_up);
   delay 1;
   wait(fl_summ_lit_up | fl_summ_lit_dn | el_dest_lit);
   compress
      lower (el_direction_up);
      raise (el_direction_dn);
      lower(el_above);
      raise(el_position);
   endcompress;
endloop;
endproc;
```

Table II.7

```
procedure at_below
loop
   wait(el_below & el_direction_dn);
   wait(fl_summ_lit_up | fl_summ_lit_dn | el_dest_lit);
   compress
      lower (el_direction_dn);
      raise (el_direction_up);
      lower(el_below);
      raise(el_position);
   endcompress;
endloop;
endproc;
```

table II.8

2.h. Second Version of Dormant

This is a second version of the procedure dormant, and is slightly out of context with the previously shown version. In this version, not only are the affected conditions represented more procedurally, but the action to leave the floor is also embedded in the procedure. This version could not replace the previously described version directly in the complete model, but it is still useful to see the difference. The two parallel branches are described below.

1. In the first branch, the conditions are modeled for opening the doors when a destination push button at the middle floor, or one of the summons buttons at the middle floor is depressed. The conditions for which floor directional light to turn on are then determined, the light is turned on, and the doors opened.
2. In the second branch, the conditions for leaving the floor are awaited. When this occurs, the preferred direction is to go up, but if there is no request to go up, the elevator goes down. Note that in this version, the branch models the elevator's leaving the floor.

```
procedure dormant_at_floor
loop
    wait(dormant)
    parallel
      wait(el_dest_lit | fl_summ_lit_up | fl_summ_lit_dn);
      if(fl_summ_lit_up & fl_summ_lit_dn & non_determ |
         fl_summ_lit_up & !fl_summ_lit_dn |
         el_dest_lit & !fl_summ_lit_dn & !continue_dn)
      then
        compress
           raise(fl_direction_up); raise(el_direction_up);
           raise(el_doors_closed);  lower(non_determ); lower(dormant);
        endcompress;
      else
        compress
           raise(fl_direction_dn); raise(el_direction_dn);
           raise(el_doors_closed);  raise(non_determ); lower (dormant);
        endcompress;
      endif;
      break;
   ||
      wait( (continue_up | continue_dn ) & !( fl_summ_lit_up | fl_summ_lit_dn
           | el_dest_lit));
      if(continue_up)
      then
         compress
            raise(el_direction_up); raise(el_above); lower(el_position);
            lower(dormant);
         endcompress;
      else
         compress
            raise(el_direction_dn); raise(el_below); lower(el_position);
            lower(dormant);
         endcompress;
      endif;
      exit;
   endparallel;
endloop;
endproc;
```

Table II.9

References

[Barringer 87] Barringer, H.
Up And Down the Temporal Way.
Computer Journal 30(2):134-148, April, 1987.

[Berry 84] Berry, G. and L. Cosserat, L.
Synchronous programming of reactive systems an introduction to ESTEREL.
Seminar in Concurrency , 1984.

[Clarke 86] Clarke, E.M., Emerson, E.A., and Sistla, A.P.
Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications.
ACM Transactions on Programming Languages and Systems 8(2):244-263, April, 1986.

[Clarke 87] Clarke, E. M. and Grumberg, O.
Research on Automatic Verification of Finite-State Concurrent Systems.
Ann. Rev. Computer Science (2):269-290, 1987.

[Firth 87] Firth, R., Wood, W., Pethia, R., Roberts Gold, L., Mosley, V., Dolce, T.
A Classification Scheme for Software Development Methods.
Technical Report CMU/SEI-87-TR-41, DTIC: ADA200606, Software Engineering Institute, 1987.

[Harel 86] Harel, David.
Statecharts: A Visual Approach to Complex Systems.
Concurrent Systems , February, 1986.

[Hoare 85] Hoare, C.A.R.
Communicating Sequential Processes.
Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

[Hughes 68] Hughes, G.E. and Cresswell, M.J.
An Introduction to Modal Logic.
Methuen and Co., New York, NY, 1968.

[Lamport 83] Lamport, L.
What Good is Temporal Logic.
Information Processing 83 :657-668, 1983.

[Lamport 86] Lamport, L.
A Simple Approach to Specifying Concurrent Systems.
Technical Report, DEC, December, 1986.

[Pnueli 85] Pnueli, A.
Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends.
Lecture Notes in Computer Science #224 :510-584, 1985.

[Rescher 71] Rescher, Nicholas and Urquhart, Alasdair.
Temporal Logic.
Springer-Verlag, Wien, New York, 1971.

[Schwartz 87] Schwartz, M. D. and Delisle, N. M.
Specifying A Lift Control System With CSP.
In *Fourth International Workshop on Software Specification and Design*, pages 21-27. The Computer Society of the IEEE, April, 1987.

[SPC 88] Software Productivity Consortium.
Strategies for Introducing Formal Methods Into the Ada Life Cycle.
Technical Report SPC-TR-88-002, Software Productivity Consortium, January, 1988.

[Wood 88] Wood, W., Pethia, R., Roberts Gold, L., Firth, R.
A Guide to the Assessment of Software Development Methods.
Technical Report CMU/SEI-88-TR-8, DTIC: ADA197416, Software Engineering Institute, 1988.

[Woodcock 87] Woodcock, J. C. P., King, S., and Sorensen, I. H.
Mathematics for Specification and Design: The Problem with Lifts.
In *Fourth International Workshop on Software Specification and Design*, pages 265-268. The Computer Society of the IEEE, April, 1987.

Table of Contents

List of Figures