Up and Down The Temporal Way

H. BARRINGER*
Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL

A formal specification of a multiple-lift system is constructed. The example illustrates and justifies one of many possible system specification styles based on temporal techniques.

Received September 1985

1. INTRODUCTION

Over the last decade there has been widespread research directed at obtaining techniques for the analysis, specification and development of concurrent systems. Several of these lines of research have led to the belief that temporal logic is a useful tool for reasoning about such systems.[1–4] The use of temporal logic enables, in particular, analysis of both safety and liveness properties in a single uniform logical framework (see Ref. 5 for extensive examples). More recently, techniques have been developed for achieving compositional temporal proof systems.[6,7] Compositionality is an essential requirement for the hierarchical development of implementations from formal specifications. Without compositionality, the check on consistency of a development step would be delayed until all interactions between the developed components are known, essentially, at the implementation level; clearly, it could be rather costly if such a consistency check then showed that the system did not achieve the overall specification. In general, compositionality can be achieved by realising that a specification of any component must include assumptions about the behaviour of the environment in which the component will reside. In the temporal framework, this requires that one can distinguish actions made by a component from those made by its environment; in Refs 6 and 7 the coarse technique of labelling actions is used for just that purpose. Although it is sometimes possible to distinguish actions by other means, as will be shown in a forthcoming article, in the presentation of the specifications below the more general labelling technique is assumed.

In this paper, I present a rather different application of our temporal specifications. Over the last year (1984/5), a lift control example has been gathering interest amongst attendees of the Ada-UK Formal Methods Group, and more recently amongst attendees of the Ada-Europe Working Group on Formal Methods. On many occasions I have been asked to produce a solution for the Ada-UK group, but I never was quite able to make their meetings! However, I believe the following sections contain what I would have presented.

Section 2 contains a statement of the original problem as set by Neil Davis of STL,† and some comments on the interpretation I have taken. Section 3 presents a first approximation, a much simpler single-lift system. Section 4 discusses some initial modifications and Section 5 presents the multiple-lift system. Final comments are made in Section 6. For convenience, an appendix contains the semantics of the logic used in this article.

* Research supported by SERC grant GR/C/05760.
† Previously STC IDEC Ltd.

2. INFORMAL REQUIREMENTS

The following is a description of the lift-control system problem as set by Neil Davis.

A Lift-Control System

An n-lift system is to be installed in a building with m floors. The lifts and the control mechanism are supplied by a manufacturer. The internal mechanisms of these are assumed (given) in this problem.

Design the logic to move lifts between floors in the building according to the following rules.

(1) Each lift has a set of buttons, one button for each floor. These illuminate when pressed and cause the lift to visit the corresponding floor. The illumination is cancelled when the corresponding floor is visited (i.e. stopped at) by the lift.

(2) Each floor has two buttons (except ground and top), one to request an up-lift and one to request a down-lift. These buttons illuminate when pressed. The buttons are cancelled when a lift visits the floor and is either travelling in the desired direction, or visiting the floor with no requests outstanding.

In the latter case, if both floor-request buttons are illuminated, only one should be cancelled. The algorithm used to decide which to service should minimise the waiting time for both requests.

(3) When a lift has no request to service, it should remain at its final destination with its doors closed and await further requests (or model a 'holding' floor).

(4) All requests for lifts from floors must be serviced eventually, with all floors given equal priority (can this be proved or demonstrated?).

(5) All requests for floors within lifts must be serviced eventually, with floors being serviced sequentially in the direction of travel (can this be proved or demonstrated?).

(6) Each lift has an emergency button which, when pressed, causes a warning signal to be sent to the site manager. The lift is then deemed 'out our service'. Each lift has a mechanism to cancel its 'out of service' status.

2.1 My interpretation of the problem

The above problem description states '*Design the logic...*'. Rather than present a model of a lift system which would hopefully satisfy the above informal constraints, the following sections of this paper present an axiomatic specification of a lift system that possesses the above properties. It is claimed that such a specification could be used, say, as a top-level description in a formal development of a lift system. One argument

for adopting this approach can be phrased as follows. If some model were given as a specification unless one accepts the given model at face value it would still be necessary to formalise the informal requirements in order to demonstrate that the model is the right one, i.e. behaves in the desired manner. Once such formalisation of requirements has been made why not use it as the specification? Clearly, the difficult question is 'how is such informality made formal?'.

The approach taken here is first to fix on the observable quantities, then describe the behaviour of the system in their terms. Designing criteria to determine what should and shouldn't be observable is a difficult problem and one which is avoided here. However, one must take great care in choosing what the observable quantities should be, i.e. in determining the right level of abstraction. If one looks with 'X-ray eyes' at a lift system, say, then it is likely that one will see parts one was not meant to see and which are likely to cloud the issue, for example, triggers for causing the lift to decelerate.

Having decided upon the observable quantities, the informal requirements are rephrased in their terms. This leads to an axiomatic specification of the system (and to all the problems that then ensue).

3. A SINGLE-LIFT SYSTEM

To motivate the specification for the multiple-lift system, a simpler single-lift system is first introduced. The informal requirements placed upon the system are as follows.

(1) The lift has a set of buttons, one button for each floor. The buttons illuminate when pressed and cause the lift to visit the corresponding floor. The illumination is cancelled when the floor is visited.

(2) Each floor has one request button. The buttons illuminate when pressed and cause the lift to stop at that floor at the next possible moment. The illumination is cancelled when the lift stops at the floor.

(3) When there are no outstanding requests for the lift, it should remain stationary at the last floor visited with the doors closed.

(4) (*a*) Every request for the lift must eventually be serviced.
(*b*) The lift should not stop at floors not requesting service.

(5) The lift should travel as far as possible without changing direction.

Note that a clause corresponding to 4*b* is missing from the main problem description. This, I believe, must be an oversight and, of course, demonstrates the dangers of informal specification (the oversight was actually discovered during the development of the formal specification).

3.1 Observable quantities

As mentioned in Section 2 above, the first step to be taken in obtaining the formal specification for the single-lift system is to agree upon the externally observable quantities. Of course, some help comes by analysing the informal requirements. More generally though, what does one observe as a user of a lift system? There are buttons to push, which also illuminate (so there must be corresponding lights), and there are lift doors (which hopefully open and close). In fact it is just those quantities, i.e. the buttons, lights and doors, which are used to describe the behaviour of the lift system. However, with regard to observation, it should be noted that the observer here is not really a user of the lift system. It is assumed the observer has a global view of the system; the state of all the lift doors and buttons (both inside and outside the lift) can be observed at any one moment.

The interface between lift system and its environment must therefore consist of these three quantities.

3.1.1 *Lift doors*

At this initial level of abstraction the doors of the lift are considered to be either in an open state (here represented by *OpenDoors*) or in a closed state (*ClosedDoors*) with no intermediate states. In the next section, where various kinds of interruption are considered, it is necessary to introduce intermediate door states.

$$DoorState = OpenDoor \mid ClosedDoors$$

$$D : \textbf{array}\ (1\ldots m)\ \textbf{of}\ DoorState$$

Thus, the state of all lift doors is given by the vector D. For convenience below, we refer to the state of the lift doors on floor i by D_i. In fact, we use that abbreviation for all vector subscription.

3.1.2 *Lift request buttons*

Both types of buttons have the same possibilities in state, a button may be in either a depressed state (Ooh, it does hurt so when I'm pushed!) or a released state. Again, for ease in the description below ***TRUE*** is used to represent the depressed state, and ***FALSE*** to represent the released state. Clearly, there are the external buttons (for calling a lift)

$$C : \textbf{array}\ (1\ldots m)\ \textbf{of}\ Boolean$$

and the internal buttons (for sending a lift to a floor)

$$S : \textbf{array}\ (1\ldots m)\ \textbf{of}\ Boolean.$$

Thus if C_i is true then somebody has the call button on floor i depressed and, likewise, if S_i is true then somebody has the 'send button for floor i' depressed.

3.1.3 *Lights*

The lights illuminating the lift request buttons can, obviously, be either on or off. Again, for ease, ***TRUE*** has been chosen to represent on and ***FALSE*** to represent off. There are the lights associated with the call buttons C

$$LC : \textbf{array}\ (1\ldots m)\ \textbf{of}\ Boolean$$

and lights associated with the send buttons S

$$LS : \textbf{array}\ (1\ldots m)\ \textbf{of}\ Boolean.$$

Note that there is a difference between the lights and the buttons which forces both to be used in the specification. The light within a button may well remain lit after that button has been released.

3.2 Behaviours

The behavioural specification of this first lift system is now presented as a collection of temporal formulae over

the 'interface' (D, C, S, LC, LS). Given that a model of a temporal formula is a possibly infinite state sequence, the intention is that the models of the temporal specification are just those state sequences which could be generated by any lift system possessing the desired properties. The temporal language used is essentially a propositional temporal μ-calculus which extends the linear temporal logic by allowing recursive definitions of temporal predicates. Rather than describe the language formally here, it is introduced by example. For those readers wishing a more complete description, Appendix 1 contains its interpretation over *ω-sequences*. Furthermore, Ref. 8 describes the language in detail and provides a proof of its expressive equivalence with extended temporal logic (ETL) of Ref. 9.

3.2.1 *Buttons and lights*

The interaction between buttons and lights (and doors) is given first. These interactions are examples of safety properties.[3]

$$\Box \bigwedge_i (C_i \Rightarrow (LC_i \mathbin{\mathcal{W}} \textbf{service at } i)) \tag{1}$$

where

$$\textbf{service at } i \stackrel{\text{def}}{=} D_i \in \mathit{OpenDoors}$$

$\Box$ is the 'always in the future' temporal operator. Hence if $\Box\phi$ is true, then the formula ϕ is true now and in every future moment. The weak binary until $\mathcal{W}$ (usually referred to as unless) has also been used. Informally, if $\phi \mathcal{W} \psi$ is true then either ϕ holds for ever, i.e. in every future moment, or ϕ holds continuously until at least ψ is true. The above formula states therefore that if ever any call button is depressed then it gets illuminated (at the same moment) and stays illuminated unless the request gets serviced. Servicing of a request at a floor is simply characterised by the fact that the lift doors for that floor are neither closed nor in a closing state (which means the lift must be present and able to accept passengers).

$$\Box \bigwedge_i (S_i \Rightarrow (LS_i \mathbin{\mathcal{W}} \textbf{service at } i)) \tag{2}$$

Similarly for the send buttons inside the lift. If one is depressed then either it is illuminated or the lift is at the floor able to provide immediate service.

$$\Box \bigwedge_i (\neg LC_i \Rightarrow (\neg LC_i \mathbin{\mathcal{W}} C_i)) \tag{3}$$

This property states that if ever a call light is unlit, it must remain unlit unless the call button is depressed. Actually, of course, because of the first property given above, even if the C_i button is depressed the light may still remain unlit if the lift is currently at that floor.

$$\Box \bigwedge_i (\neg LS_i \Rightarrow (\neg LS_i \mathbin{\mathcal{W}} S_i)) \tag{4}$$

This is similar to the above for the send button lights.

Finally, under this heading it is necessary to express that if a light in a button is lit then there is no immediate service at present, i.e. the corresponding lift doors are in a closed state.

$$\Box \bigwedge_i ((LC_i \Rightarrow \neg \textbf{service at } i) \wedge (LS_i \Rightarrow \neg \textbf{service at } i)) \tag{5}$$

Actually, the above five properties can be combined as the following single property.

$$\begin{aligned} \Box \bigwedge_i \big(& ((C_i \Rightarrow (LC_i \mathbin{\mathcal{W}} \textbf{service at } i)) \wedge \\ & ((\textbf{service at } i \vee \neg LC_i) \Rightarrow (\neg LC_i \mathbin{\mathcal{W}}^+ C_i)) \wedge \\ & (S_i \Rightarrow (LS_i \mathbin{\mathcal{W}} \textbf{service at } i)) \wedge \\ & ((\textbf{service at } i \vee \neg LS_i) \Rightarrow (\neg LS_i \mathbin{\mathcal{W}}^+ S_i))\big) \end{aligned} \tag{6}$$

This has introduced the strict version of the binary unless operator. $\phi \mathcal{W}^+ \psi$ is equivalent to $\phi \wedge \bigcirc(\phi \mathcal{W} \psi)$ and hence requires ϕ to be true now. This neater expression of the property is used in preference to the formulae (1), (2), (3), (4) and (5) above!

In summary, the above formula (6) fixes the interplay between buttons and lights. The description does not fix how buttons get pushed or released (that is controlled by the user's finger, see 3.2.4); it does, however, fix how the light inside the button goes on and off (controlled by the state of the button and the lift doors). In fact, if a button is pushed and never released then the lift will always eventually return to that floor whenever it leaves it!

3.2.2 *Lift door behaviour*

Since, at this initial level of development of the list specification, there are only two states for the doors (open or closed) the only requirement that needs to be placed is that lift doors on different floors are not open at the same time (otherwise there may be passengers falling down the lift shaft!). This requirement is captured by following (health and) safety property.

$$\Box \Big(\bigwedge_i D_i \in \mathit{ClosedDoors} \vee \bigvee_i \Big(D_i \in \mathit{OpenDoors} \wedge \bigwedge_{j \neq i} D_j \in \mathit{ClosedDoors} \Big) \Big) \tag{7}$$

So, it is always the case that either all the doors are properly closed, or that at most one door is open. By overloading notation this can be more succinctly written as below.

$$\Box \Big(\Big(\sum_i D_i \in \mathit{OpenDoors} \Big) \leqslant 1 \Big)$$

3.2.3 *Servicing behaviour*

Here constraints on the order of servicing requests are constructed. Three basic constraints are made.

(1) All requests must eventually be serviced, and no floor should be stopped at when there is no unserviced request for that floor.

(2) The lift should not pass a floor for which there is an outstanding (unserviced) request.

(3) The lift should not change direction if there is an outstanding request whose service would cause the lift to continue in the same direction.

In order to express these constraints formally, it is rather useful to define some higher-level temporal interval schema. First is the notion of the lift travelling uninterruptedly from floor i, say, to floor j. This is so when the doors on floor i remain unclosed until all doors (possibly apart from j) are closed until the doors on floor

j are not closed. The diagram below typifies this situation; time increase from left to right.

$D_i \in OpenDoors$	$\bigwedge_k D_k \in ClosedDoors$	$D_j \in OpenDoors$
Lift seen at floor i	Lift nowhere to be seen	

Formally, this is written as below. For convenience, the interval is constructed from when the door is in an open state (rather than including the closing state).

$$\textbf{from } i \textbf{ to } j \stackrel{\text{def}}{=} D_i \in OpenDoors \; \mathcal{U}^+ \; ((\bigwedge_k D_k \in ClosedDoors) \; \mathcal{U} \; D_j \in OpenDoors)$$

Note that the strict version ($\mathcal{U}^+$) of the strong until ($\mathcal{U}$) has been used. The temporal formula $\phi \; \mathcal{U} \; \psi$ is equivalent to $(\phi \; \mathcal{W} \; \psi) \wedge \Diamond \psi$ and hence guarantees that ψ will eventually hold. The use above ensures that the predicate characterises an inclusive interval 'from i to j'.

A similar characterisation can be given for the lift just having come from a floor to its current position. This uses the past time equivalent to the binary until operator, the since ($\mathcal{S}$) operator. The temporal formula $\phi \; \mathcal{S} \; \psi$ is true when ϕ has held in every moment since ψ held and, furthermore, ψ did actually hold in the past.

$$\textbf{at } i \textbf{ from } j \stackrel{\text{def}}{=} D_i \in OpenDoors \; \mathcal{S}^+ \; ((\bigwedge_k D_k \in ClosedDoors) \; \mathcal{S} \; D_j \in OpenDoors)$$

The characterisation of the lift travelling from floor i stopping off at floor j then going on to floor k can be constructed by conjoining the above two definitions.

$$\textbf{from } i \textbf{ via } j \textbf{ to } k \stackrel{\text{def}}{=} \textbf{at } j \textbf{ from } i \wedge \textbf{from } j \textbf{ to } k$$

Unserviced requests can also be characterised using the past time since operator. Thus,

$$\textbf{request } k \textbf{ is unserviced} \stackrel{\text{def}}{=} (LC_k \; \mathcal{S} \; C_k) \vee (LS_k \; \mathcal{S} \; S_k)$$

states that a request for the lift to go to floor k is unserviced if either the light on floor k has remained illuminated since the button was pushed or the light in the lift for the kth button has remained illuminated since that button was pushed. Remember from (6) above that the light being illuminated means that some call is unserviced.

An alternative formulation of this requirement is that the doors on floor k have been closed since the request to go to floor k. Such a description is rather awkward to modify when two call buttons per floor are introduced in Section 4.3. Also, notice that the above definition indicates that if the lift is at a floor and the doors start to close then depression of the call button will not keep the lift at that floor. This may seem unreasonable, however, so an alternative strategy is developed in the next section.

Building upon this notion, a lift leaves a floor with an outstanding request if the following formula is true.

$$\textbf{left } i \textbf{ with request } k \textbf{ unserviced} \stackrel{\text{def}}{=} D_i \in OpenDoors \; \mathcal{U}^+ (D_i \in ClosedDoors \wedge \textbf{request } k \textbf{ is unserviced})$$

The expression of the constraints 1, 2 and 3 above is now straightforward. First, all requests must be eventually serviced, and only requests are serviced.

$$\bigwedge_i \Box ((C_i \vee S_i) \Rightarrow \Diamond \textbf{ service at } i) \tag{8}$$

$$\bigwedge_i \Box (\textbf{service at } i \Rightarrow (\textbf{service at } i \; \mathcal{S} \; \textbf{request } i \textbf{ is unserviced}) \tag{9}$$

Next, the lift does not miss floors.

$$\bigwedge_i \bigwedge_j \Box \; (\textbf{from } i \textbf{ to } j \Rightarrow \neg \bigvee_k (\textbf{left } i \textbf{ with request } k \textbf{ unserviced} \wedge k \textbf{ between } i \textbf{ and } j)) \tag{10}$$

where

$$k \textbf{ between } i \textbf{ and } j \stackrel{\text{def}}{=} \min(i, j) < k < \max(i, j)$$

Consider any two consecutive stops which the lift makes, say at floor i and then at j. There must not be an outstanding request, which was made before leaving floor i, for some floor between i and j.

Finally, the lift must keep travelling as far as possible without any change of direction.

$$\bigwedge_i \bigwedge_j \bigwedge_k \Box \; (\textbf{from } i \textbf{ via } j \textbf{ to } k \Rightarrow (i < j > k \Rightarrow \neg \bigvee_l (\textbf{left } j \textbf{ with request } l \textbf{ unserviced} \wedge i < j < l) \wedge i > j < k \Rightarrow \neg \bigvee_l (\textbf{left } j \textbf{ with request } l \textbf{ unserviced} \wedge i > j > l))) \tag{11}$$

This constraint considers the times when the lift changes direction. Consider any three consecutive stops, i, j and k. If the lift changed direction from up to down ($i < j > k$) then, when leaving floor j, there must not be an outstanding request for a floor above floor j (as the lift should travel in the same direction for as long as possible). Similarly, for the change from down to up ($i > j < k$).

3.2.4 *Actions*

As yet only interactions between the observable quantities have been described. Although the following three clauses (12, 13, 14) are not essential, it is useful to distinguish environment actions from component actions, using the E and Π propositions of the computational model.[10, 6, 7] Here, actions of the environment are considered to be just those which a passenger of the lift may make. Since in this first system a passenger can only push or release buttons, the environment action proposition E is identified as below.

$$\Box (E \Rightarrow \bigvee_i (C_i \wedge \bigcirc \neg C_i \vee \neg C_i \wedge \bigcirc C_i \vee S_i \wedge \bigcirc \neg S_i \vee \neg S_i \wedge \bigcirc S_i)) \tag{12}$$

The component actions are those that the lift makes. Essentially, the 'driving' actions are the door-movement actions. These are identified by the proposition Π.

$$\Box(\Pi \Rightarrow \bigvee_i (D_i \in ClosedDoors \wedge \bigcirc D_i \in OpenDoors \vee D_i \in OpenDoors \wedge \bigcirc D_i \in ClosedDoors) \quad (13)$$

Note that the switching on (or off) of the lights occurs as a consequence of the environment requests (or lift door actions).

$$\Box(E \vee \Pi) \quad (14)$$

This clause ensures that all actions satisfy the requirements implied by (12) and (13).

3.2.5 *Initial conditions*

Initially, the lift doors are all closed and there are no outstanding requests for lift service. The position of the lifts is, of course, unknown to an observer of the system.

$$\bigwedge_i (D_i \in ClosedDoors \wedge \neg C_i \wedge \neg S_i \wedge \neg LC_i \wedge \neg LS_i) \quad (15)$$

3.3 General remarks

The overall specification for the single-lift system is thus the conjunction of all the appropriate preceding temporal formulae. For example,

*LIFTSPEC*1

(15) ∧	*initial conditions*
(12) ∧ (13) ∧ (14) ∧	*action definitions*
(6) ∧	*call, send service consequences*
(7) ∧	*lift door sequencing*
(8) ∧ (9) ∧ (10) ∧ (11)	*servicing behaviours*

Given some implementation, say *LIFT*, it must then be proved that

$$\vdash \{LIFT\}\, LIFTSPEC1$$

(following the proof systems presented in Refs 6–8).

In this paper such a development is not continued. It should be observed that the specification as presented places no restriction on the model of parallelism, whether interleaving or otherwise. For example, it is possible for buttons to be pushed when doors are opening, etc. However, it should also be observed that the model of parallelism assumed by the proof systems mentioned above is interleaving. Hence a model of the implementation would not have such overlap of environment and component actions.

3.4 Exercising the specification

Having produced a formal specification, it can sometimes be useful and reassuring to exercise or test it in some sense to determine whether certain properties hold. There are differing views on how this should be done. Some researchers believe this should be a mental activity, others believe that formal specifications should be executable. What can be done with the above formal specification? It is not executable (however note that some temporal logic is executable),[11] but the basic language used is decidable and hence formulae can be mechanically decided. Thus the specification can be exercised by formalising queries about the system and then proving that they are valid deductions.

Such an approach was adopted for verifying a few simple properties of the single-lift system. For example, it was validated that if no requests are ever made then no lift door ever opens, if a request is made then the associated illumination eventually gets switched off, etc. These properties were checked with the decision procedures described in Ref. 12.

4. ENHANCEMENTS TO THE SINGLE LIFT

In this section we consider three enhancements to the system specified above. The first allows the doors to be interrupted whilst they are being closed. The second handles an emergency button as mentioned in Section 2 above. The third, seemingly quite trivial, extension is to consider request buttons as originally defined in the problem description.

4.1 'Foot in the door'

In the Mathematics Tower in Manchester there are two lifts to service 18 floors. Imagine one of the lifts is inoperable and the other is behaving as specified above. You are at the ground floor and see the lift just about to leave, i.e. the doors are closing (in fact the doors are either fully open or fully closed). Unfortunately, there is no immediate way of stopping the doors from closing. A foot in the door just seems to send you to hospital! Clearly, there is a need to allow this kind of interruption. The basic problem is, however, that this type of interruption is uncontrollable. Such an interruption comes from the environment and therefore a badly behaved environment could stop the lift from ever leaving a floor. Thus, guaranteed servicing of requests can no longer be promised. The obvious place to start the modification is the 'lift door opening and closing' sequence behaviour. It is clearly necessary to introduce intermediate door states, and so the following is now taken as the door interface.

$$\begin{aligned} DoorState &= \underline{CLOSED} \mid \underline{OPENING} \mid \underline{OPEN} \mid \underline{CLOSING} \\ OpenDoors &= \underline{OPENING} \mid \underline{OPEN} \mid \underline{CLOSING} \\ ClosedDoors &= \underline{CLOSED} \end{aligned}$$

The basic operation of any lift door causes it to cycle through the possible states, i.e. from *CLOSED* to *OPENING* to *OPEN* to *CLOSING* to *CLOSED* and so forth. More formally,

$$\bigwedge_i (\nu\xi . [D_i = \underline{CLOSED}\ \mathcal{W}^+ (D_i = \underline{OPENING}\ \mathcal{U}^+ (D_i = \underline{OPEN}\ \mathcal{U}^+ (D_i = \underline{CLOSING}\ \mathcal{U}^+ \xi)))]) \quad (16)$$

The temporal formula $\nu\xi . \chi(\xi)$ denotes the maximal fixed point solution to the temporal equivalence $\phi \Leftrightarrow \chi(\phi)$, where χ is a formula dependent on the temporal variable ϕ (one can think of χ as the body of a recursive definition).

Thus a door may remain closed for ever ($D_i = \underline{CLOSED}\ \mathcal{W}^+ \ldots$), but should it start to open it will eventually be fully open ($\ldots \mathcal{U}^+ (D_i = \underline{OPEN} \ldots)$), will eventually start to close ($D_i = \underline{OPEN}\ \mathcal{U}^+ (D_i = \underline{CLOSING} \ldots)$) and finally repeat the cycle ($\ldots \mathcal{U}^+ \xi$).

The use of strict operators guarantees that every door state will be entered.

Although the basic door control behaviour has been described recursively, such behaviour could have been described using just the $\Box$, $\mathcal{U}$ and $\mathcal{W}$ temporal operators. For example,

$$\bigwedge_i \Box((D_i = \underline{CLOSED} \Rightarrow (D_i = \underline{CLOSED}\ \mathcal{W}\ D_i = \underline{OPENING})) \wedge (D_i = \underline{OPENING} \Rightarrow (D_i = \underline{OPENING}\ \mathcal{U}\ D_i = \underline{OPEN})) \wedge (D_i = \underline{OPEN} \Rightarrow (D_i = \underline{OPEN}\ \mathcal{U}\ D_i = \underline{CLOSING})) \wedge (D_i = \underline{CLOSING} \Rightarrow (D_i = \underline{CLOSING}\ \mathcal{U}\ D_i = \underline{CLOSED}))) \wedge \bigwedge_i D_i = \underline{CLOSED}$$

However, it is felt that this is rather clumsy. Also, do note that such elimination of recursion is not always possible.

Now it must be arranged that the door closing sequence can be interrupted. A fairly common behaviour is to reopen the door when it gets jammed, and then to try closing it again. Of course, that sequence may go on indefinitely. This latter behaviour is most easily expressed using two 'recursive definitions'; a vector of fixpoints results.

$$\bigwedge_i (\nu\langle\xi_1, \xi_2\rangle . \langle D_i = \underline{CLOSED}\ \mathcal{W}^+ \xi_2, D_i = \underline{OPENING}\ \mathcal{U}^+ (D_i = \underline{OPEN}\ \mathcal{U}^+ (D_i = \underline{CLOSING}\ \mathcal{U}^+ (\xi_1 \vee \xi_2))))\rangle)_1 \quad (17)$$

So far so good, but there is the small matter of how the normal closing sequence is enforced when no feet get in the way. An extra interface vector variable is introduced.

$$J : \textbf{array}(1\,..\,m)\ \textbf{of}\ \textit{Boolean}$$

If a foot gets in the way during the closure of the door on floor i, J_i is set from $\underline{FALSE}$ to $\underline{TRUE}$ which, in turn, will cause the doors to start opening again. Thus both the environment (12) and component actions (13) require change.

$$\Box(E \Rightarrow \bigvee_i (C_i \wedge \bigcirc\neg C_i \vee \neg C_i \wedge \bigcirc C_i \vee S_i \wedge \bigcirc\neg S_i \vee \neg S_i \wedge \bigcirc S_i \vee \neg J_i \wedge \bigcirc J_i)) \quad (18)$$

$$\Box(\Pi \Rightarrow \bigvee_i (D_i = \underline{CLOSED} \wedge \bigcirc D_i = \underline{OPENING} \vee D_i = \underline{OPENING} \wedge \bigcirc D_i = \underline{OPEN} \vee D_i = \underline{OPEN} \wedge \bigcirc D_i = \underline{CLOSING} \vee D_i = \underline{CLOSING} \wedge \bigcirc(D_i = \underline{CLOSED} \vee D_i = \underline{OPENING}))) \quad (19)$$

Next, a jam can only occur if the appropriate door is closing and when one does occur the door must eventually start opening again.

$$\bigwedge_i \Box(J_i \Rightarrow (D_i = \underline{CLOSING}\ \mathcal{U}^+\ D_i = \underline{OPENING})) \quad (20)$$

Because the door state *OpenDoors* now includes the $\underline{CLOSING}$ state, it is necessary to redefine the predicate **service at** i.

$$\textbf{service at}\ i \stackrel{\text{def}}{=} D_i = \underline{OPENING} \vee D_i = \underline{OPEN}$$

And now, of course, eventual service can only be guaranteed if a door does not get continually stopped from closing properly (21).

$$\bigwedge_i \Box((C_i \vee S_i) \Rightarrow \Diamond(\textbf{service at}\ i \vee \bigwedge_i \Box(D_j \in \mathit{OpenDoors}))) \quad (21)$$

Also, a door may now open because of a jam (22).

$$\bigwedge_i \Box(\textbf{service at}\ i \Rightarrow (\textbf{service at}\ i\ \mathcal{S}\ (\textbf{request}\ i\ \textbf{is unserviced} \vee J_i)) \quad (22)$$

4.2 Other emergencies

Again it is usual to find an emergency or alarm button inside the lift compartment. Here the above specification is extended to handle such traumatic occurrences. In fact the 'emergency' button as described in Section 2 above will be specified.

The lift system can be thought of as cycling through 'in-service' and then 'out-of-service' phases. The 'in-service' phase is essentially the earlier lift behaviour; the 'out-of-service' phase starts with an alarm 'to the site manager' and then waits (allowing anything to happen) for the lift system to be put back in service. Assuming *INSERVICE* is a temporal formula which specifies 'in-service' behaviour and, correspondingly, *OUTOFSERVICE* specifies out-of-service behaviour, the lift service behaviour is given by the following temporal formula.

$$\mathit{LIFTSPEC2} \stackrel{\text{def}}{=} \nu\xi . (\mathit{INSERVICE}\ \mathscr{C}\ \mathit{OUTOFSERVICE}\ \mathscr{C}\ \xi)$$

This definition introduces the temporal operator $\mathscr{C}$ (combine).

Informally, the combine of two temporal formulae, ϕ and ψ, is a temporal formula whose models (behaviour sequences) can be seen as the fusion (concatenation with overlap) of the models of the ϕ with those of ψ. Thus the *LIFTSPEC2* behaviour cycles around the composition of the *INSERVICE* behaviour followed by the *OUTOFSERVICE* behaviour. The composition is such that *INSERVICE* may continue for ever if the emergency button is not depressed.

To supply the appropriate signals from and to the environment two further boolean variables are introduced as part of the interface state, *Em* (for emergency) and *Al* (for alarm). Like the other lift buttons, *Em* is $\underline{TRUE}$ when in a depressed state and $\underline{FALSE}$ when in the released state. Also, associated with the button is its illuminating light *LEm* which is $\underline{FALSE}$ when unlit, etc. *Al* is $\underline{TRUE}$ when the alarm is ringing and $\underline{FALSE}$ when silent. The 'out-of-service' behaviour is presented first.

$$\mathit{OUTOFSERVICE} \stackrel{\text{def}}{=} (\mathit{Al} \wedge \mathit{LEm})\ \mathcal{W}^+\ (\mathit{LEm}\ \mathcal{W}^+\ (\textbf{init} \wedge \textbf{fin}))$$

where

$$\textbf{fin} \stackrel{\text{def}}{=} \neg \oplus \underline{TRUE}$$

and **init** defines some re-initialisation (here unspecified).

$\oplus$ is the strong (or existential) next-time operator, introduced because of the possibility of no future state. $\oplus\phi$ will be true only if there is a next moment in the future and ϕ holds in that moment (note that $\bigcirc$ is taken as a weak (or universal) next-time operator, $\bigcirc\phi$ will be vacuously true if there is no future state). **fin**, therefore, will only be true in those states that have no future and is thus a formula which can determine the end of time. (Note that $\bigcirc\underline{FALSE}$ is an equivalent formula to **fin**.) Its use in *OUTOFSERVICE* above guarantees that the 'in-service' behaviour begins as soon as the service engineer has reset the system.

So the alarm is ringing unless somebody shuts it off and then the lift system is in a state of limbo unless it gets re-initialised (presumably by the service engineer). Of course, during the whole out-of-service period, the emergency button is illuminated.

To define the 'in-service' behaviour it is necessary to modify some of the previous behaviours. Before proceeding with those modifications, it would be rather annoying if the alarm started ringing for no apparent reason. Thus,

$$(\neg Al \wedge \neg LEm)\ \mathcal{W}\ \textbf{fin} \tag{23}$$

but, of course, remember that this 'always' is only applying during the 'in-service' behaviour. Then, to ensure that the service stops as soon as the emergency button is pressed, the following must hold.

$$\Box(Em \Rightarrow \textbf{fin}) \tag{24}$$

The door control specification which made promises about future openings and closings obviously needs modification. Such behaviour must now be interruptable by the emergency.

$$\begin{aligned}\bigwedge_i (v\langle \xi_1, \xi_2\rangle . \langle\ & D_i = \underline{CLOSED}\ \mathcal{W}^+\ (\xi_2 \vee Em),\\ & D_i = \underline{OPENING}\ \mathcal{U}^+\\ & ((D_i = \underline{OPEN}\ \mathcal{U}^+\\ & ((D_i = \underline{CLOSING}\ \mathcal{U}^+(\xi_1 \vee \xi_2 \vee Em))\\ & \vee Em))\\ & \vee Em)\end{aligned} \tag{25}$$

The control mechanism has been weakened such that should the emergency button be pushed then the doors stop whatever they are doing. Similarly, the promise to open the door when a foot gets in the way must be weakened.

$$\bigwedge_i \Box(J_i \Rightarrow (D_i = \underline{CLOSING}\ \mathcal{U}^+\ (D_i = \underline{OPENING} \vee Em))) \tag{26}$$

Clearly, the requirement on eventual service of lift request is also in need of repair.

$$\bigwedge_i \Box((C_i \vee S_i) \Rightarrow \Diamond(\textbf{service at } i \vee \textbf{service blocked})) \tag{27}$$

where

$$\textbf{service blocked} \stackrel{\text{def}}{=} \bigvee_j \Box(D_j \in OpenDoors) \vee Em$$

Finally, the environment actions should be modified to include the possible actions on *Em*, and the component actions modified to include the alarm mechanism *Al*. Such modifications are assumed as (18)′ and (19)′. The desired behaviour is therefore given as below.

$$\begin{aligned}INSERVICE \stackrel{\text{def}}{=}\ & (15) \wedge (18)' \wedge (19)' \wedge (14) \wedge\\ & (6) \wedge\\ & (23) \wedge (24) \wedge\\ & (25) \wedge (7) \wedge\\ & (26) \wedge\\ & (27) \wedge (22) \wedge (10) \wedge (11)\end{aligned}$$

4.3 Two call buttons per floor

In this section the specification is further modified to reflect the original requirements on lift call buttons and their associated lights. Obviously, the vector variables for the call button and its light, *C* and *LC*, should be replaced by a pair, one for up and one for down.

$$CU, LU, CD, LD : \textbf{array}(1\,..\,m)\ \textbf{of}\ Boolean$$

Similar to the requirements captured by the formula (6), the following is obtained.

$$\begin{aligned}\Box(\ & \bigwedge_{i \neq m} ((CU_i \Rightarrow (LU_i\ \mathcal{W}\ \textbf{up service at } i))\\ & \wedge (\textbf{up service at } i \Rightarrow (\neg LU_i \mathcal{W}^+\ CU_i)))\\ \wedge & \bigwedge_{i \neq 1} ((CD_i \Rightarrow (LD_i\ \mathcal{W}\ \textbf{down service at } i))\\ & \wedge (\textbf{down service at } i \Rightarrow (\neg LD_i\ \mathcal{W}^+\ CD_i)))\\ \wedge & \bigwedge_i ((S_i \Rightarrow (LS_i \mathcal{W}\ \textbf{service at } i))\\ & \wedge (\textbf{service at } i \Rightarrow (\neg LS_i\ \mathcal{W}^+\ S_i))))\end{aligned} \tag{28}$$

where

$$\begin{aligned}\textbf{up service at } i \stackrel{\text{def}}{=}\ & \textbf{service at } i \wedge \textbf{up required at } i \wedge\\ & (\textbf{at } i \textbf{ from above} \Rightarrow\\ & (\neg(\textbf{servicing down at } i) \wedge\\ & \neg \bigvee_{k<i} (\textbf{request } k \textbf{ is unserviced})))\end{aligned}$$

$$\begin{aligned}\textbf{down service at } i \stackrel{\text{def}}{=}\ & \textbf{service at } i \wedge \textbf{down required at } i \wedge\\ & (\neg(\textbf{at } i \textbf{ from above}) \Rightarrow\\ & (\neg(\textbf{servicing up at } i) \wedge\\ & \neg \bigvee_{k>i} (\textbf{request } k \textbf{ is unserviced})))\end{aligned}$$

$$\begin{aligned}\textbf{up required at } i \stackrel{\text{def}}{=}\ & D_i \in OpenDoors\ \mathcal{S}^+\\ & (D_i = \underline{CLOSED} \wedge LU_i \vee CU_i)\end{aligned}$$

$$\begin{aligned}\textbf{down required at } i \stackrel{\text{def}}{=}\ & D_i \in OpenDoors\ \mathcal{S}^+\\ & (D_i = \underline{CLOSED} \wedge LD_i \vee CD_i)\end{aligned}$$

$$\begin{aligned}\textbf{required at } i \stackrel{\text{def}}{=}\ & D_i \in OpenDoors\ \mathcal{S}^+\\ & (D_i = \underline{CLOSED} \wedge LS_i)\end{aligned}$$

$$\begin{aligned}\textbf{servicing up at } i \stackrel{\text{def}}{=}\ & D_i \in OpenDoors\ \mathcal{S}^+\\ & (LU_i \wedge \bigcirc\neg LU_i \vee CU_i \wedge \neg LU_i)\end{aligned}$$

servicing down at $i \stackrel{\text{def}}{=} D_i \in OpenDoors\ \mathscr{S}^+ (LD_i \wedge \bigcirc\neg LD_i \vee CD_i \wedge \neg LD_i)$

at i **from above** $\stackrel{\text{def}}{=} \bigvee_{j>i}$ (**at** i **from** j)

request k **is unserviced** $\stackrel{\text{def}}{=} (LU_k\ \mathscr{S}\ CU_k) \vee (LD_k\ \mathscr{S}\ CD_k) \vee (LS_k\ \mathscr{S}\ S_k)$

The formulation of **up service at** i (**down service at** i) appears quite tricky, and some justification is clearly warranted. The lift will be providing an up (down) service if the following conditions are met. The lift must be at the floor. There must have been a request to go up (down). If it is the case that the lift came from above (not above) in the down (up) direction then it must not be the case that the lift is already servicing a down (up) request or the lift has a request to continue in the same direction. For, if the latter had been the case, the lift should have possibly (because there may have been some other reason for stopping) gone straight to that lower (higher) floor.

Although this appears satisfactory there are circumstances when a lift, quite correctly, travels down (up) to a floor, services an up (down) request, but then continues its travel downwards (upwards). Imagine the following situation. The lift arrives at a floor from above with no external request but satisfying an internal request. A small boy presses the up button. It does not get illuminated because the lift can service the request. However, before the lift leaves he presses the down button, as is the wont of small boys, and lo and behold the button does not get illuminated, because of the lift's overall desire not to change direction. Note that if he had pressed the down button and then the up button, the down button would not light but the up one would light up because the lift cannot change from its preferred direction of travel. Naturally, the small boy can confuse the system even further by then requesting, internally, that the lift should go up. (I don't know whether the up light should go then go off! This system leaves the light on so that when the lift comes back again the real passenger who wants to go up can cuff the small boy's ears. Actually, the real passenger should not be able to tell which way the lift went.)

In section 3.2.3 where servicing behaviour was defined, the notions of unserviced requests were formulated. These must be altered to reflect the new types of requests.

up request k **is unserviced** $\stackrel{\text{def}}{=}$

$$(LU_k\ \mathscr{S}\ CU_k) \vee (LS_k\ \mathscr{S}\ S_k)$$

down request k **is unserviced** $\stackrel{\text{def}}{=}$

$$(LD_k\ \mathscr{S}\ CD_k) \vee (LS_k\ \mathscr{S}\ S_k)$$

left i **with up request** $k \stackrel{\text{def}}{=}$

$$D_i \in OpenDoors\ \mathscr{U}^+ (D_i = \underline{CLOSED} \wedge \textbf{up request } k \textbf{ is unserviced})$$

left i **with down request** $k \stackrel{\text{def}}{=}$

$$D_i \in OpenDoors\ \mathscr{U}^+ (D_i = \underline{CLOSED} \wedge \textbf{down request } k \textbf{ is unserviced})$$

left i **with request** $k \stackrel{\text{def}}{=}$

$$D_i \in OpenDoors\ \mathscr{U}^+ (D_i = \underline{CLOSED} \wedge \textbf{request } k \textbf{ is unserviced})$$

Property (8) requires minor alteration to reflect the new types of request. Similarly, property (9) stating that some service occurs only if it had been requested must also change.

$$\bigwedge_i \Box((CU_i \vee CD_i \vee S_i) \Rightarrow \Diamond(\textbf{service at } i \vee \textbf{service blocked})) \quad (29)$$

$$\bigwedge_i (\textbf{service at } i \Rightarrow (\textbf{up service at } i \vee \textbf{down service at } i \vee \textbf{required at } i)) \quad (30)$$

The property (10), stating that the lift does not miss floors with appropriate requests, becomes as follows.

$$\bigwedge_i \bigwedge_j \Box(\textbf{from } i \textbf{ to } j \Rightarrow ((i<j \Rightarrow \neg \bigvee_k (\textbf{left } i \textbf{ with up request } k \wedge i<k<j)) \wedge (i>j \Rightarrow \neg \bigvee_k (\textbf{left } i \textbf{ with down request } k \wedge i>k>j)))) \quad (31)$$

Notice that it has been necessary to allow the lift to pass a floor when travelling up (respectively, down) which has an unserviced down (respectively, up) request. Also, notice that internal requests cannot be missed.

Property (11), characterising allowed changes of direction, must be restated using the latest definition of leaving a floor with an outstanding request.

$$\bigwedge_i \bigwedge_j \bigwedge_k \Box(\textbf{from } i \textbf{ via } j \textbf{ to } k \Rightarrow ((i<j>j \Rightarrow \neg \bigvee_l (\textbf{left } j \textbf{ with request } l \wedge i<j<l)) \wedge (i>j<k \Rightarrow \neg \bigvee_l (\textbf{left } j \textbf{ with request } l \wedge i>j>l)))) \quad (32)$$

5. THE MULTIPLE-LIFT SYSTEM

Building upon the specifications given in the previous two sections, it is relatively easy to produce the specification of the multiple-lift system, as originally proposed in the problem description (Section 2). The major modification is to extend the one-dimensional vector for the lift doors into two dimensions and thus cover several lifts. Remember, however, that a multiple-lift system is not just a replication of single lifts; there is only one pair of lift request buttons per floor. The first section below (5.1) presents a straightforward extension, without considering the emergency situation. The second section (5.2) considers the difficult problem of how to capture that servicing is in some sense optimal. Then in Section 5.3, emergencies and alarms are added (to the confusion). Finally, a review is made in Section 5.4.

5.1 Single to multiple lifts

5.1.1 *Interface*

D : **array**(1 . . n)**of array**(1 . . m)**of** *DoorState*
S, LS, J : **array**(1 . . n)**of array**(1 . . m)**of** *Boolean*
CU, CD, LU, LD : **array**(1 . . m)**of** *Boolean*

Now the state of the doors is contained in the two-dimensional vector D. The first index determines the particular lift shaft, the second index determines the floor. Similarly, the send buttons, their lights, and the jamming signal must be two-dimensional. The other buttons and their lights remain one-dimensional, as there is only one pair per floor.

5.1.2 *Button and light behaviour*

The previous property (28) is extended by the extra lift index. Additionally, the notions of up and down service have been slightly changed, and a lift can now provide an up service in the following situation. A lift arrives at a floor from above to service, say, an internal request. Although there may be external requests to floors below, provided there is no outstanding internal request that lift may service an up request at that floor; this is because when there are multiple lifts in service some other lift may service that lower-floor request. It will, of course, be necessary to place extra servicing constraints to ensure that the multiple-lift system behaves like the previous single-lift system when there is only one lift.

$$\bigwedge_{l\in 1..n} \Box\Big(\bigwedge_{i\neq m} ((CU_i \Rightarrow (LU_i \mathbin{\mathcal{W}} \textbf{up service at } i)) \wedge (\textbf{up service at } i \Rightarrow (\neg LU_i \mathbin{\mathcal{W}^+} CU_i)))$$
$$\wedge \bigwedge_{i\neq 1} ((CD_i \Rightarrow (LD_i \mathbin{\mathcal{W}} \textbf{down service at } i)) \wedge (\textbf{down service at } i \Rightarrow (\neg LD_i \mathbin{\mathcal{W}^+} CD_i)))$$
$$\wedge \bigwedge_{i} ((S_{li} \Rightarrow (LS_{li} \mathbin{\mathcal{W}} \textbf{service at } i \textbf{ by } l)) \wedge (\textbf{service at } i \textbf{ by } l \Rightarrow (\neg LS_{li} \mathbin{\mathcal{W}^+} S_{li})))) \quad (33)$$

where

service at i **by** $l \stackrel{\text{def}}{=}$

$$D_{li} = \underline{OPENING} \vee D_{li} = \underline{OPEN}$$

service at $i \stackrel{\text{def}}{=}$

$$\bigvee_{l\in 1..n} (\textbf{service at } i \textbf{ by } l)$$

up service at i **by** $l \stackrel{\text{def}}{=}$

service at i **by** $l \wedge$
up required at i **by** $l \wedge$
(at i **from above by** $l \Rightarrow$
$(\neg$**(servicing up at** i **by** $l) \wedge$
$\neg \bigvee_{k<i} (l$ **has request** $k)))$

up service at $i \stackrel{\text{def}}{=}$

$$\bigvee_{l\in 1..n} (\textbf{up service at } i \textbf{ by } l)$$

down service at i **by** $l \stackrel{\text{def}}{=}$

service at i **by** $l \wedge$
down required at i **by** $l \wedge$
$(\neg$**(at** i **from above by** $l) \Rightarrow$
$(\neg$**(servicing up at** i **by** $l) \wedge$
$\neg \bigvee_{k>i} (l$ **has request** $k)))$

down service at $i \stackrel{\text{def}}{=}$

$$\bigvee_{l\in 1..n} (\textbf{down service at } i \textbf{ by } l)$$

up required at i **by** $l \stackrel{\text{def}}{=}$

$$D_{li} \in OpenDoors \mathbin{\mathcal{S}} (D_{li} = \underline{CLOSED} \wedge LU_i \vee CU_i)$$

down required at i **by** $l \stackrel{\text{def}}{=}$

$$D_{li} \in OpenDoors \mathbin{\mathcal{S}} (D_{li} = \underline{CLOSED} \wedge LD_i \vee CD_i)$$

required at i **by** $l \stackrel{\text{def}}{=}$

$$D_{li} \in OpenDoors \mathbin{\mathcal{S}} (D_{li} = \underline{CLOSED} \wedge LS_{li})$$

servicing up at i **by** $l \stackrel{\text{def}}{=}$

$$D_{li} \in OpenDoors \mathbin{\mathcal{S}} (LU_i \wedge \bigcirc \neg LU_i \vee CU_i \wedge \neg LU_i)$$

servicing down at i **by** $l \stackrel{\text{def}}{=}$

$$D_{li} \in OpenDoors \mathbin{\mathcal{S}} (LD_i \wedge \bigcirc \neg LD_i \vee CD_i \wedge \neg LD_i)$$

at i **from above by** $l \stackrel{\text{def}}{=}$

$$\bigvee_{j>i} (\textbf{at } i \textbf{ from } j \textbf{ by } l)$$

at i **from** j **by** $l \stackrel{\text{def}}{=}$

$$(D_{li} \in OpenDoors \mathbin{\mathcal{S}^+} (\bigwedge_{k} (D_{lk} = \underline{CLOSED}) \mathbin{\mathcal{S}} D_{lj} \in OpenDoors))$$

l **has request** $k \stackrel{\text{def}}{=}$ $(LS_{lk} \mathbin{\mathcal{S}} S_{lk})$

5.1.3 *Door behaviour*

Similar to the above section, the previous door control behaviour is extended by the extra lift index.

$$\bigwedge_{l}\bigwedge_{i} (\nu\langle\xi_1, \xi_2\rangle . \langle D_{li} = \underline{CLOSED} \mathbin{\mathcal{W}^+} \xi_2, D_{li} = \underline{OPENING} \mathbin{\mathcal{U}^+} (D_{li} = \underline{OPEN} \mathbin{\mathcal{U}^+} (D_{li} = \underline{CLOSING} \mathbin{\mathcal{U}^+} (\xi_1 \vee \xi_2))))\rangle)_1 \quad (34)$$

$$\bigwedge_{l} \Box (\sum_{i} D_{li} \in OpenDoors) \leqslant 1 \quad (35)$$

$$\bigwedge_{l}\bigwedge_{i} \Box (J_{li} \Rightarrow (D_{li} = \underline{CLOSING} \mathbin{\mathcal{U}^+} D_{li} = \underline{OPENING}) \quad (36)$$

5.1.4 *Servicing behaviour*

Because of the additional lift index, internal requests must be serviced by their particular lift, unlike external calls.

$$\bigwedge_l \bigwedge_i \Box((CU_i \vee CD_i) \Rightarrow \Diamond(\textbf{service at } i \vee \bigwedge_l (l \textbf{ service blocked})) \wedge S_{li} \Rightarrow \Diamond(\textbf{service at } i \textbf{ by } l \vee l \textbf{ service blocked})) \quad (37)$$

where

$$l \textbf{ service blocked} \stackrel{\text{def}}{=} \bigvee_i \Box(D_{li} \in OpenDoors)$$

$$\bigwedge_l \bigwedge_i \Box(\textbf{service at } i \textbf{ by } l \Rightarrow (\textbf{up service at } i \textbf{ by } l \vee \textbf{down service at } i \textbf{ by } l \vee \textbf{required at } i \textbf{ by } l)) \quad (38)$$

$$\bigwedge_i \Box(\textbf{service at } i \Rightarrow (\textbf{up service at } i \vee \textbf{down service at } i \vee \textbf{many required at } i) \wedge \quad (39)$$

$$\textbf{up service at } i \Rightarrow \textbf{one up service at } i \wedge$$

$$\textbf{down service at } i \Rightarrow \textbf{one down service at } i)$$

where

$$\textbf{one up service at } i \stackrel{\text{def}}{=} \bigvee_l (\textbf{up service at } i \textbf{ by } l \wedge \bigwedge_{l' \neq l} (\neg \textbf{up service at } i \textbf{ by } l'))$$

$$\textbf{one down service at } i \stackrel{\text{def}}{=} \bigvee_l (\textbf{down service at } i \textbf{ by } l \wedge \bigwedge_{l' \neq l} (\neg \textbf{down service at } i \textbf{ by } l'))$$

$$\textbf{many required at } i \stackrel{\text{def}}{=} \bigvee_l (\textbf{required at } i \textbf{ by } l)$$

This latter property, (39), is necessary to avoid several lifts servicing the same request. Notice that when there is only one lift in the multiple system, the formula states no more than (38). Note that (38), however, is still required for the multiple system.

$$\bigwedge_l \bigwedge_i \bigwedge_j \Box(\textbf{from } i \textbf{ to } j \textbf{ by } l \Rightarrow ((i < j \Rightarrow \neg \bigvee_k (i < k < j \wedge (l \textbf{ left with request } k \vee (l \textbf{ left } i \textbf{ with up request } k \wedge \neg l \textbf{ from } i \textbf{ to } j \textbf{ with up request } k)))) \quad (40)$$

$$\wedge (i > j \Rightarrow \neg \bigvee_k (i > k > j \wedge (l \textbf{ left with request } k \vee (l \textbf{ left } i \textbf{ with down request } k \wedge \neg l \textbf{ from } i \textbf{ to } j \textbf{ with down request } k))))))$$

where

$$\textbf{from } i \textbf{ to } j \textbf{ by } l \stackrel{\text{def}}{=} D_{li} \in OpenDoors \; \mathcal{U}^+ \; (\bigwedge_k (D_{lk} = \underline{CLOSED}) \; \mathcal{U} \; D_{lj} \in OpenDoors)$$

$$l \textbf{ left } i \textbf{ with request } k \stackrel{\text{def}}{=} D_{li} \in OpenDoors \; \mathcal{U}^+ \; (D_{li} = \underline{CLOSED} \wedge (LS_{lk} \; \mathcal{S} \; S_{lk}))$$

$$l \textbf{ left } i \textbf{ with up request } k \stackrel{\text{def}}{=} D_{li} \in OpenDoors \; \mathcal{U}^+ \; (D_{li} = \underline{CLOSED} \wedge (LU_k \; \mathcal{S} \; CU_k))$$

$$l \textbf{ left } i \textbf{ with down request } k \stackrel{\text{def}}{=} D_{li} \in OpenDoors \; \mathcal{U}^+ \; (D_{li} = \underline{CLOSED} \wedge (LD_k \; \mathcal{S} \; CD_k))$$

$$l \textbf{ from } i \textbf{ to } j \textbf{ with up request } k \stackrel{\text{def}}{=} (D_{li} \in OpenDoors \wedge LU_k) \; \mathcal{U}^+ \; ((\bigwedge_d (D_{ld} = \underline{CLOSED}) \wedge LU_k) \; \mathcal{U} \; (D_{lj} \in OpenDoors \wedge LU_k))$$

$$l \textbf{ from } i \textbf{ to } j \textbf{ with down request } k \stackrel{\text{def}}{=} (D_{li} \in OpenDoors \wedge LD_k) \; \mathcal{U}^+ \; ((\bigwedge_d (D_{ld} = \underline{CLOSED}) \wedge LD_k) \; \mathcal{U} \; (D_{lj} \in OpenDoors \wedge LD_k))$$

This property has been modified to allow a lift to skip servicing a floor provided that some other lift will have serviced the outstanding request before the first lift stops at another floor. Again, note that when there is only one lift in the system, the behaviour characterised is as before.

$$\bigwedge_l \bigwedge_i \bigwedge_j \bigwedge_k \Box(\textbf{from } i \textbf{ via } j \textbf{ to } k \textbf{ by } l \Rightarrow ((i < j > k \Rightarrow \neg \bigvee_r (i < j < r \wedge (l \textbf{ left } j \textbf{ with request } r \vee (l \textbf{ left } j \textbf{ with up request } r \wedge \neg l \textbf{ from } j \textbf{ to } k \textbf{ with up request } t)))) \quad (41)$$

$$\wedge (i > j < k \Rightarrow \neg \bigvee_r (i > j > r \wedge (l \textbf{ left } j \textbf{ with request } r \vee (l \textbf{ left } j \textbf{ with down request } r \wedge \neg l \textbf{ from } j \textbf{ to } k \textbf{ with down request } r))))))$$

where

$$\textbf{from } i \textbf{ via } j \textbf{ to } k \textbf{ by } l \stackrel{\text{def}}{=} \textbf{at } j \textbf{ from } i \textbf{ by } l \wedge \textbf{from } j \textbf{ to } k \textbf{ by } l$$

In the above reformulation, extensions similar to those in (40) have been made. A lift is allowed to ignore a request whose service would cause the lift to continue travelling in the direction provided some other lift has that outstanding request serviced before the first lift stops again.

5.1.5 *Initial conditions*

It is now required to ensure that the extra variables are initialised.

$$\bigwedge_l \bigwedge_i (D_{li} = \underline{CLOSED} \wedge \neg S_{li} \wedge \neg LS_{li} \wedge \neg J_{li} \wedge \neg CU_i \wedge \neg LU_i \wedge \neg CD_i \wedge \neg LD_i) \quad (42)$$

5.2 Optimal servicing?

Here questions of optimality are considered. A number of questions about the above lift system come to mind. In particular the following two seem rather relevant.

- What forces all the lifts to be active?
- If all the lifts are stationary at, say, different floors, does the nearest lift service a call?

The immediate answer to the first question is 'nothing'. To the second comes the reply 'not necessarily'. Of course it would be good to enforce these kinds of optimality.

As a first attempt at enforcing activity a sort of fairness constraint could be applied. For example,

$$\bigwedge_l \bigwedge_i (\Box\Diamond CU_i \Rightarrow \Box\Diamond(LU_i \wedge \neg\bigcirc LU_i \wedge \textbf{service at } i \textbf{ by } l) \wedge \Box\Diamond CD_i \Rightarrow \Box\Diamond(LD_i \wedge \neg\bigcirc LD_i \wedge \textbf{service at } i \textbf{ by } l))$$

So if there are infinitely many requests to go up there will be infinitely many services by any particular lift. Similarly for calls to go downwards. However, this is rather weak in that a lift may be particularly ideal and refuse to work for a very long time. Can a stronger statement be formulated?

The question of getting the nearest lift to service a request is a rather difficult problem. The major difficulty is that the position of a lift is only known when it is servicing a request. If the lift is between floors, then that is all that is known. To confound the issue, because the doors of a lift must be closed when it is stationary (i.e. not required) its actual position must also be unknown. All that is known is where it appeared last!

As a first attempt at getting some degree of optimality consider the following. If a lift arrives at a floor to service an up or down request, it ought not to be the case that there is another lift which is both nearer and has remained stationary since the time when the first lift left its previous floor.

$$\bigwedge_l \bigwedge_i \bigwedge_j \Box((\textbf{at } i \textbf{ from } j \textbf{ by } l \wedge \neg i \textbf{ requested by } l) \Rightarrow \neg \bigvee_{a \neq l} (\bigvee_k (|j-k| < |j-i| \wedge (a \textbf{ stationary at } k \mathcal{S} \textbf{ service at } i \textbf{ by } l)) \vee (j < |j-i| \wedge a \textbf{ stationary at ground})))$$

where

$$i \textbf{ requested by } l \stackrel{\text{def}}{=} D_{li} \in OpenDoors \mathcal{S} LS_{li}$$

The problem that remains now is how to characterise 'stationary'. One possible way has a lift stationary at a floor when it is seen at that floor and then never seen again. However, this is rather a weak notion. Perhaps a better approach has a lift stati ıary at a floor when it is last seen at that floor and has no outstanding internal requests to be serviced. Thus,

$$l \textbf{ stationary at } k \stackrel{\text{def}}{=} \bigwedge_i (D_{li} = \underline{CLOSED} \wedge \neg LS_{li}) \mathcal{S} D_{lk} \in OpenDoors$$

$$l \textbf{ stationary at ground} \stackrel{\text{def}}{=} \bigwedge_i (D_{li} = \underline{CLOSED} \wedge \neg LS_{li}) \mathcal{S} \textbf{ beg}$$

Is the above the best that can be obtained? There is still a considerable flaw. The above is all right if the lift is covering large distances in single hops, but the lift which is making small journeys will usually appear to be doing the right thing. A better, but much more complicated solution can be obtained by summing the total distance travelled by a lift from the instance of a call to its servicing of that call (remember there may be several floors visited). It then ought not to be the case that there is a stationary (since the call) lift nearer than the total distance covered by the servicing lift. First, a predicate to determine whether the lift has covered a shorter distance.

$$s \textbf{ to } d \textbf{ by } l \textbf{ less than } r \stackrel{\text{def}}{=} (\nu\xi \,.\, \chi(\xi))(s, d, r)$$

where

$$\chi \stackrel{\text{def}}{=} \bigwedge_{i \neq s} (\textbf{at } d \textbf{ from } i \textbf{ by } l \Rightarrow \textbf{at } d \textbf{ from } i \textbf{ with } \xi(s, i, r-|d-i|) \textbf{ by } l) \wedge (\textbf{at } d \textbf{ from } s \textbf{ by } l \Rightarrow r > |d-s|)$$

and

$$\textbf{at } i \textbf{ from } j \textbf{ with } \phi \textbf{ by } l \stackrel{\text{def}}{=} D_{li} \in OpenDoors \mathcal{S}^+ (\bigwedge_k (D_{lk} = \underline{CLOSED}) \mathcal{S} (D_{lj} \in OpenDoors \wedge \phi))$$

This predicate recursively retraces the route taken by the lift to travel from floor s to floor d subtracting the intermediate distances from the supplied bound r. If the final bound is greater than the last hop then the route was indeed shorter!

Such a predicate can then be incorporated into the property (33) given above. Its inclusion is sketched below.

$$\Box \bigwedge_i (CU_i \Rightarrow (LU_i \mathcal{W} \bigvee_l (\textbf{up service at } i \textbf{ by } l \wedge \bigwedge_{a \neq l} \bigwedge_k \bigwedge_j (((a \textbf{ stationary at } k \vee a \textbf{ stationary at ground} \wedge k = 1) \mathcal{S} (\neg LU_i \wedge l \textbf{ next at } j)) \Rightarrow j \textbf{ to } i \textbf{ by } l \textbf{ less than } |k-i|))))$$

where

$$l \textbf{ next at } j \stackrel{\text{def}}{=} \bigvee_i (D_{li} \in OpenDoors) \mathcal{U}^+ ((\bigwedge_i D_{li} = \underline{CLOSED}) \mathcal{U} D_{lj} \in OpenDoors)$$

So, if a lift arrives at floor i to service an up request then it had better be the case that the journey taken was shorter than a journey by any other stationary lift. Of course this does not enforce nearest servicing if all lifts are active; however, it does help in keeping lifts active!

5.3 Emergencies and all that

To add the emergency button to each of the lifts requires just minor changes to the above properties essentially as before. Some care, however, is required to fit it all together. First, the emergency buttons, lights and alarms must be given.

$$Em, LEm, Al : \textbf{array}(1\,..\,n)\textbf{of}\ Boolean$$

As in Section 4.2, an 'in service' and 'out of service' behaviour is described, but one for each lift. Overall service is given by the following.

$$\bigwedge_{l \in 1..n} (\nu\xi\,.(INSERVICE_l \mathscr{C} OUTOFSERVICE_l \mathscr{C} \xi))$$

with

$$OUTOFSERVICE_l \stackrel{\text{def}}{=} (Al_l \wedge LEm_l)\ \mathscr{W}^+ \\ (LEm_l\ \mathscr{W}\ (\mathbf{init}_l \wedge \mathbf{fin}))$$

So, the alarm rings unless it gets switched off and anything happens (to lift l only) unless lift l is initialised. During the whole of the out-of-service period, the emergency button light is on!

Rather than give the modification for all the in-service properties, only a couple of changes are given explicitly, together with general comments about the required changes. First note that all properties given in Section 5.2 above were, essentially, quantified over the lifts. It has been necessary to remove this quantification to an outer level, i.e. to the overall service (as immediately above). Then, as in Section 4.2, the promises about future behaviour must be carefully guarded in case an emergency button gets depressed. Consider the property (33) covering the call buttons, lights and services. Because of the way **up service at** i has been written, i.e. as a disjunction over the possible lifts, it is necessary to stipulate that only 'in service' lifts should be considered. Thus $\neg LEm_l$, of course indicating that the emergency button has not been pushed, is added to the definitions of up (and down) service.

up service at i **by** $l \stackrel{\text{def}}{=}$

$\neg LEm_l \wedge$

service at i **by** $l \wedge$

up required at i **by** $l \wedge$

(**at** i **from above by** $l \Rightarrow$

(¬(**servicing down at** i **by** l) ∧

$\neg \bigvee_{k \leq i}$ (l **has request** k)))

A similar change is made to the blocked service predicate used in the property (37) describing eventual service of a call.

$$l\ \textbf{service blocked} \stackrel{\text{def}}{=} (\bigvee_i \Box\,(D_{li} \in OpenDoors)) \vee Em_l$$

Other obvious and related changes are made to the previous descriptions, and the following in-service behaviour description results. Notation for naming the properties has been abused, for example $(33)_l$ refers to the (33) property without quantification over lifts.

$INSERVICE_l \stackrel{\text{def}}{=}$

$(42)_l \wedge (18)'_l \wedge (19)'_l \wedge (14) \wedge$	*initialisation, etc.*
$(23)'_l \wedge (24)'_l \wedge$	*alarms, etc.*
$(33)_l \wedge$	*call up, down lights*
$(34)_l \wedge (35)_l \wedge (36)_l \wedge$	*door control*
$(37)_l \wedge (38)_l \wedge (39)_l \wedge$	*servicing constraints*
$(40)_l \wedge (41)_l$	

5.4 Has the desired system been specified?

Here a comparison between informal requirements and formal specification is made in an attempt to determine whether the formally specified multiple-lift system is the right one. The requirements as written down in Section 2 are considered in turn.

(1, 2) Yes. Each lift has the appropriate set of buttons. They illuminate in the manner required (property 33 specifies that behaviour). In particular, only one of the call up / down buttons is cancelled by a lift service. This is because a call up is only cancelled by a lift providing an up service and, of course, up and down services are mutually exclusive (see Section 4.3).

(3) Yes. The door control behaviour (34) is such that the doors will always eventually close unless prevented by some obstruction. The doors must then remain closed until there is some request (37, 38, 39).

(4) Yes. The first servicing behaviour property (37) guarantees eventual service. All floors must be serviced eventually, and the servicing is such that lifts are essentially not allowed to pass by requests to stop (40, 41). In that sense all floors have equal priority.

(5) Yes. Internal requests are serviced sequentially (40, 41) as far as possible. The proviso is simply because the position of a lift at the time a request is made may not be known, i.e. it may be between floors.

(6) Yes. Emergency buttons are provided and behave in the desired manner.

So it appears the system described informally has been specified; each of the informal requirements has been enshrined by some formal statement. However, it may well be the case that the informal requirements included some subtle contradiction, and therefore such a contradiction would be present in the formal specification. In the temporal framework, an inconsistency or contradiction in the specification means that there is no model which satisfies the specification; in other words there can be no implementation which fits the specification. If such an inconsistency were not found at this initial stage of specification there would clearly be much wasted effort in attempting to develop an impossible solution.

The important question is therefore 'can inconsistencies in the formal specification be easily detected?'. If the specification has been written using a decidable temporal language, such as the propositional temporal logic used in this paper, consistency can be mechanically checked,

albeit rather slowly. Ref. 12 provides examples of decision procedures for some temporal languages. For undecidable languages, for example first-order temporal logics, inconsistencies can only be determined, in general, by deducing false from the formal specification in a proof-theoretic sense (more or less by hand).

6. COMMENTS

The temporal specification of the multiple-lift system presented in the previous sections has been developed by a process of extension to a rather naïve and basic single-lift system. This, I claim, is a natural approach to the development of specifications of complex systems. In the lift example, the given informal requirements were studied and then abstracted to a much simpler set which could then be formalised without too many iterations. The abstract interface (single buttons, associated lights and two-state doors) was then gradually extended to include more observable aspects of the real system; for example, two-state doors got extended to four-state doors to allow for interruption (feet in the way when closing). This extension process is not a refinement of the interface; new observable parts were added to the interface which were not previously present in some abstract form, and hence the normal refinement proof obligations (containment of behaviours) do not apply. However, some consistency checks are possible. Consider the first extension made, i.e. two-state doors to four-state doors together with the door jam signal. The behaviour of the extended system when run in an environment which never jams the door should be exactly that of the original naïve system. Such a consistency check is straightforward in our temporal framework here. Letting *LIFTSPEC* and *LIFTSPEC'* denote the original and extended specifications respectively, one must show that

$$LIFTSPEC' \wedge \Box \bigwedge_i \neg J_i \Rightarrow LIFTSPEC.$$

Naturally, this is left as an exercise to the interested reader! However, not all the extensions made have such simple consistency checks; the change of interface from one call button per floor to two buttons per floor is such a case.

In general, documenting the formalisation of a set of informal requirements, as above, is both worthwhile and necessary. But perhaps it would be more useful to ensure that the many fruitless avenues explored are also documented. Such a task is extremely difficult to follow and one that certainly didn't occur during the construction of the lift specification presented here, although it should have. Worthy of note, however, are the following mistakes which I made during early attempts at this formal specification: (1) attempting to work with two complex an observable interface too early in the development process; (2) not abstracting to a high enough level initially.

The resultant temporal specification is, in my mind, a useful 'top-level' abstract specification which could be used in the development of some real-world lift control system. An interesting exercise now is to propose temporal specifications of various (parallel) sub-components and prove that when appropriately composed they behave as originally specified, i.e. as a multiple-lift system. Such a decomposition might be considered as an actual lift controller and lift motor systems. Appropriate techniques for composing such temporal specifications have been presented in Refs 6 and 7.

The temporal specification style chosen is just one of many possible techniques, see also for example Refs 13–15, 17. My choice in approach is essentially governed by the desire to have a style which (automatically) avoids over-specification (or implementation bias) combined with naturality. It is our belief that although temporal formulae do appear strange and difficult to understand initially, there is usually a most natural correspondence with the 'informal' natural-language requirements. For example, compare the natural-language translation of constraint (10) with the actual requirement. It should be noted that such a close correspondence between temporal formalism and natural language was put to excellent use in a book where informal arguments were obtained directly from formal temporal proofs.[16]

7. ACKNOWLEDGEMENTS

I am most grateful to Graham Gough for having painstakingly read through and pointed out several mistakes in the previous version of this document, to Ruurd Kuiper for hearing my worries about lift behaviours, to the attendees at a recent joint Ada-UK/Ada-Europe working group meeting for many useful comments raised during my presentation of the lift specification, and last but not least to all my other colleagues who have listened patiently to my witterings about lifts.

This work was partially supported by SERC grant GR/C/05760.

REFERENCES

1. L. Lamport, What good is temporal logic? *Proceedings of IFIP '83*, pp. 657–668. North-Holland, Amsterdam (1983).
2. Z. Manna, Logic of programs. *Proceedings of IFIP '80*, pp. 41–51. North-Holland, Amsterdam (1980).
3. S. S. Owicki and L. Lamport, Proving liveness properties of concurrent programs. *ACM TOPLAS* **4**, (3), 455–495 (1982).
4. A. Pnueli, The temporal logic of programs. *Proc. 18th Symposium on Foundations of Computer Science, Providence, RI*, pp. 46–7 (1977).
5. Z. Manna and A. Pnueli, Verification of concurrent programs: the temporal framework. In *The Correctness Problem in Computer Science*, edited R. S. Boyer and J. S. Moore, pp. 215–273. International Lecture Series in Computer Science, Academic Press, London (1982).
6. H. Barringer, R. Kuiper and A. Pnueli, Now you may compose temporal logic specifications. *Proceedings of the 16th ACM Symposium on the Theory of Computing, Washington*, pp. 51–63 (1984).
7. H. Barringer, R. Kuiper and A. Pnueli, A compositional approach to a CSP-like language. *Proceedings of the IFIP Working Conference 'The Role of Abstract Models in Information Processing'*. Vienna (1985).
8. H. Barringer, R. Kuiper and A. Pnueli, The Temporal Mu-Calculus, in preparation (1986).
9. P. Wolper, Temporal logic can be more expressive.

Proceedings of the 22nd Symposium on Foundations of Computer Science, pp. 340–347 (1981).
10. H. Barringer and R. Kuiper, Hierarchical development of concurrent systems in a temporal logic framework. *Proceedings of the NSF/SERC Seminar on Concurrency, CMU, Pittsburgh* (July 1984).
11. B. Moszkowski, Executing temporal logic programs. *Proceedings of the NSF/SERC Seminar on Concurrency, CMU, Pittsburgh* (1984); *also Technical Report no. 55*, Computer Laboratory, University of Cambridge.
12. G. Gough, Decision procedures for temporal logic, *M.Sc. Dissertation*, University of Manchester (1984).
13. B. Hailpern and S. S. Owicki, Modular verification of computer communication protocols. *IEEE Trans. on Communications COM-31*, 1, pp. 56–68 (Jan. 1983).
14. L. Lamport, Specifying concurrent program modules, *ACM TOPLAS*, **5** (2) (April 1983), pp. 190–222.
15. R. Schwartz and P. Melliar-Smith, From state machines to temporal logic: specification methods for protocol standards. *IEEE Trans. on Communications* (Dec. 1982).
16. M. Ben-Ari, *Principles of Concurrent Programming*. Prentice-Hall, Englewood Cliffs, N.J. (1982).
17. R. Schwartz, P. M. Melliar-Smith and F. Vogt, *An Interval Logic for Higher-level Temporal Reasoning: Language Definition and Examples*. Technical Report CSL-138, SRI International (1983).

APPENDIX A. THE TEMPORAL LANGUAGE

A.1 Basic alphabet

The basic symbols of the temporal language are divided into two groups:

(1) local symbols, i.e. symbols whose values are state- or transition-dependent

$P \in \mathbf{P}$ state propositions
$y \in \mathbf{Y}$ state variables
$l \in \mathbf{L}$ transition variables, for example E and Π;

(2) global symbols, i.e. symbols whose values are fixed for the complete sequence

$x \in \mathbf{X}$ global variables
$f \in \mathbf{F}$ function symbols
$q \in \mathbf{Q}$ predicate symbols.

Terms are constructed in the usual way from state, transition and global variables, or by the application of appropriate function symbols to terms. Atomic formulae can then be built from state propositions or by the application of predicates to terms.

The logical constants are the standard truth constants,

$$\underline{TRUE}, \underline{FALSE}$$

the standard first-order logical operators,

$$\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \forall, \exists$$

the unary future temporal operators,

$\oplus$ (strong next time), $\bigcirc$ (weak next time),
$\Diamond$ (eventually), $\Box$ (always)

the binary future temporal operators,

$\mathcal{U}$ (strong until), $\mathcal{W}$ (weak until),
$\mathcal{U}^+$ (strict until), $\mathcal{W}^+$ (strict unless),
$\mathscr{C}$ (combine), $\mathscr{C}^*$ (iterated combine)

the unary past-time temporal operators,

$\ominus$ (strong previous), $\odot$ (weak previous),
$\diamondsuit$ (some time previously), $\boxdot$ (always previously)

the binary past-time temporal operators,

$\mathcal{S}$ (strong since), $\mathcal{Z}$ (weak since),
$\mathcal{S}^+$ (strict since), $\mathcal{Z}^+$ (strict weak since)

the maximal and minimal fixed-point constructors ν and μ.

Temporal formulae are then constructed from atomic formulae, temporal variables, or by appropriate application of logical operators or fixed-point constructors. A well-formed temporal formula has no free temporal variables, i.e. it must be closed somewhere by a ν or μ form.

A.2 Models and interpretations

Assuming a fixed domain, D, and fixed interpretations for the function and predicate symbols, a model $\mathscr{M}$ over which a temporal formula is interpreted is a 5-tuple,

$$\mathscr{M} = \langle \alpha, \sigma, n, I, J \rangle,$$

where

α assigns D-values to the global variables,
σ is a non-empty finite or infinite sequence of states and transitions

$$\sigma \overset{\text{def}}{=} s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \xrightarrow{t_3} \dots$$

n is the index into σ which gives the current state,
I is a state interpretation assigning D-values to each state variable and truth values **tt**, **ff** to state propositions,
J is a transition interpretation assigning D-values to each transition variable.

Given a model $\mathscr{M} = (\alpha, \sigma, n, I, J)$ it is possible to define, inductively, the interpretation of temporal formulae over $\mathscr{M}$. In general, this interpretation only involves change to the sequence σ and the index n and, therefore, in the following $\phi|^n_\sigma$ ($t|^n_\sigma$) abbreviates the value of the formula ϕ (term t) over the model $\mathscr{M}$.

The sequence operator o (fusion) will be required in the definition of the $\mathscr{C}$ and $\mathscr{C}^*$, and is defined below.

$$\sigma_1 \circ \sigma_2 \overset{\text{def}}{=} \begin{array}{ll} \text{if} & \sigma_1 = s_0 \to s_1 \to \dots s_k, \quad \text{and} \\ & \sigma_2 = s_k \to s_{k+1} \to \dots \\ \text{then} & s_0 \to s_1 \to \dots s_k \to s_{k+1} \to \dots \\ \text{otherwise} & \sigma_1. \end{array}$$

Terms

$x|^n_\sigma \overset{\text{def}}{=} \alpha(x)$ global variables

$y|^n_\sigma \overset{\text{def}}{=} I(s_n, y)$ state variables

$l|^n_\sigma \overset{\text{def}}{=} J(t_n, l)$ transition variables

$f(t_1, ., t_k)|^n_\sigma \overset{\text{def}}{=} F_f(t_1|^n_\sigma, ., t_k|^n_\sigma)$ function applications

Atomic formulae

$P|^n_\sigma \stackrel{\text{def}}{=} I(s_n, P)$ state propositions

$q(t_1, .., t_k)|^n_\sigma \stackrel{\text{def}}{=} Q_q(t_1|^n_\sigma, .., t_k|^n_\sigma)$ predicate applications

Now assuming the standard interpretation for the standard logical symbols (i.e. for *TRUE*, *FALSE*, $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\exists$, $\forall$), formulae ϕ constructed using the temporal operators are interpreted over $\sigma = s_0 \rightarrow s_1 \rightarrow \ldots$ as follows.

$\bigcirc\phi|^n_\sigma = \mathbf{tt}$ iff $\phi|^{n+1}_\sigma = \mathbf{tt}$ or $n = length(\sigma)$
$\oplus\phi|^n_\sigma = \mathbf{tt}$ iff $\phi|^{n+1}_\sigma = \mathbf{tt}$ and $n < length(\sigma)$

Note that if n is the length of σ then $\bigcirc\phi|^n_\sigma$ is **tt** and hence the formula $\bigcirc$*FALSE* is true only at the end of the sequence σ. Similarly, $\neg\oplus$*TRUE* will be **tt** only at the end of a sequence. Similarly interpretation holds for the previous operators.

$\odot\phi|^n_\sigma = \mathbf{tt}$ iff $\phi|^{n-1}_\sigma = \mathbf{tt}$ or $n = 0$
$\bigcirc\phi|^n_\sigma = \mathbf{tt}$ iff $\phi|^{n-1}_\sigma = \mathbf{tt}$ and n > 0

Again note that the weak previous operator $\odot$ will always give **tt** at the beginning of a sequence, i.e. $n = 0$, and that the strong version $\oplus$ gives **ff**.

$\Diamond\phi|^n_\sigma = \mathbf{tt}$ iff there is some $i \geqslant 0$ such that $\phi|^{n+i}_\sigma = \mathbf{tt}$ and $n+i \leqslant length(\sigma)$.

$\Box\phi|^n_\sigma = \mathbf{tt}$ iff for every $i \geqslant 0$, $\phi|^{n+i}_\sigma = \mathbf{tt}$.

$\phi\ \mathcal{U}\ \psi|^n_s = \mathbf{tt}$ iff there is some $i \geqslant 0$ such that (*a*) $\psi|^{n+i}_\sigma = \mathbf{tt}$ and $n+i \leqslant length(\sigma)$ and (*b*) for all j, $j < i$, $\phi|^{n+j}_\sigma = \mathbf{tt}$.

$\phi\ \mathcal{W}\ \psi|^n_\sigma = \mathbf{tt}$ iff $((\phi\ \mathcal{U}\ \psi) \vee \Box\phi)|^n_\sigma = \mathbf{tt}$.

$\phi\ \mathcal{U}^+\ \psi|^n_\sigma = \mathbf{tt}$ iff $(\phi \wedge \oplus(\phi\ \mathcal{U}\ \psi))|^n_\sigma = \mathbf{tt}$.

$\phi\ \mathcal{W}^+\ \psi|^n_\sigma = \mathbf{tt}$ iff $(\phi \wedge \bigcirc(\phi\ \mathcal{W}\ \psi))|^n_\sigma = \mathbf{tt}$.

The past-time duals of the above are similar.

$\Diamond\phi|^n_\sigma = \mathbf{tt}$ iff there is some $i \leqslant n$ such that $\phi|^{n-i}_\sigma = \mathbf{tt}$.

$\Box\phi|^n_\sigma = \mathbf{tt}$ iff for every $i \leqslant n$, $\phi|^{n-i}_\sigma = \mathbf{tt}$.

$\phi\ \mathcal{S}\ \psi|^n_\sigma = \mathbf{tt}$ iff iff there is some $i \leqslant n$ such that (*a*) $\psi|^{n-i}_\sigma = \mathbf{tt}$ and (*b*) for all j, $j < i$, $\phi|^{n-j}_\sigma = \mathbf{tt}$.

$\phi\ \mathcal{Z}\ \psi|^n_\sigma = \mathbf{tt}$ iff $((\phi\ \mathcal{S}\ \psi) \vee \Box\phi)|^n_\sigma = \mathbf{tt}$.

$\phi\ \mathcal{S}^+\ \psi|^n_\sigma = \mathbf{tt}$ iff $(\phi \wedge \bigcirc(\phi\ \mathcal{S}\ \psi))|^n_\sigma = \mathbf{tt}$.

$\phi\ \mathcal{Z}^+\ \psi|^n_\sigma = \mathbf{tt}$ iff $(\phi \wedge \bigcirc(\phi\ \mathcal{Z}\ \psi))|^n_\sigma = \mathbf{tt}$.

The binary (future) chop operators.

$\phi\ \mathcal{C}\ \psi|^n_\sigma = \mathbf{tt}$ iff there are σ', σ'' with $length(\sigma') \geqslant n$ and $\sigma' \circ \sigma'' = \sigma$ such that if σ' is infinite then $\phi|^n_{\sigma'} = \mathbf{tt}$ otherwise both $\phi|^n_{\sigma'} = \mathbf{tt}$ and $\psi|^0_{\sigma''} = \mathbf{tt}$.

$\phi\ \mathcal{C}^*\psi|^n_\sigma = \mathbf{tt}$ iff either there are $\sigma_1, \sigma_2, \ldots, \sigma_k$, σ_{k+1}, with $length(\sigma') \geqslant n$ and $\sigma = \sigma_1 \circ \sigma_2 \ldots \sigma_k \circ \sigma_{k+1}$ such that $\phi|^n_{\sigma_1} = \mathbf{tt}$, $\phi|^0_{\sigma_i} = \mathbf{tt}$ for $i \in \{2..k\}$ and $\psi|^0_{\sigma_{k+1}} = \mathbf{tt}$ or there are $\sigma_1, \sigma_2, \ldots, \sigma = \sigma_1 \circ \sigma_2 \circ \ldots$ such that $\phi|^n_{\sigma_1} = \mathbf{tt}$ and for all $i \geqslant 2$, $\phi|^0_{\sigma_i} = \mathbf{tt}$ or there are $\sigma_1, \sigma_2, \ldots, \sigma_k, \sigma = \sigma_1 \circ \sigma_2 \circ \ldots \circ \sigma_k$ such that σ_k is infinite and $\phi|^n_{\sigma_1} = \mathbf{tt}$ and for $i \in \{1..k\}$, $\phi|^0_{\sigma_i} = \mathbf{tt}$

Finally, the fixed-point operators.

$\nu\xi.\chi(\xi)|^n_\sigma = \mathbf{tt}$ iff for all $k \geqslant 0$, $\chi^k(\underline{TRUE})|^n_\sigma = \mathbf{tt}$ where $\chi^k(\underline{TRUE})$ is the temporal formula

$$\underbrace{\chi(\chi(\ldots\chi(\underline{TRUE})\ldots))}_{k\text{-times}}$$

Assume the temporal formula $\chi(\xi)$ contains only positive occurrences of the temporal variable ξ, i.e. ξ occurs only under an even number of negations. The temporal formula $\nu\xi.\chi(\xi)$, in fact, then denotes the maximal fixed-point solution, with respect to implication ordering, of the implication $\xi \Rightarrow \chi(\xi)$. The obvious extension of ν to vectors is also used.

The interpretation of the minimal fixed-point construction is given in an analogous way.

$\mu\xi.\chi(\xi)|^n_\sigma = \mathbf{tt}$ iff there is some $k \geqslant 0$, $\chi^k(\underline{FALSE})$ where $\chi(\underline{FALSE})$ is the temporal formula

$$\underbrace{\chi(\chi(\ldots\chi(\underline{FALSE})\ldots))}_{k\text{-times}}$$