Report Number: WUCS-94-4

1994-01-01

Reasoning about Places, Times, and Actions in the Presence of Mobility

C. Donald Wilcox and Gruia-Catalin Roman

The current trend toward portable computing systems (e.g., cellular phones, laptop computers) brings with it the need for a new paradigm for thinking about designing distributed applications. We introduce the term mobile to refer to distributed systems that include moving, autonomous agents which loosely cooperate to accomplish a tastk. The fluid nature of hte interconnections between components in a mobile system provides new challenges and new opportunities for the research community. While we do not propsoe to have fully grasped the consequences of these systems, we believe that the notions of place, time, and action will be central in... Read complete abstract on page 2.

Reasoning about Places, Times, and Actions in the Presence of Mobility

C. Donald Wilcox and Gruia-Catalin Roman

Complete Abstract:

The current trend toward portable computing systems (e.g., cellular phones, laptop computers) brings with it the need for a new paradigm for thinking about designing distributed applications. We introduce the term mobile to refer to distributed systems that include moving, autonomous agents which loosely cooperate to accomplish a tastk. The fluid nature of hte interconnections between components in a mobile system provides new challenges and new opportunities for the research community. While we do not propsoe to have fully grasped the consequences of these systems, we believe that the notions of place, time, and action will be central in any model that is developed. In this paper, we show that these concepts can be expressed and reasoned about in the UNITY logic with a minimal amount of additional notation. We choose as an example an elevator control system, with minor modifications to give the system mobile characteristics. We begin with a high-level specification of the control system, one which does not include any mobile characteristics, and introduce the notions of place, time, and action as they arise in the specification refinement process. The result of the refinement is an abstract program, a specification of the local actions of the system along with restrictions on teh cooperation patterns between the various components.

Reasoning about Places, Times, and Actions in the Presence of Mobility

C. Donald Wilcox
Gruia-Catalin Roman

WUCS-94-04

April 1994

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Printed 4/22/94

Abstract

The current trend toward portable computing systems (e.g., cellular phones, laptop computers) brings with it the need for a new paradigm for thinking about and designing distributed applications. We introduce the term *mobile* to refer to distributed systems that include moving, autonomous agents which loosely cooperate to accomplish a task. The fluid nature of the interconnections between components in a mobile system provides new challenges and new opportunities for the research community. While we do not propose to have fully grasped the consequences of these systems, we believe that the notions of place, time, and action will be central in any model that is developed. In this paper, we show that these concepts can be expressed and reasoned about in the UNITY logic with a minimal amount of additional notation. We choose as an example an elevator control system, with minor modifications to give the system mobile characteristics. We begin with a high-level specification of the control system, one which does not include any mobile characteristics, and introduce the notions of place, time, and action as they arise in the specification refinement process. The result of the refinement is an abstract program, a specification of the local actions of the system along with restrictions on the cooperation patterns between the various components.

1. Introduction

One of the most exciting recent developments in the computing arena is the trend towards autonomy, mobility, and global communication. Cellular phone networks, laptop computers, autonomous vehicles, and intelligent badges are just a few of the application areas benefiting from this new technological revolution. Each of these applications involve distributed systems that include autonomous agents which change physical location, communicate intermittently, lack centralized control, exhibit cooperative behavior, and demand customized access to globally available resources. We choose the term *mobile* to refer to systems that share these particular characteristics.

While traditional multiprocessor and network architectures have well-established models for use in algorithm design and performance evaluation (e.g., shared-variables and message-passing), appropriate models for the study of mobile computing still need to be developed. Without putting forth at this time any specific proposals, we contend that place, time, and action are likely to emerge as central concepts in many models of mobile computing. The very notion of mobility presupposes the ability of a computing agent to alter its location in a manner which is visible to the overall computation. This may take place when communication with a moving vehicle is handed over to a new cell, when a portable computer enters a new room, or when a person, upon arriving home, reestablishes contact with some public or private network. In the first two examples computing elements physically travel through space while in the third case stationary equipment provides mobile users with ubiquitous access to computing resources. In distributed computing the notion of process is often used as a surrogate for place—arbitrary delays in message delivery are used to model (indirectly) the fact that processes occupy possibly distinct locations in space. In mobile computing, explicit modeling of an agent's location may be required since a location change may impact the overall behavior of the system. A command ship exiting a broadcast perimeter, for instance, may leave the rest of the fleet unable to perform its mission.

The proposal to include some notion of time in the model is more controversial, since it seems to imply that mobile systems are by definition real-time systems. There is some evidence, however, to suggest that time is of the essence in many mobile computations. First, it is no longer appropriate to assume arbitrary delays in message delivery, so the relation between time and space requires careful reevaluation. Further, the passage of time affects the location of computing agents and, therefore, alters the system's behavior. Weather data being broadcast periodically in a limited area may be missed by a passing train if the interval between broadcasts is greater than the travel time through the broadcast zone, and incomplete transfer of data may occur if the train exits the zone before the broadcast is complete. In this example, it is conceivable that, by knowing the movement pattern of an agent, requested data could be prefetched and delivered to next cell down the road in anticipation of a hand-off. Finally, location and velocity data could be used to control the data transmission rate, a high rate for when the agents are near each other and a low rate when they are far apart.

Because the composition of a mobile system is highly dynamic, the underlying model must permit both the creation of actions and dynamic changes in the nature of their interactions. The same agent, for instance, may participate at different times and in different places in point to point, multi-cast, or broadcast communication with a variety of other agents. The same action may have different consequences depending on when and where it is executed. The overall repertoire of interactions among actions associated with various agents must be richer than in the case of traditional architectures. Stationary components of the systems may rely on shared memory and message passing to communicate among themselves. Wireless communication, on the other hand, may assume the form of synchronous point-to-point communication or of one-way radio broadcast received only by listening agents. Furthermore, interactions among agents may not be possible at all times, and while eventual reception of the data cannot assumed any longer, the delivery order can be taken for granted. These assumptions are quite different from those typically made about a distributed systems (in the absence of failures).

Despite all these complicating factors, application-oriented models of mobile computing will need to create the illusion of utmost simplicity, i.e., a clean abstract picture in which place, time, and actions may not appear at all. Unfortunately, elegant and dependable application interfaces will have to be supported by system software and communication protocols whose design and verification will necessarily have to consider issues such as those outlined above. Providing reliable mobile computing systems is likely to entail increased reliance on formal derivation techniques during design. The goal of this paper is to demonstrate that simple extensions to established formal specification and design techniques can accommodate the concepts of place, time, and action in a manner consistent with design requirements for mobile systems. We make no attempt to offer a completely general or comprehensive treatment of mobile computing, merely a representative illustration of the manner in which several important concepts can be factored into a formal design process.

Our investigation centers around the specification refinement methods associated with the UNITY logic, a specialization of linear temporal logic. (The presentation, while of a technical nature, is intended for a relatively broad audience. All proofs have been omitted and notation is introduced only as needed.) A UNITY specification consists of assertions over an abstract global state of the system as a whole. Safety properties are expressed in terms of **invariants**—properties which hold for all states of the computation—and an **unless** relation which constrains the set of permissible state transitions. Progress properties are (usually) captured by the **leads-to** relation which expresses the notion that certain states must eventually be reached from certain other specified states. Design is viewed as a process by which an initial highly-abstract specification is gradually refined into increasingly more concrete specifications up to a point where constructing a correct program becomes a trivial task.

Faithful to the notion that application-oriented specifications ought to be simple and concise, we assume the starting point to be a typical UNITY-style high-level specification free of references to places, times, and actions. The concept of place is captured by introducing an additional attribute for each component of the abstract state. Locus information is used during refinement to guide data partitioning among processes, to distinguish local data transformations from interprocess communication, and to reason about mobility and its impact on the communication capabilities of moving system components. Time is introduced as a distinguished variable not accessible to the programmer. Timing constraints (or assumptions) are expressed as invariants involving auxiliary variables which record the time associated with particular state transitions in the system. Such invariants are used to prove that specific undesirable behaviors (such as missing a needed broadcast) are actually ruled out when these temporal conditions are imposed upon the physical realization of the system. The introduction of actions is the first step towards writing a concrete program. Each action is characterized as a local transformation—intuitively corresponding to a single statement execution—or as the local effect of some interprocess communication involving several distinct locations. Ultimately, we generate not a concrete program in some particular programming language, but an abstract one whose actions capture local effects and synchronization requirements defining non-local interactions (stated as invariant properties). The former may be realized as simple statements; the latter may involve system software managing communications across a traditional network or using wireless technology. Modularity and possible reusability of components provides the motivation for this particular formulation of the abstract program.

The remainder of the paper illustrates these ideas on an elevator control problem modified to exhibit some characteristics of a mobile system while preserving the simplicity and familiarity of an easy to understand application. Mobility comes from the fact that the control system is associated with the elevator cabin. Wireless communication is introduced as a mechanism by which the elevator becomes aware of the existence (but not the source) of service requests above and below. As the elevator moves, the communication pattern changes with floors "above" becoming floors "below." A general specification of the elevator problem appears in Section 2. Several refinements of the initial specification are described in Section 3. They provide a brief tutorial to UNITY-style specification refinement. The concept of place and the associated notation are introduced in Section 4 which continues the refinement process. Section 5 outlines our approach to reasoning about time and related refinements. The abstract program is generated in Section 6, which also presents our proposed notation for actions and synchronization. Related work and methodological implications of our approach are considered at the end of Sections 2 to 6. Conclusions appear in Section 7.

2. Initial Specification

In this section we introduce the elevator problem used to illustrate our methods and we provide an overview of the UNITY logic underlying our general approach. In order to make the presentation accessible to a broad audience, we will explain the logic and accompanying notation informally. Our primary objective is to develop, for the elevator problem, a formal specification that is at once both concise and general. Concise in the sense that it is expressed in a few simple properties which can be easily understood; and general in the sense that no reasonable design solutions are being ruled out from the onset. Because this is an exercise intended to introduce and explain certain narrow technical issues relating to the specification and design of mobile systems, we will occasionally make simplifying assumptions whose net effect is to reduce the complexity of our example without changing the essence of the design process.

The elevator system consists of a single cabin servicing an *N* floor building. The cabin has doors (which may be opened or closed) and *N* request buttons, one for each floor. When one of the request buttons is pressed, it is lighted and remains lit until the cabin arrives at the requested floor with the doors opened. Similarly, at each floor there is a pair of call buttons, one for up and one for down. Naturally there is no down button on the first floor, nor is there an up button on the *N*th floor. When a call button on a floor is pressed, it is lighted and remains lit until the elevator arrives at the floor with the doors opened and servicing the direction of the call. For safety reasons, the elevator doors must be closed while the elevator is moving. To add an element of mobility we will assume that the computer controlling the elevator movement resides with the cabin itself and has no hard-wired communication links with the call buttons. Thus, the call buttons may be accessed only when the elevator is in the immediate proximity of the particular floor. Knowledge about the existence of calls above or below the cabin is received via some nonspecific wireless signal being broadcast repeatedly by any floor having a lit button. The cabin can differentiate signals only as originating above or below. More specifically, we allow the cabin to know that there is a call at a floor *f* only when the cabin is *at f*. All that can be known otherwise is that there is a request in a particular direction, either up or down from the cabin's current location, but not the exact position of the request.

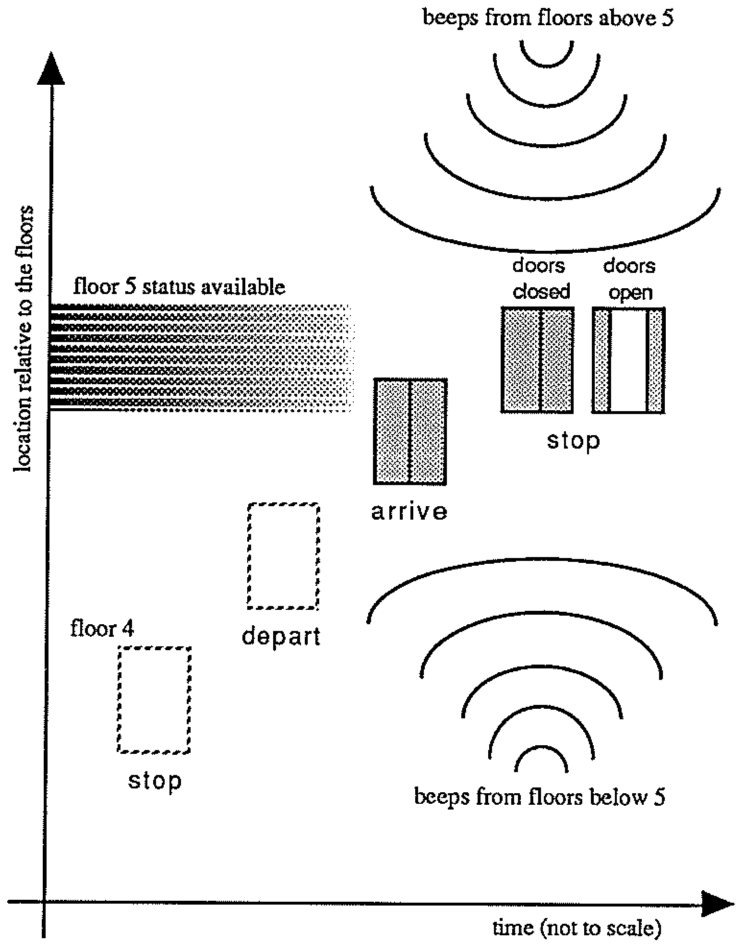


Fig. 1. Representative behavior and interaction pattern

For the sake of simplicity, we treat the cabin and the floor buttons as a single closed system, and for the time being, we ignore architectural details having to do with mobility and wireless communication. These issues will be brought back into the picture as we refine the specification towards an implementable program. At this point we focus on issues of state representation and on high level policies or performance goals governing the elevator movement. To ensure some minimal level of *quality of service*, we require that the elevator not change service direction as long as it has calls and requests still to be serviced in the current direction. This requirement eliminates pathologic behaviors which might cause calls to be unserviced for arbitrarily-long periods of time. Later design steps will have to implement this policy by establishing rules which identify the conditions under which the elevator stops at a floor, moves along the elevator shaft, or changes direction. We model the elevator's movement using three states: *arrive*, *stop*, *depart*. In this model, the elevator makes the decision whether or not to stop at a floor while in the *arrive* state, and makes the decision on service direction in the *stop* state. The *depart* state models the state in which the elevator has committed to a decision and is moving in accordance with that decision. This three-state model maps to a two state model in which *arrive* and *depart* correspond to the elevator moving.

up(f)	The up call button on floor *f* is lit.
dn(f)	The down call button on floor *f* is.
rq(f)	The request button for floor *f* inside the elevator is lit.
dir(δ)	The service direction is *δ*. If *δ* equals -1, the elevator is servicing floors below the cabin; if *δ* equals 1, floors above the cabin are being serviced.
at(f)	The cabin is on or departing floor *f*.
open	The doors of the elevator are open.
stop	The elevator is stopped.
arrive	The elevator has just arrived on a floor, and is still moving.
depart	The elevator is heading for the next floor.

Fig. 2. The abstract state space

The state space is outlined in figure 2. The predicates *up(f)*, *dn(f)*, and *rq(f)* model the up call buttons on the floors, the down call buttons on the floors, and the request buttons in the cabin, respectively. When one of these predicates is true, it means not only that the button has been pressed, but that it is lit, and that it is in some way impacting the elevator's decision-making process. The current service direction is *dir(δ)*, where the parameter *δ* has the value –1 or 1: *dir(1)* means that the elevator is servicing requests for floors above its current location; *dir(–1)* floors below. There is no implication of movement in the service direction — the elevator has a current service direction even when stopped. However, if the elevator does move, it will move in the current service direction. We model the elevator cabin's current location using the predicate *at(f)*. While it is reasonable to consider the cabin's position as a real number between 1 and *N*, using such a representation introduces complexities into the specification which have little to do with the behavior of the elevator, but are rather an artifact of the continuity of real numbers. Since our goal in the initial specification is simplicity and clarity, we opt for a model which considers the cabin to be always at some floor, so the parameter *f* in *at* assumes only integer values between 1 and *N*. We model the elevator cabin doors with a single predicate *open*, which is true when the doors are opened, and false when they are closed, since this is the only characteristic of the doors that is of interest to us. Finally, there are the predicates *arrive*, *stop*, and *depart*, as described in the previous paragraph.

floor(f)	$\equiv f \in \{1, \ldots, N\}$
on(f)	$\equiv at(f) \wedge (stop \vee arrive)$
move	$\equiv arrive \vee depart$
closed	$\equiv \neg open$
fromto(f,δ,f′)	$\equiv sign(f'-f)=\delta$
call(f)	$\equiv up(f) \vee dn(f)$
needed(f,δ)	$\equiv rq(f) \vee (\delta=1 \wedge up(f)) \vee (\delta=-1 \wedge dn(f))$
work(f)	$\equiv rq(f) \vee call(f)$

Fig. 3. Basic definitions

The elevator's behavior specification is given in figures 4 and 5. To simplify the specification, we make use of some definitions which are given in figure 3. The type specification (F1-F5) contains predicates which make explicit the valid range for free variables in the state predicates and formalizes relationships among them, e.g., F3 expresses the mutual exclusivity of the predicates *arrive*, *stop*, and *depart*. While these predicates are important, they are rarely referenced explicitly in the remainder of the specification refinement. All these predicates are invariant, i.e., they hold initially and throughout the execution of the program.

F1a:	**inv.**	$up(f) \Rightarrow floor(f) \wedge f < N$
F1b:	**inv.**	$dn(f) \Rightarrow floor(f) \wedge 1 < f$
F1c:	**inv.**	$rq(f) \Rightarrow floor(f)$
F2a:	**inv.**	$at(f) \Rightarrow floor(f)$
F2b:	**inv.**	$at(f) \wedge at(f') \Rightarrow f = f'$
F2c:	**inv.**	$\langle \exists f :: at(f) \rangle$
F3a:	**inv.**	$arrive \Rightarrow \neg depart$
F3b:	**inv.**	$depart \Rightarrow \neg arrive$
F3c:	**inv.**	$stop \Leftrightarrow \neg move$
F4:	**inv.**	$(dir(1) \wedge \neg dir(-1)) \vee (dir(-1) \wedge \neg dir(1))$
F5:	**inv.**	$\neg(((at(1) \wedge dir(-1)) \vee (at(N) \wedge dir(1))) \wedge depart)$

Fig. 4. Properties constraining the form of the state predicates

The behavior specification for the elevator system is given in figure 5[1]. When the elevator arrives at a floor, it must make a decision whether to stop at the floor or to continue on to the next floor (F6). If there is a request at the floor in the current service direction, it must be serviced before the elevator leaves the floor (F12). Since we require that the elevator be stopped before the doors can be opened (F9), the elevator will stop, turn off the appropriate call and request lights (F11), and open the doors, at which time the call is considered serviced. Until the call is serviced, the call light must remain on (F13); there is no mechanism for canceling a call. If there are additional calls on other floors in the same service direction, then the elevator must close the doors and depart the floor without changing service direction (F7, F12). If there are no calls in the current service direction, the elevator may change service direction only when stopped (F10). Movement between floors is modeled as occurring in the transition from *depart* to *arrive* — when the elevator makes this state transition, it also moves one floor in the current service direction (F8).

All of these properties are safety properties and they do not require that anything actually happen. The elevator could satisfy the specification thus far by not doing anything. To gain the desired activity, we need only add a single progress property to the specification: if there is a call or request for a floor, it will be serviced eventually (F14)[2]. At this point we have a specification which is concise, complete, and formal (except for mobility and wireless communication constraints). We are ready to start the refinement process.

1 Along with invariant properties, this specification makes use of formulas of the form

p **unless** *q*

This formula requires that, if the system is in a state in which *p* is *true* and *q* is not, then executing any statement leaves the system in a state in which either *p* or *q* is *true*. The **unless** property models constraints on permissible state transitions.

2 (F13) is a **leads-to** property (written $\mapsto$). It specifies that if the system ever reaches a state in which the left-hand side is *true*, then eventually it must reach a state in which the right-hand side is *true*. In general, **leads-to** properties do not require that the left-hand side remain *true* in the meantime. The **leads-to** relation is transitive, i.e., given $P \mapsto Q$ and $Q \mapsto R$, one can conclude $P \mapsto R$.

F6: $\mathrm{at}(f) \land \mathrm{arrive}$ **unless** $\mathrm{at}(f) \land ((\mathrm{stop} \land \mathrm{closed}) \lor \mathrm{depart})$
Having arrived at a floor, the elevator can either stop at the floor or immediately depart

F7: $\mathrm{at}(f) \land \mathrm{stop}$ **unless** $\mathrm{at}(f) \land \mathrm{depart}$
Once stopped, the elevator can begin moving away from the floor

F8: $\mathrm{at}(f) \land \mathrm{depart} \land \mathrm{dir}(\delta)$ **unless** $\mathrm{at}(f+\delta) \land \mathrm{arrive}$
The elevator moves in the current service direction,
departing one floor and then arriving at the next

F9: **inv.** $\mathrm{open} \Rightarrow \mathrm{stop}$
The doors are open only if the elevator is stopped

F10: $\mathrm{dir}(\delta)$ **unless** $\mathrm{stop} \land \mathrm{closed}$
The elevator changes directions only when stopped with the doors closed

F11a: **inv.** $\mathrm{dir}(1) \land \mathrm{at}(f) \land \mathrm{stop} \land \mathrm{open} \Rightarrow \neg\mathrm{up}(f) \land \neg\mathrm{rq}(f)$
F11b: **inv.** $\mathrm{dir}(-1) \land \mathrm{stop} \land \mathrm{at}(f) \land \mathrm{open} \Rightarrow \neg\mathrm{dn}(f) \land \neg\mathrm{rq}(f)$
The call lights are not turned on when the elevator is immediately available

F12: $\mathrm{dir}(\delta) \land \mathrm{needed}(f,\delta) \land \langle \exists f' : \mathrm{at}(f') :: \mathrm{fromto}(f',\delta,f) \lor f'=f \rangle$
unless $\mathrm{at}(f) \land \mathrm{stop} \land \mathrm{dir}(\delta) \land \mathrm{open}$
The elevator services all calls and requests in the current service direction before changing directions

F13a: $\mathrm{up}(f)$ **unless** $\mathrm{at}(f) \land \mathrm{stop} \land \mathrm{dir}(1) \land \mathrm{open}$
F13b: $\mathrm{dn}(f)$ **unless** $\mathrm{at}(f) \land \mathrm{stop} \land \mathrm{dir}(-1) \land \mathrm{open}$
F13c: $\mathrm{rq}(f)$ **unless** $\mathrm{at}(f) \land \mathrm{stop} \land \mathrm{open}$
Calls and requests are not canceled except when actually serviced

F14: $\mathrm{needed}(f,\delta) \mapsto \mathrm{at}(f) \land \mathrm{stop} \land \mathrm{dir}(\delta) \land \mathrm{open}$
Calls and requests are eventually serviced

Fig. 5. Behavior specification

Discussion. Although the number of concurrency models that have been proposed so far is overwhelming, when it comes to reasoning about concurrent computations the dominant models fall into only two broad categories: event-based and state-based. Event-based models (such as CSP[8] and CCS[14]) sought their mathematical foundation in algebra, while state-based models (such as temporal logic[17], TLA[12], and UNITY[2]) turned to logic. Algebraic models favor constructivist approaches to software design; they start with simple components having known properties and combine them into larger ones whose properties may be computed. The use of logic, on the other hand, favors starting with an initial highly-abstract specification which is gradually refined to the point where it contains so much detail that writing a correct program becomes trivial.

Among state-based approaches, UNITY combines the expressive power of the linear-time logic with the conceptual simplicity of Hoare-style predicates. All properties are expressed in terms of predicates that hold in every state. Reasoning about execution sequences becomes unnecessary. By considering properties of the global system state the formulation of the initial abstract specification is simplified to a significant extent while design biases and modeling artifacts are kept to a minimum. This is made evident by the specification proposed in this section, a specification which ultimately addresses only three issues: abstract state representation, service obligations, and service policy. Even though the predicate *at(f)* appears in the specification, there is nothing to suggest that the solution must be a mobile system or that wireless communication must be present in the final design. These considerations will be factored in as we gradually bias refinements towards compatibility with the physical constraints the system must eventually meet.

Regardless of the model one uses, achieving simplicity requires a solid understanding of the problem, proper choice of notation, and elegant modeling of the environment—it is a generally accepted fact that a complete specification must include the environment. A key decision we took regarding modeling the environment was to treat the elevator cabin software and its environment (floor buttons) as a closed system. The closed-system assumption is motivated by our desire to avoid dealing with conditional properties, which can add a certain degree of complexity to the specification and verification processes. As long as the assumptions about the environment are clearly distinguishable from the properties of the cabin software and the environment is not unduly restricted, the design process is simplified without loss of generality. The price we pay for this becomes evident only if we compose the cabin software with some other device and attempt to prove properties of the composite, in such a case an equivalent open-system specification may need to be developed. This potential penalty may be acceptable if the design is made simpler by the closed-system assumption—our experience to date leads us to believe that this is actually the case.

3. Specification Refinement

The goal of this section is to review UNITY-style specification refinement and to set the stage for introducing the concepts of *place*, *time*, and *action*. These concepts are not present in the initial specification because the issues which motivate them (in this specific problem) relate primarily to the design rather than to the essence of the problem. We believe that in some situations such concepts could appear even in the initial specification. However, for the purposes of this presentation, we prefer a problem where these concepts are introduced gradually, as they become relevant to the design process. Our strategy in this section is to refine the initial specification to the point where the control logic is revealed and the necessity of considering the distribution of data and the mobility of the components becomes apparent.

The basic idea behind specification refinement is to massage the initial abstract specification until it becomes sufficiently concrete to suggest an immediate implementation. Properties are gradually strengthened, with abstract data representations migrating towards traditional data structures and processing goals being reshaped into algorithms. In this section, we will focus our attention on refining progress (F14) in order to replace a service obligation by a decision-making process that governs the elevator and door movements. The motivation behind these refinements is provided by properties F6-F8, which describe the transitions which are allowed within this process. Figure 6 presents these transitions graphically. We present the refinements informally, with proofs omitted for the sake of brevity (the proofs can be found in a related technical report).

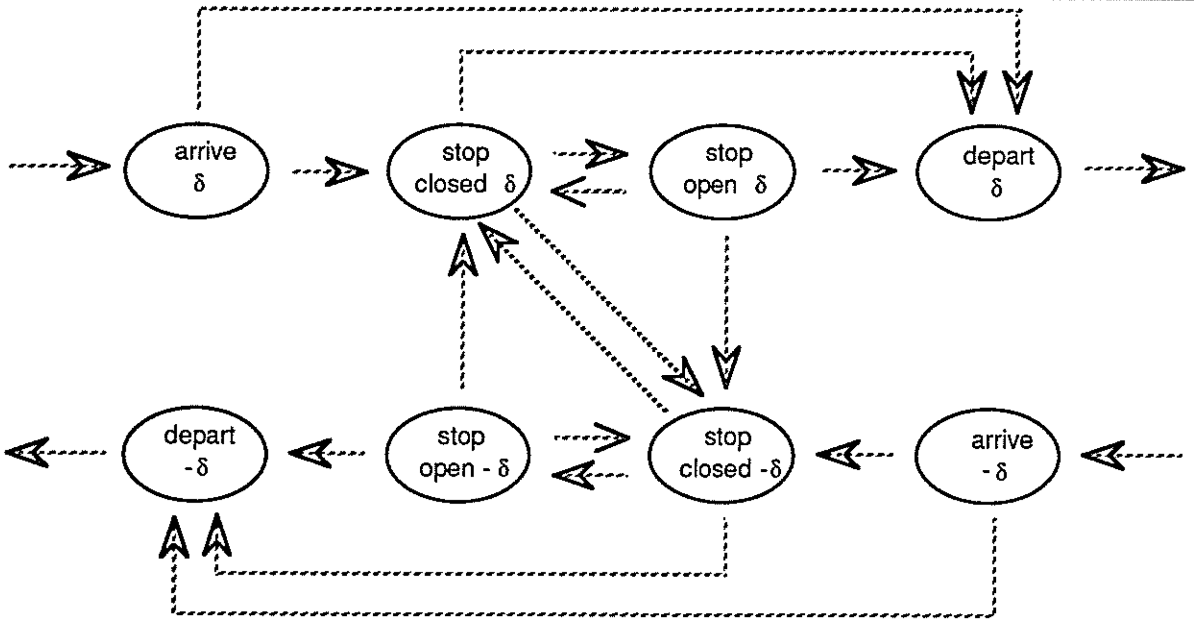


Fig. 6. Legal state transitions
(δ represents the current service direction)

Refinement 1: Introduce elevator movement. In the first refinement, we simply observe that when there is work to do, the elevator must move, and eventually this movement should result in the request being serviced. We refine F14 to require that, when the elevator is making a decision (in either the *arrive* or *stop* states), then the presence of work to be done must result in the elevator moving. The refined specification (shown in figure 7) encapsulates a decision which for the time being makes use of global information, i.e., the presence of calls at floors other than the one at which the cabin is located. If such a call exists, then the elevator must eventually *arrive* at that floor with the call (F14.1). If the elevator is *on* a floor needing service (where *on* implies that the elevator is in either the *arrive* or *stop* states) and heading in the right direction, then the elevator must stop at that floor (F14.2). Finally, when the elevator is departing one floor (*depart* state), we require it to arrive at the next floor (*arrive* state). The proof that refined properties (F14.1-F14.3) imply the original property (F14) follows from the transitivity of the **leads-to** relation.

F14: $\text{needed}(f,\delta) \mapsto \text{at}(f) \wedge \text{stop} \wedge \text{dir}(\delta) \wedge \text{open}$

is refined to

F14.1: $\text{needed}(f,\delta) \wedge \text{on}(f') \wedge \text{dir}(\delta') \wedge (f \neq f' \vee \delta \neq \delta') \mapsto \text{needed}(f,\delta) \wedge \text{on}(f) \wedge \text{dir}(\delta)$

F14.2: $\text{needed}(f,\delta) \wedge \text{on}(f) \wedge \text{dir}(\delta) \mapsto \text{at}(f) \wedge \text{stop} \wedge \text{dir}(\delta) \wedge \text{open}$

F14.3: $\text{at}(f) \wedge \text{depart} \wedge \text{dir}(\delta) \mapsto \text{at}(f+\delta) \wedge \text{arrive}$

where

$\text{on}(f) \equiv \text{at}(f) \wedge (\text{stop} \vee \text{arrive})$

Fig. 7. Introduce elevator movement

Refinement 2: Define a distance metric. F14.1 is not specific about how the elevator moves. It simply requires that the elevator arrive at floors which must be serviced. Physically, the elevator moves from one floor to the next, gradually getting closer to a particular floor requiring service. Some formal definition of "closer" is useful in measuring (and proving) progress. We choose as a metric the maximum number of floors which the elevator would have to travel, given its current location and service direction, before it would pass by the given floor moving in the correct direction (under the assumption that the elevator does not change service directions if there is work to do in the current direction). For example, if the elevator is currently on the third floor heading down, and there is a call on the fourth floor to go up, then *dist*((3,–1),(4,1)) is 5, since the elevator can move down at most two floors, at which time it must change service direction, and will arrive at the fourth floor after traveling three additional floors. The refinement is detailed in figure 8. The proof of correctness for this refinement makes use of the *induction principle* for **leads-to** (see [2])—given a well-founded metric which measures how close the current state is from a desired goal, the fact that in any state the metric eventually decreases guarantees that the goal state is eventually reached. This refinement still hides the decision making process guiding the elevator movement. However, we are now at a point where it is possible to expose the process in one simple refinement.

F14.1: $\text{needed}(f,\delta) \wedge \text{on}(f') \wedge \text{dir}(\delta') \wedge (f \neq f' \vee \delta \neq \delta') \mapsto \text{needed}(f,\delta) \wedge \text{on}(f) \wedge \text{dir}(\delta)$

is refined to

F14.1.1: $\text{needed}(f,\delta) \wedge \text{on}(f') \wedge \text{dir}(\delta') \wedge \text{dist}((f',\delta'),(f,\delta)) = k \wedge k > 0$

$\mapsto \quad (\text{needed}(f,\delta) \wedge \langle \exists f'', \delta'' : \text{on}(f') \wedge \text{dir}(\delta') :: \text{dist}((f',\delta'),(f,\delta)) < k \rangle) \vee$

$\quad (\text{needed}(f,\delta) \wedge \text{on}(f) \wedge \text{dir}(\delta))$

where

$$\text{dist}((j,\delta'),(i,\delta)) = \begin{cases} 2(N-1)-\delta(i-j) & \text{if } \delta=\delta' \wedge \text{fromto}(i,\delta,j) \\ \delta(i-j) & \text{if } \delta=\delta' \wedge (\text{fromto}(j,\delta,i) \vee i=j) \\ 2N-(i+j) & \text{if } \delta\neq\delta' \wedge \delta=-1 \\ i+j-2 & \text{if } \delta\neq\delta' \wedge \delta=1 \end{cases}$$

Fig. 8. Define a distance metric

Refinement 3: Expose the processing logic controlling the elevator. Our final refinement in this section is motivated by properties F6 and F7, which tell *what* the elevator might do but not *why*. Here we make explicit the movement control logic for the elevator as it pertains to each floor—the refinement is shown in figure 9. The **until** property is a **leads-to** combined with an **unless**, and unlike **leads-to**, it requires that the predicate on the left-hand side remain true up to the point when the right-hand side is established. Each of the three new properties in figure 9 reflect one of the three states in which the elevator can make movement decisions: it is arriving at a floor (F14.1.1.1), it is stopped with the doors opened (F14.1.1.2), or it is stopped with the doors closed (F14.1.1.3).

When arriving at a floor, the elevator must decide whether or not to stop. By F14.1.1.1, the elevator does not stop at the floor if it is not needed there but is needed elsewhere in the current service direction. All other conditions result in the elevator stopping at the floor. When stopped at a floor with the doors opened (F14.1.1.2): (1) the elevator can continue going in the same service direction if there are calls or requests that must be attended to, or (2) it can close the door while changing the service direction. The decision is made concurrently with the closing of the doors, thus making it impossible for the elevator to become held at a floor by repeatedly calling the elevator to the current floor after the doors are closed but before the elevator departs. Finally, when stopped at a floor with the doors closed, but with work to be performed (F14.1.1.3): (1) the elevator can open the doors to service a request at the current floor; (2) it can continue in the current service direction to service a call or request, or (3) it can change service directions to attend to a call or request in the opposite direction. Note that if there is no work to be done in any direction, the elevator's behavior is unconstrained.

The form of these properties is typical of progress properties in a reactive system. The left-hand side of each property is a predicate which describes that portion of the system's state which is under the control of the system itself, in this case the cabin's location and service direction. The right-hand side of each property is a disjunction of possible transitions, each of which combines the required transition with those conditions outside of the direct control of the system which effect the decision of which transition to make. This form is required by the use of the **until** property, which requires that the left-hand side of the property remain true up to the point that the right-hand side is established. If the left-hand side contained predicates which reflected portions of the state space over which the system has no direct control (in this case the call and request buttons), then it would not be possible to guarantee that the left-hand side was not falsified without placing undesirable constraints on the behavior of the environment. The form of these properties comes up again when we introduce actions, when the reactive portions of the right-hand side become enabling conditions for the actions in the abstract program.

F14.1.1: $\text{needed}(f,\delta) \wedge \text{on}(f') \wedge \text{dir}(\delta') \wedge \text{dist}((f',\delta'),(f,\delta)) = k \wedge k > 0$
$\quad\mapsto$ $(\text{needed}(f,\delta) \wedge \langle \exists\, f'', \delta'' : \text{on}(f'') \wedge \text{dir}(\delta'') :: \text{dist}((f'',\delta''),(f,\delta)) < k \rangle) \vee$
$\quad(\text{needed}(f,\delta) \wedge \text{on}(f) \wedge \text{dir}(\delta))$

is refined to

F14.1.1.1: $\text{work\_in}(f,\delta') \wedge \text{at}(f) \wedge \text{dir}(\delta) \wedge \text{arrive}$
until $(\neg\text{needed}(f,\delta) \wedge \text{must\_go}(f,\delta) \wedge \text{at}(f) \wedge \text{depart} \wedge \text{dir}(\delta)) \vee$
$(\text{at}(f) \wedge \text{stop} \wedge \text{closed} \wedge \text{dir}(\delta))$

F14.1.1.2: $\text{work\_in}(f,\delta') \wedge \text{at}(f) \wedge \text{stop} \wedge \text{open} \wedge \text{dir}(\delta)$
until $(\text{must\_go}(f,\delta) \wedge \text{at}(f) \wedge \text{depart} \wedge \text{dir}(\delta)) \vee$
$(\neg\text{must\_go}(f,\delta) \wedge \text{at}(f) \wedge \text{stop} \wedge \text{closed} \wedge \text{dir}(-\delta))$

F14.1.1.3: $\text{work\_in}(f,\delta') \wedge \text{at}(f) \wedge \text{stop} \wedge \text{closed} \wedge \text{dir}(\delta)$
until $(\text{at}(f) \wedge \text{stop} \wedge \text{dir}(\delta) \wedge \text{open}) \vee$
$(\neg\text{needed}(f,\delta) \wedge \text{must\_go}(f,\delta) \wedge \text{at}(f) \wedge \text{depart} \wedge \text{dir}(\delta)) \vee$
$(\neg\text{needed}(f,\delta) \wedge \neg\text{must\_go}(f,\delta) \wedge \text{work\_in}(f,-\delta) \wedge$
$\quad\text{at}(f) \wedge \text{stop} \wedge \text{closed} \wedge \text{dir}(-\delta))$

where

$\text{must\_go}(f,\delta) \equiv \langle \exists\, f' : \text{fromto}(f,\delta,f') :: \text{work}(f') \rangle$
$\text{work\_in}(f,\delta) \equiv \text{needed}(f,\delta) \vee \text{must\_go}(f,\delta)$

Fig. 9. Expose the processing logic controlling the elevator

The proof that this refinement is correct is more complex. Our approach is to introduce a metric which measures distance with greater precision than *dist*, a metric which accounts for movement within the floor as well as between floors, and to show that the new refinement reduces this metric. It should also be noted that this refinement includes F14.2, which is therefore no longer needed.

Having refined the progress properties to the point that the mechanism by which the elevator makes movement decisions is explicit, we could generate an executable program from this specification. However, the decision-making process is dependent upon information about the location of remote calls (in the predicate *must_go*), which is incompatible with the assumed underlying architecture. This latter concern will guide the remainder of our refinements. However, before we can consider this requirement formally we need to formalize the notion that a particular predicate involves data *at some location*. This is the subject of the next section.

Discussion. In sequential programming, formal derivation enjoys a prestigious tradition [5, 6, 15, 16]. UNITY [2] builds directly on this formal foundation by extricating proofs from the program text and by advocating a refinement strategy in which the program emerges only in the last step. The kind of refinements appearing in this section are illustrative of a process in which a high-level property is replaced by lower-level properties which imply it. Several case studies involving the application of this methodology to a variety of problems are available in the literature [2, 9, 23] and an industrial-grade application [24] has been reported recently. The approach was also integrated in a methodology for the derivation of concurrent rule-based programs [20] and for architecture-directed refinement [21]. An alternate approach to specification refinement is that of program or action refinement. Lamport [11] is credited with introducing the notion of refinement mappings which led to much subsequent research along these lines (see [4] for representative papers). Although the more operational flavor of action refinement appeals to many researchers, we contend that operational approaches are better suited in the later stages of design.

4. The Concept of Place

One simple approach to introducing the notion of space is to attach location as an additional attribute in the state representation. Since our specifications characterize states by using predicates over the state space, it is only natural to treat the location attribute as an extra argument of any of the state predicates. This way one can think of a property as being true at some particular location. We call these augmented state predicates *spatial predicates*. Since the technique is basically one of data refinement, coupling invariants are used to relate the original state predicates to their spatial counterparts. To facilitate distinguishing spatial predicates from normal state predicates, we will write spatial predicates using a sans serif font in all caps, whereas normal state predicates are written in lower case using a serif font. The new notation and several useful definitions appear in figure 10.

P1:	$\mathsf{P}(v)\#\lambda \equiv \mathsf{P}(\lambda, v)$
P2:	$\mathsf{P}(v) \equiv \langle \exists\, \lambda :: \mathsf{P}(v)\#\lambda \rangle$
P3:	$\mathsf{P}(v)@\lambda \equiv \mathsf{P}(v)\#\lambda \wedge \langle \forall\, \lambda' : \lambda' \neq \lambda :: \neg \mathsf{P}(v)\#\lambda' \rangle$
P4:	$\mathsf{P}(v)@ \equiv \langle \exists\, \lambda :: \mathsf{P}(v)@\lambda \rangle$
P5:	$[\mathsf{P}(x) \wedge \mathsf{Q}(y) \wedge r(z)]\#\lambda \equiv \mathsf{P}(x)\#\lambda \wedge \mathsf{Q}(y)\#\lambda \wedge r(z)$

Fig. 10. Spatial notation

A spatial predicate *P* is a state predicate which is quantified over some set Λ, the set of *places*. We introduce the notations $P\#\lambda$, read "*P* is true *at least at* location λ"; and $P@\lambda$, read "*P* is true *only at* location λ" where λ ranges over Λ. The new notation (see property P1) is simply a convenient way of writing predicates which have one additional (distinguished) free variable, and reasoning about spatial predicates follows the rules of first-order predicate logic. Ordinary predicates may be seen as spatial predicates in which the location is existentially bound (P2) thus indicating that the predicate is true *somewhere* (as opposed to everywhere). The shorthand notation @ is used when a predicate is true in a unique place (P3). If the specific location is not important (P4), it can be omitted, as in $P(v)@$. Finally, we allow spatial qualifiers to distribute over other logical operators (P5). When this is done, the spatial quantification affects only spatial properties. This notation proved very useful in differentiating among local and global properties since the latter kind involve more than one location.

Refinement 4: Introduce location information. Using our notation, we can now refine the specification to include explicit references to the places where various state components are located. The set of places Λ over which the spatial predicates are quantified is the set of floors, i.e., integers between 1 and *N*. The spatial predicates are introduced in figure 11. The assignment of locations to predicates is straightforward: properties of call buttons are located on the respective floors (F16) and all other properties are co-located with the elevator cabin (F17, F18). The predicates *CABIN#f* and *ON#f* are the spatial counterparts of *at(f)* and *on(f)* (F19).

F16a:	**inv.** $\mathsf{UP}\#f \Leftrightarrow up(f)$
F16b:	**inv.** $\mathsf{DN}\#f \Leftrightarrow dn(f)$
F16c:	**inv.** $\mathsf{CALL}\#f \Leftrightarrow call(f)$
F17a:	**inv.** $\mathsf{ARRIVE}\#f \Leftrightarrow at(f) \wedge arrive$
F17b:	**inv.** $\mathsf{STOP}\#f \Leftrightarrow at(f) \wedge stop$
F17c:	**inv.** $\mathsf{DEPART}\#f \Leftrightarrow at(f) \wedge depart$
F18a:	**inv.** $\mathsf{DIR}(\delta)\#f \Leftrightarrow at(f) \wedge dir(\delta)$
F18b:	**inv.** $\mathsf{RQ}(f')\#f \Leftrightarrow at(f) \wedge rq(f')$
F18c:	**inv.** $\mathsf{OPEN}\#f \Leftrightarrow at(f) \wedge open$
F18c:	**inv.** $\mathsf{CLOSED}\#f \Leftrightarrow at(f) \wedge closed$
F19a:	**inv.** $\mathsf{CABIN}\#f \Leftrightarrow [\mathsf{ARRIVE} \vee \mathsf{STOP} \vee \mathsf{DEPART}]\#f$
F19b:	**inv.** $\mathsf{ON}\#f \Leftrightarrow on(f)$

Fig. 11. Introducing spatial predicates

Refinement 5: Segregate local and global predicates. Now that we have a convenient notation for expressing location, we can refine the specification to make explicit those properties which require access to non-local information. All progress properties (F41.1.1.1-F14.1.1.3) use the predicate *work_in*, which refers to floors other than the one at which the cabin is located. Here we re-write this property to separate its local and non-local components, as outlined in figure 12.

$at(f) \wedge work\_in(f,\delta)$

is refined to

$LREQ(\delta)\#f \vee rcall(f,\delta)$

where

$LREQ(\delta)\#f \equiv [CABIN \wedge ((\delta=1 \wedge UP) \vee (\delta=-1 \wedge DN) \vee \langle \exists f' : fromto(f,\delta,f') \vee f=f' :: RQ(f') \rangle)]\#f$
$rcall(f,\delta) \equiv (CABIN\#f \wedge \langle \exists f' : fromto(f,\delta,f') :: CALL\#f' \rangle)$

Fig. 12. Refining *work_in* into local and non-local components

The transformation is purely mechanical. Access to information local to the cabin is encapsulated into the spatial predicate *LREQ*, while the predicate *rcall* reflects the non-local considerations. Naturally, the remainder of the refinements are motivated by a desire to replace *rcall* with a local predicate while retaining the correct behavior of the elevator. A similar transformation reveals that the predicate *needed(f,δ)* can be computed using only information which is located at the floor *f*. Thus, the predicate *needed* for a particular floor is local to the cabin if the cabin is currently located at that floor. This transformation is shown in figure 13, which also serves to define the meaning of the spatial predicate *NEEDED(δ)#f*.

$$
\begin{aligned}
& needed(f,\delta) \\
\equiv\; & ((\delta=1 \wedge UP\#f) \vee (\delta=-1 \wedge DN\#f) \vee RQ(f)\#f \\
\equiv\; & [((\delta=1 \wedge UP) \vee (\delta=-1 \wedge DN) \vee RQ(f)]\#f \\
\equiv\; & NEEDED(\delta)\#f
\end{aligned}
$$

Fig. 13. Refining *needed* into a spatial predicate

Refinement 6: Limit references to remote floors. While there is no way to refine out references to remote floors, it is possible to separate knowledge about the remote floors from the mechanisms by which this knowledge is acquired. First, we observe that the use of *fromto* in *work_in* is existentially quantified over all floors in a particular direction; i.e., it is not necessary for the cabin to know the exact floor at which a call exists, only its relative direction. Second, because the elevator only needs to know about the presence of remote requests when making a service decision, it need only have accurate information when it actually commits to a decision. With these two observations in mind, we introduce a new spatial predicate *LCALL(δ)*, co-located with the cabin and containing the cabin's current view of *rcall* (see figure 14). In keeping with the first observation, the predicate *LCALL* is quantified over directions (F20). From the second observation, we require that *LCALL* exactly matches *rcall* before the elevator leaves a decision-making state, of which there are three: the elevator can be in the *arrive* state (F21), it can be stopped at a floor with the doors opened (F22), or it can be stopped with the doors closed (F23). Because we wish for the elevator to be able to respond instantly to calls when stopped with the doors closed, we require that at such times *LCALL* always exactly match *rcall*. The problem of guaranteeing that the elevator can accurately compute *rcall* is the motivation behind the remaining refinements.

F20: $\textbf{inv. } \mathsf{LCALL}(\delta)\#f \Rightarrow \mathsf{CABIN}\#f$

F21: $\mathsf{ARRIVE}\#f \textbf{ unless } \mathsf{ARRIVE}\#f \land (\mathsf{LCALL}(\delta)\#f \Leftrightarrow \mathrm{rcall}(f,\delta))$

F22: $[\mathsf{STOP} \land \mathsf{OPEN}]\#f \textbf{ unless } [\mathsf{STOP} \land \mathsf{OPEN}]\#f \land (\mathsf{LCALL}(\delta)\#f \Leftrightarrow \mathrm{rcall}(f,\delta))$

F23: $\textbf{inv. } [\mathsf{STOP} \land \mathsf{CLOSED}]\#f \Rightarrow (\mathsf{LCALL}(\delta)\#f \Leftrightarrow \mathrm{rcall}(f,\delta))$

Fig. 14. Constraints over the spatial predicate *LCALL*

This definition of *LCALL* allows us to replaces *rcall* with *LCALL* throughout the specification (figure 15). The correctness of this refinement follows immediately from observing that in F14.1.1.3, every time *LCALL* is referenced, the elevator is stopped with the doors are closed, and so F23 applies; and that in F14.1.1.1 and F14.1.1.2, the transition from the left-hand side to the right-hand side is constrained by F21 and F22. Additionally, at this time we perform the mechanical transformation of re-writing the progress properties using spatial predicates.

F14.1.1.1: $\mathrm{work\_in}(f,\delta') \land \mathrm{at}(f) \land \mathrm{arrive} \land \mathrm{dir}(\delta)$
$\quad\textbf{until}\ (\neg\mathrm{needed}(f,\delta) \land \mathrm{must\_go}(f,\delta) \land \mathrm{at}(f) \land \mathrm{depart} \land \mathrm{dir}(\delta)) \lor$
$\quad\quad (\mathrm{at}(f) \land \mathrm{stop} \land \mathrm{closed} \land \mathrm{dir}(\delta))$

F14.1.1.2: $\mathrm{work\_in}(f,\delta') \land \mathrm{at}(f) \land \mathrm{stop} \land \mathrm{open} \land \mathrm{dir}(\delta)$
$\quad\textbf{until}\ (\mathrm{must\_go}(f,\delta) \land \mathrm{at}(f) \land \mathrm{depart} \land \mathrm{dir}(\delta)) \lor$
$\quad\quad (\neg\mathrm{must\_go}(f,\delta) \land \mathrm{at}(f) \land \mathrm{stop} \land \mathrm{closed} \land \mathrm{dir}(-\delta))$

F14.1.1.3: $\mathrm{work\_in}(f,\delta') \land \mathrm{at}(f) \land \mathrm{stop} \land \mathrm{closed} \land \mathrm{dir}(\delta)$
$\quad\textbf{until}\ (\mathrm{at}(f) \land \mathrm{stop} \land \mathrm{dir}(\delta) \land \mathrm{open}) \lor$
$\quad\quad (\neg\mathrm{needed}(f,\delta) \land \mathrm{must\_go}(f,\delta) \land \mathrm{at}(f) \land \mathrm{depart} \land \mathrm{dir}(\delta)) \lor$
$\quad\quad (\neg\mathrm{needed}(f,\delta) \land \neg\mathrm{must\_go}(f,\delta) \land \mathrm{work\_in}(f,-\delta) \land$
$\quad\quad\quad \mathrm{at}(f) \land \mathrm{stop} \land \mathrm{closed} \land \mathrm{dir}(-\delta))$

are refined to

F14.1.1.1.1: $[(\mathsf{LREQ}(\delta') \lor \mathsf{LCALL}(\delta')) \land \mathsf{ARRIVE} \land \mathsf{DIR}(\delta)]\#f$
$\quad\textbf{until}\ [\neg\mathsf{NEEDED}(\delta) \land \mathsf{LCALL}(\delta) \land \mathsf{DEPART} \land \mathsf{DIR}(\delta)]\#f \lor$
$\quad\quad [\mathsf{STOP} \land \mathsf{CLOSED} \land \mathsf{DIR}(\delta)]\#f$

F14.1.1.2.1: $[(\mathsf{LREQ}(\delta') \lor \mathsf{LCALL}(\delta')) \land \mathsf{STOP} \land \mathsf{OPEN} \land \mathsf{DIR}(\delta)]\#f$
$\quad\textbf{until}\ [\mathsf{LCALL}(\delta) \land \mathsf{DEPART} \land \mathsf{DIR}(\delta)]\#f \lor$
$\quad\quad [\neg\mathsf{LCALL}(\delta) \land \mathsf{STOP} \land \mathsf{CLOSED} \land \mathsf{DIR}(-\delta)]\#f$

F14.1.1.3.1: $[(\mathsf{LREQ}(\delta') \lor \mathsf{LCALL}(\delta')) \land \mathsf{STOP} \land \mathsf{CLOSED} \land \mathsf{DIR}(\delta)]\#f$
$\quad\textbf{until}\ [\mathsf{STOP} \land \mathsf{OPEN} \land \mathsf{DIR}(\delta)]\#f \lor$
$\quad\quad [\neg\mathsf{NEEDED}(\delta) \land \mathsf{LCALL}(\delta) \land \mathsf{DEPART} \land \mathsf{DIR}(\delta)]\#f \lor$
$\quad\quad [\neg\mathsf{NEEDED}(\delta) \land \neg\mathsf{LCALL}(\delta) \land (\mathsf{LREQ}(-\delta) \lor \mathsf{LCALL}(-\delta)) \land$
$\quad\quad\quad \mathsf{STOP} \land \mathsf{CLOSED} \land \mathsf{DIR}(-\delta)]\#f$

Fig. 15. Limit references to remote floors

Refinement 7: Formulate the rules for updating data about remote floors. Without considering the mechanisms for updating *LCALL*, we can define the conditions under which the update ought to take place. This involves a refinement of F21 and F22, shown in figure 16. Here, we require that the elevator not exit the *arrive* (F21.1) or *open* (F22.1) states when there is a remote call but *LCALL* has not been correctly set. Further, we reset *LCALL* when entering either of these states, and in all other states, if *LCALL* is set, it must remain so (F24). Finally, *LCALL* cannot become set if *rcall* is not true (F25). Note that this does not constitute a refinement of F23. While we can show that this invariant is not invalidated in the transitions into the *stop-closed* states, we must retain the requirement that in this state, if *rcall* becomes true, then *LCALL* must also immediately become true. The proof that this is a correct refinement relies on the fact that we can prove that, in the *arrive* and *stopped-open* states, *LCALL(δ)#f* is true only if *rcall(f,δ)* is true.

F21: ARRIVE#f **unless** ARRIVE#f $\wedge$ (LCALL(δ)#f $\Leftrightarrow$ rcall(f,δ))

F22: [STOP $\wedge$ OPEN]#f **unless** [STOP $\wedge$ OPEN]#f $\wedge$ (LCALL(δ)#f $\Leftrightarrow$ rcall(f,δ))

are refined to

F21.1: ARRIVE#f $\wedge$ rcall(f,δ) **unless** [ARRIVE $\wedge$ LCALL(δ)]#f
F22.1: [STOP $\wedge$ OPEN]#f $\wedge$ rcall(f,δ) **unless** [STOP $\wedge$ OPEN $\wedge$ LCALL(δ)]#f
F24: LCALL(δ) **unless** (ARRIVE $\vee$ (STOP $\wedge$ OPEN)) $\wedge$ $\neg$LCALL(δ)
F25: $\neg$LCALL(δ)#f **unless** rcall(f,δ)

Fig. 16. Rules for updating data about remote floors

Refinement 8: Model the wireless communication. F21.1, F22.1, and F23 remain difficult to implement because they require the cabin to be able to test *rcall* directly. In this refinement, we introduce into the specification the mechanism for communicating between the floors and the cabin. We remind the reader that each floor having a pending call transmits on a periodic basis a radio signal, a *beep*; the cabin can detect the direction, but not exact location, of a beep. The behavior of the beep is described in figure 17. We model the beeps using a spatial predicate *BEEP(i)*, where the index *i* is the total number of beeps issued by the given floor since the elevator system began execution—because the actual implementation is not concerned with the index *i*, these unbounded values pose no difficulties in the generation of the program. A beep on floor *f* is modeled by a transition from a state in which *BEEP(i)#f* is true to a state in which *BEEP(i+1)#f* is true. The floors beep at regular intervals when there is a pending call (F28)—this should allow the cabin to know about the call. A beep is always associated with the pressing of a call button (F27) when the cabin is not already at the floor—this should help guarantee that *LCALL* and *rcall* agree when the elevator is in a decision-making state. Finally, a floor does not actually beep when the cabin is *on* the floor, allowing the cabin to distinguish between local and remote calls.

F26a: *INIT* $\Rightarrow$ BEEP(0)#f
F26b: **inv.** BEEP(i)#f $\Rightarrow$ floor(f) $\wedge$ i $\geq$ 0
F26c: **inv.** BEEP(i)#f $\wedge$ BEEP(j)#f $\Rightarrow$ i = j
F26d: **inv.** $\langle \exists$ i :: BEEP(i)#f $\rangle$

F27: [$\neg$CALL $\wedge$ BEEP(i)]#f
unless [CALL $\wedge$ BEEP(i+1) $\wedge$ $\neg$ON]#f $\vee$ [CALL $\wedge$ BEEP(i) $\wedge$ ON]#f
F28: [CALL $\wedge$ BEEP(i)]#f
unless [CALL $\wedge$ BEEP(i+1) $\wedge$ $\neg$ON]#f $\vee$ [$\neg$CALL $\wedge$ BEEP(i)]#f

Fig. 17. Definition of spatial predicate BEEP

Having characterized the behavior of the floors, we now turn to formalize the communication that takes place between the floors and the cabin. This refinement, shown in figure 18 simply requires that whenever there is a beep, then *LCALL* is correctly set to reflect it. There are three refined properties, each more-or-less reflecting one of the three decision-making states. F27.1 requires that, when a call is first made on a floor other than the one at which the cabin is currently located, then not only must the floor beep immediately, but *LCALL* must also be set at the same time. F28.1 and F28.2 require that, if the elevator is in a decision-making state with *LCALL* not correctly reflecting *rcall*, then the cabin cannot leave its current state until it has heard a beep, at which time *LCALL* will be correctly updated. F28.3 specifies that in all other states, if the floor beeps, the beep is reflected in *LCALL*, while disallowing beeps when the cabin is *on* the floor. Together, F28.1-F28.3 refine F28.

F21.1: $\mathrm{ARRIVE}\#f \wedge \mathrm{rcall}(f,\delta)$ **unless** $[\mathrm{ARRIVE} \wedge \mathrm{LCALL}(\delta)]\#f$

F22.1: $[\mathrm{STOP} \wedge \mathrm{OPEN}]\#f \wedge \mathrm{rcall}(f,\delta)$ **unless** $[\mathrm{STOP} \wedge \mathrm{OPEN} \wedge \mathrm{LCALL}(\delta)]\#f$

F23: **inv.** $[\mathrm{STOP} \wedge \mathrm{CLOSED}]\#f \Rightarrow (\mathrm{LCALL}(\delta)\#f \Leftrightarrow \mathrm{rcall}(f,\delta))$

F27: $[\neg\mathrm{CALL} \wedge \mathrm{BEEP}(i)]\#f$
unless $[\mathrm{CALL} \wedge \mathrm{BEEP}(i+1) \wedge \neg\mathrm{ON}]\#f \vee [\mathrm{CALL} \wedge \mathrm{BEEP}(i) \wedge \mathrm{ON}]\#f$

F28: $[\mathrm{CALL} \wedge \mathrm{BEEP}(i)]\#f$
unless $[\mathrm{CALL} \wedge \mathrm{BEEP}(i+1) \wedge \neg\mathrm{ON}]\#f \vee [\neg\mathrm{CALL} \wedge \mathrm{BEEP}(i)]\#f$

are refined to

F27.1: $[\neg\mathrm{CALL} \wedge \mathrm{BEEP}(i)]\#f$
unless $([\mathrm{CALL} \wedge \mathrm{BEEP}(i+1)]\#f \wedge \langle \exists f',\delta : \mathrm{fromto}(f',\delta,f) :: \mathrm{LCALL}(\delta)\#f' \rangle) \vee$
$[\mathrm{CALL} \wedge \mathrm{BEEP}(i) \wedge \mathrm{ON}]\#f$

F28.1: $[\mathrm{ARRIVE} \wedge \neg\mathrm{LCALL}(\delta)]\#f \wedge [\mathrm{CALL} \wedge \mathrm{BEEP}(i)]\#f' \wedge \mathrm{fromto}(f,\delta,f')$
unless $[\mathrm{ARRIVE} \wedge \mathrm{LCALL}(\delta)]\#f \wedge \mathrm{BEEP}(i+1)\#f'$

F28.2: $[\mathrm{STOP} \wedge \mathrm{OPEN} \wedge \neg\mathrm{LCALL}(\delta)]\#f \wedge [\mathrm{CALL} \wedge \mathrm{BEEP}(i)]\#f' \wedge \mathrm{fromto}(f,\delta,f')$
unless $[\mathrm{STOP} \wedge \mathrm{OPEN} \wedge \mathrm{LCALL}(\delta)]\#f \wedge \mathrm{BEEP}(i+1)\#f'$

F28.3: $[(\neg\mathrm{ARRIVE} \wedge \neg(\mathrm{STOP} \wedge \mathrm{OPEN})) \vee \mathrm{LCALL}(\delta)]\#f \wedge [\mathrm{CALL} \wedge \mathrm{BEEP}(i)]\#f' \wedge \mathrm{fromto}(f,\delta,f')$
unless $([\mathrm{CALL} \wedge \mathrm{BEEP}(i+1) \wedge \neg\mathrm{ON}]\#f' \wedge \langle \exists f'',\delta' : \mathrm{fromto}(f'',\delta',f') :: \mathrm{LCALL}(\delta')\#f'' \rangle) \vee$
$[\neg\mathrm{CALL} \wedge \mathrm{BEEP}(i)]\#f'$

Fig. 18. Model wireless communication

We now have in place a mechanism for guaranteeing that the cabin can determine whether there are calls to service in a particular direction without having to actually detect the presence of calls at remote floor. However, F28.1 and F28.2 require that the elevator not leave a decision-making state until it knows for sure that there is not going to be a beep. Formalizing the requirement and the solution in terms of timing constraints is the subject of the next section.

Discussion. Most often, the concept of space makes its way into the refinement process in terms of considerations regarding the allocation (mapping) of data across the underlying architecture. In a network, for instance, distant processes cannot share data but must exchange information via messages. Arguments about possible allocations may provide the rational for the refinement steps but are factored in the design process in a totally informal manner. The introduction of spatial qualifications allows us to give a more formal treatment of the allocation problem. While this may be beneficial in general, it becomes essential when dealing with mobile systems where the underlying architecture is in a state of continual flux due to changes in the location of and interconnection pattern among the components of the network.

Our approach for introducing the concept of place has its inspiration in positional logic [18] and was accommodated by a minimal amount of notational veneer. While the basic specification model did not change, additional discipline in the application of the refinement process is required. The first issue we faced was when and how to introduce space in the refinement process. We chose to discuss spatial qualifications as soon as we had to consider the notions of local versus remote access to information, i.e., the allocation became a relevant issue. Coupling invariants provided the mechanics for relating the location-free specifications to the location-explicit specifications. Because the relationships between the two kinds of specifications are invariant. one could make use of predicates which are not spatially qualified throughout the remainder of the refinements. This, however, would defeat the purpose of spatial qualifications and we chose to carry the spatial notation in the subsequent refinements. The notation allows us to reason formally about and to give syntactic form to several notions important in mobile computing: location-dependent properties, co-location of properties associated with moving objects, and local versus global data access. Finally, spatial qualification promises to play an important role in modeling certain aspects of wireless communication such as broadcast range, distance-dependent transmission rate, etc.

5. The Concept of Time

Time is represented by a distinguished variable T, not available to the program and referred to as the *current time*. In contrast to the way we introduced spatial qualifications, time is not an attribute of state components but an additional distinguished component of the state. In a given program state, the predicate $\&t$ holds if the current time is t (T1). The value of T is always positive (T2), unique (T3), and strictly increasing (T4). A formal definition of time in our logic is given in figure 19.

T1:	**inv.** $\&t \equiv (T = t)$
T2:	**inv.** $\langle \exists\, t : t \geq 0 :: \&t \rangle$
T3:	**inv.** $\&t \wedge \&t' \Rightarrow t = t'$
T4:	$\&t$ **unless** $\langle \exists\, t' : t' > t :: \&t' \rangle$

Fig. 19. Basic properties of time

In the specific case of the elevator problem, time appears as a constraint over the frequency with which beeps must be generated so as to guarantee that the cabin does not take certain decisions prematurely. Timing assumptions simply eliminate certain kinds of behaviors. One way of thinking about this approach is to assume that we have available a history of all the events (state transitions) happening in the system and the timestamps associated with each occurrence. Any ordering that violates the timing assumptions is thrown out. For a specific problem we do not need to consider all events, only a selected subset. The timing assumptions become invariant properties that make reference to the recorded timestamps.

The sequence of time values representing the successive, individual occurrences of some event is called a *timestamp*. A timestamp records the time values associated with each transition from false to true for some given predicate. If we have a timestamp **S** which is used to hold this sequence for some property $P(x)$, we write:

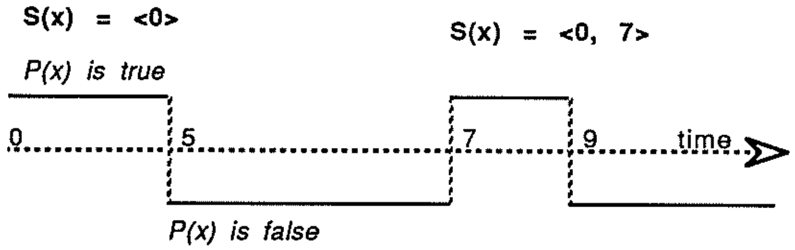
S.x **stamps** P(x)

which is defined by the specification found in figure 20. To distinguish timestamps from other predicates, we will use a bold, fixed-width font when writing the names of timestamps. The initial value of a timestamp is dependent upon the initial state of the property it records: if that property is initially true, then the timestamp starts with a single 0, otherwise the timestamp is initially empty (T6). The timestamp is changed only by appending to the sequence the value of T when the stamped property changes from false to true (T7); otherwise the timestamp remains unchanged (T8).

T6: $INIT \Rightarrow (P(x) \wedge S.x = [0]) \vee (\neg P(x) \wedge S.x = \varepsilon)$

T7: $\neg P(x) \wedge S.x{=}\sigma$ **unless** $P(x) \wedge \langle \exists\, t : \&t :: S.x{=}\sigma{\bullet}t \rangle$

T8: $P(x) \wedge S.x{=}\sigma$ **unless** $\neg P(x) \wedge S.x{=}\sigma$



with the added notational conventions

S[i] denotes the i'th element of the sequence
S[1] denotes the first element of the sequence
S$ denotes the last element of the sequence; S alone is used when the context is clear
t∈S denotes the presence of some element with value *t* in the sequence
|S| denotes the number of entries in the sequence S.

Fig. 20. Definition of a timestamp

A *timing constraint* is an invariant property relating one or more timestamps to the current system state. Timing constraints can be used to express restrictions on when an event takes place, on the order in which events occur, or on the separation (in time) between successive events. Since timing constraints are safety properties, they do not require that an event occur: rather they state restrictions on the times at which the event may occur. In our example, we will use timing constraints to express the requirements that the beeps occur at regular intervals of no more than *b* seconds, that the cabin doors remain open for at least *b* seconds, and that the elevator remain in the *arrive* state for at least *b* seconds. We can then show that properties F28.1 and F28.2 are satisfied in the presence of the timing properties.

Refinement 9: Guarantee timely broadcast reception. We first require that the interval between beeps be no more than *b* seconds if the elevator is not currently sitting on the floor. The timing constraint (F29, shown in figure 21) is written as an invariant relationship between the current system state, the current time, and the value of the timestamp. It requires that, in all cases, either less than *b* seconds have elapsed since the last beep, there is no call at the current floor, or the cabin is currently on the floor.

Timestamp definitions
BEEP.i.f **stamps** beep(i)#f

Timing constraint
F29: **inv.** $\&t \wedge$ BEEP(i)#f $\Rightarrow$ (BEEP.i.f$ + b > t) $\vee$ ¬CALL#f $\vee$ ON#f

Fig. 21. Timing constraint on beep

The two timing constraints on the behavior of the cabin are similar in form, and so we present them together. They require that the cabin remain in the *arrive* or *stop-open* states for at least *b* seconds; i.e., long enough to hear at least one beep. In both cases, the formal requirement (F30, F31, in figure 22) states that, if less than *b* seconds have elapsed since the elevator entered one of these two states, then the elevator is still in the given state.

Timestamp definitions

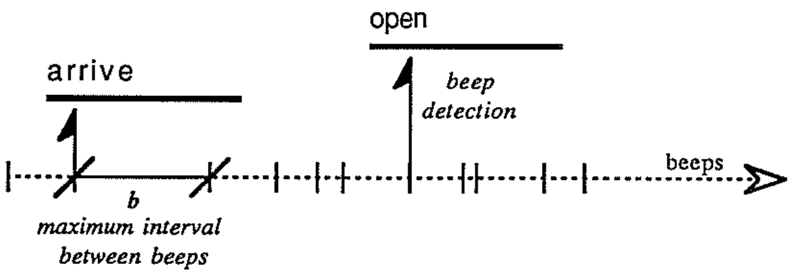
`OPEN` **stamps** open
`ARRIVE` **stamps** arrive

Timing constraint
F30: **inv.** $\&t \wedge$ `OPEN`$\$ + b > t \Rightarrow$ open
F31: **inv.** $\&t \wedge$ `ARRIVE`$\$ + b > t \Rightarrow$ arrive

Fig. 22. Timing constraint on the cabin

At this point, it is possible to show that these timing constraints (F29-F31) represent a refinement of F28.1 and F28.2. The refinement is shown in figure 23, which simply puts the pieces of the refinement together in one place.

F28.1: [ARRIVE $\wedge \neg$LCALL(δ)]#f $\wedge$ [CALL $\wedge$ BEEP(i)]#f' $\wedge$ fromto(f,δ,f')
unless [ARRIVE $\wedge$ LCALL(δ)]#f $\wedge$ BEEP(i+1)#f'
F28.2: [STOP $\wedge$ OPEN $\wedge \neg$LCALL(δ)]#f $\wedge$ [CALL $\wedge$ BEEP(i)]#f' $\wedge$ fromto(f,δ,f')
unless [STOP $\wedge$ OPEN $\wedge$ LCALL(δ)]#f $\wedge$ BEEP(i+1)#f'



are refined to

F29: **inv.** $\&t \wedge$ BEEP(i)#f $\Rightarrow$ (`BEEP.i.f`$\$ + b > t$) $\vee \neg$CALL#f $\vee$ ON#f
F30: **inv.** $\&t \wedge$ `OPEN`$\$ + b > t \Rightarrow$ open
F31: **inv.** $\&t \wedge$ `ARRIVE`$\$ + b > t \Rightarrow$ arrive
F32: BEEP(i)#f **unless** BEEP(i+1)#f $\wedge \langle \exists$ f',δ : fromto(f',δ,f) :: LCALL(δ)#f' $\rangle$

Fig. 23. Replace global knowledge with timing constraints

It should be noted that the exact means whereby the timing constraints will be implemented is of no concern to us here: they could represent engineering constraints on the design of the elevator hardware, they could be built-in as constraints on the scheduler for the software that is used to control the elevator, or they could be implemented in the actual control software using timers and flags. What matters here is that, with these constraints in place, we are able to guarantee the desired behavior.

There remains one open issue in the specification. F32 requires that programs which will ultimately be located at two distinct locations interact with each other. While we understand the mechanism whereby this communication will take place, i.e., the beeps, the continued presence of predicates which reference information having multiple locations is uncomfortable, especially since we know that the programs which perform the separate actions of beeping and setting *LCALL* will not be co-located. The mechanisms used to separate local transformations from remote communication will be the subject of the next section.

Discussion. Several real-time models that relate to our work include timed transition systems [7], timed automata [13], and event-based systems [22]. Our approach, however, was influenced primarily by Abadi and

Lamport [1] who argued successfully that real-time can be accommodated by existing formal systems in simple ways and demonstrated it using TLA. It is simplicity that we sought in our strategy for reasoning about real-time properties. The TLA approach relies on introducing a clock and absolute timers which prevent the clock from advancing prematurely and actions from executing too early. All timing properties assume the form of safety properties. Because UNITY specifications are given in terms of properties of states rather than actions, we turned to an analogous but less procedural approach. The clock action is no longer necessary, we just need to make sure that it is not disallowed by the specification. The deadlines set by timers in TLA are replaced by timestamps which establish baselines relative to which deadlines are specified; rather than scheduling actions, invariants impose restrictions on the permissible system states in the interval between the baseline and the deadline. All the timing constraints appearing in this section assume this special form and are used to express lower- and upper-bounds on the occurrence time for selected events. Many other kinds of constraints can be formulated by imposing any desired constraints among timestamps—our choice in this section was ultimately based how easy was to prove the correctness of the associated refinements.

Our approach has several attractive features. The mechanism for advancing time is left completely undefined which attests to one of the advantages of abstract non-operational specifications. Time can advance separate from or simultaneously with other state transitions—many models require time to advance between actions—and all causal dependencies are preserved by the timestamp recordings. Timestamps are associated only with events that are actually involved in timing constraints. Timing constraints are maintained separately from the rest of the specification and, as shown later, are incorporated without change in the abstract program generated at the end of the refinement process. Finally, the notation used to express timing constraints is intuitive, consistent with the UNITY style, and minimal.

6. Actions

The next logical step in a UNITY-style refinement is the generation of a concrete program, i.e., a set of conditional multiple-assignment statements. However, this would introduce both a new notation and a dependence upon the particular implementation strategy imposed by the computational model of UNITY. For these reasons, we prefer to focus our attention on developing yet another specification from which concrete statements could be trivially derived. We call such a specification an *abstract program*. Formally, an abstract program consists of a collection of actions along with data representation invariants which define the state space, timing constraints which capture timing assumptions about the program behavior, and spatial invariants which relate properties of the system to the location-independent initial specification. An *action*, in turn, consists of a name, a count of the number of times the corresponding statement executed, and a logic specification of the state transitions it entails.

In the abstract sense, an action always exists even though it may not always be enabled. When executed, an action which is local to one component may execute asynchronously with respect to actions in other components, or it may act synchronously with one or more other actions. In the latter case, the result is a global action (one involving multiple locations). The approach introduced in this section allows us to specify such location-dependent global interactions in a modular fashion by separating the specification of the local effects of actions from the specification of the conditions under which the synchronization occurs. The net result is a reduction in the complexity of the resulting abstract program when compared against a corresponding UNITY program—the number of statements in a mobile system would increase combinatorially as the number of possible interactions among actions increases. (For the purpose of this paper it is convenient to think of each independent action and each possible interaction as corresponding directly to a UNITY statement.) Beside wireless communication we also allow components to share data when they happen to be present at the same location. This provides a way of modeling interactions one might observe when a mobile component (e.g., a laptop) is directly connected via a cable to some other (possibly stationary) component. In such cases the corresponding action continues to be treated as being local.

Figure 24 summarizes the action notation used in this section. Each action has a name *A(x)#f* which includes instantiated parameters and a specific location. Associated with each action there is a counter *%A(x)#f* which is incremented only if the action takes a step. The purpose for the counter is to allow us to specify synchronization requirements among actions and it does not appear explicitly in the abstract program. The state transitions associated with a specific action are captured by the **takes-to** relation which is defined directly in terms of the UNITY **ensures**[3] property. By employing the **takes-to** relation as in definition A1a (*A(x)#f* ***takes*** *P(x)* ***to*** *Q(x)* ***when*** *R(x)*) we accomplish three things: we hide the use of the action counter; we make implicit the requirement that an action has a local effect by requiring that both the enabling condition, $P(x) \wedge R(x)$, and the consequence of the action, $Q(x)$, involve the same location as the action itself; and, finally, we account for the possibility that the component may exist in a reactive environment which may disable the action without it being executed. This last condition is necessary because the **ensures** property requires that there is a statement in the program which is guaranteed to make the desired transition if executed in a state in which the left-hand side is true, and that once the left-hand side becomes true, it must remain so until the transition is made. It is generally difficult to write **ensures** properties in a reactive system, since the enabling conditions are typically out of the control of the program itself (and are instead under the control of the environment). Thus, it is necessary to formulate (as we have done here), a property which allows for either the execution or the disabling of the statement to occur once the statement becomes enabled—in one case the counter is incremented while in the other it is not. If the environment cannot disable the action, $R(x)$ is always true and the **when** clause can be omitted.

The variant of **takes-to** appearing in A1b is less restrictive with regard to the effect of a particular action, allowing it to involve two distinct spatial locations: the one where the component is currently positioned and the one where the component moves to after the execution of the action. Without this we would not be able to express the notion of mobility, e.g., the elevator traveling from one floor to the next. The last definition, **tracks**, formalizes the meaning of one action, $A(x)$, executing in synchrony with one of several other distant actions $B_i(x_i)$. The synchronization is conditional on a global predicate S. In modeling wireless communication the predicate S may be used to capture the conditions under which the communication is feasible, e.g., the distance between the transmitter and the receiver is within some specified range.

It is our intent to express abstract programs in terms of these newly introduced relations, which are specific to the modeling of mobile systems, and without any explicit use of any **ensures** properties. The latter are to be used only to prove inference rules that relate **takes-to** and **tracks** to **unless** and **leads-to** properties used up to this point in the refinement process.

A0: $\%A(x)\#f{=}k$ **unless** $\%A(x)\#f{=}k{+}1$

A1a: $A(x)\#f$ **takes** $P(x)$ **to** $Q(x)$ **when** $R(x) \equiv$
$[P(x) \wedge \%A(x){=}k \wedge R(x)]\#f$
ensures $(Q(x)\#f \wedge \%A(x)\#f{=}k{+}1) \vee [P(x) \wedge \%A(x){=}k \wedge \neg R(x)]\#f$

A1b: $A(x)\#f$ **takes** $P(x)$ **to** $Q(x,f,f')$ **when** $R(x) \equiv$
$[P(x) \wedge \%A(x){=}k \wedge R(x)]\#f$
ensures $(Q(x,f,f') \wedge \%A(x)\#f{=}k{+}1) \vee [P(x) \wedge \%A(x){=}k \wedge \neg R(x)]\#f$

A2: $A(x_0)\#f_0$ **tracks** $\langle\, i : 1 \le i \le n :: B_i(x_i)\#f_i \,\rangle$ **when** $S(\ldots x_i,\ldots,f_i,\ldots) \equiv$
$S(\ldots x_i,\ldots,f_i,\ldots) \wedge \%A(x_0)\#f_0{=}k_0 \wedge \langle\, \forall\, i : 1 \le i \le n :: \%B_i(x_i)\#f_i{=}k_i \,\rangle$
unless $(\%A(x_0)\#f_0{=}k_0{+}1 \wedge \langle\, \exists\, i : 1 \le i \le n :: \%B_i(x_i)\#f_i{=}k_i{+}1 \,\rangle) \vee$
$\neg S(\ldots x_i,\ldots,f_i,\ldots)$

Fig. 24. Properties related to actions

Refinement 10: Introduce floor actions. We begin by defining the actions located at each floor. They are shown in figure 25. The actions *PressUp* and *PressDn* model the pressing of the call buttons. There are two properties for each, since the action must not only turn on the call lights (e.g., ¬UP **to** UP), but must also cause the floor to beep when the cabin is not on the floor (F33a and F33c). When the cabin is on the floor, however, turning on the call light can take place if the cabin is not currently stopped with its doors open servicing

[3] An **ensures** property is a progress property which is intended to describe a single, atomic transformation. Formally, **ensures** is an **unless** along with the requirement that there is at least one statement in the program which will always make the transformation if executed in a state satisfying the left-hand side.

the correct direction (F33b and F33d), but the beep is inhibited. Note that the actions access the current service direction, which is clearly data that is local to the cabin. This is possible because the cabin is located on the floor and we permit components that are co-located to share data. This type of data sharing is unique to mobile systems and its exploitation will likely be a central issue in the design of these systems. The last action associated with the floors is the independent beep that occurs every *b* seconds when there is a call on the floor (F34). The enabling condition reflects the requirement that a floor not beep when the cabin is on the floor.

F33a: PressUp#f **takes**
$\neg\mathsf{UP} \wedge 1 \le f < N \wedge \mathsf{BEEP}(i)$ **to** $\mathsf{UP} \wedge \mathsf{BEEP}(i+1)$ **when** $\neg\mathsf{ON}$

F33b: PressUp#f **takes**
$\neg\mathsf{UP} \wedge 1 \le f < N \wedge \mathsf{BEEP}(i)$ **to** $\mathsf{UP} \wedge \mathsf{BEEP}(i)$ **when** $\mathsf{ON} \wedge \neg(\mathsf{DIR}(1) \wedge \mathsf{OPEN})$

F33c: PressDn#f **takes**
$\neg\mathsf{DN} \wedge 1 < f \le N \wedge \mathsf{BEEP}(i)$ **to** $\mathsf{DN} \wedge \mathsf{BEEP}(i+1)$ **when** $\neg\mathsf{ON}$

F33d: PressDn#f **takes**
$\neg\mathsf{DN} \wedge 1 < f \le N \wedge \mathsf{BEEP}(i)$ **to** $\mathsf{DN} \wedge \mathsf{BEEP}(i)$ **when** $\mathsf{ON} \wedge \neg(\mathsf{DIR}(-1) \wedge \mathsf{OPEN})$

F34: Beep#f **takes** $\mathsf{CALL} \wedge \mathsf{BEEP}(i)$ **to** $\mathsf{BEEP}(i+1)$ **when** $\neg\mathsf{ON}$

Fig. 25. Floor actions

Refinement 11: Introduce cabin actions. The cabin is involved in four kinds of actions, depicted in figure 26. First, a request button may be turned on (F35). The request may be for any floor except for the one on which the doors are open—the reader should recall that all the predicates appearing in this action definition are spatially qualified by implication, i.e., ¬OPEN refers to ¬OPEN#f. As in the case of pressing the call buttons on the floor, there are some subtleties involved in modeling indeterminate environmental events as actions or program statements in a closed system. It may appear, for instance, that our abstract program requires each button to be pressed infinitely often in any infinite execution of the system (due to the fairness requirements of UNITY). In fact this is not the case since the action *PressRq(f')* might be selected for execution only when the doors are open on *f'*.

The second group of actions involves the control logic of the elevator (F36-42) which determines when doors are opened and closed, when to stop or go, and when to change direction. Since all these actions are disjoint, they could actually be grouped under a single name. They all involve local data and no change of position. The third kind of actions deal with the movement of the cabin from one floor to the next (F43). The specification for *Arrive#f* is deceivingly simple and compact. In reality, the change of location involves the invalidation of all the predicates that held at the previous location. This need not reflected in the specification we give here since we plan to carry over into the abstract program all the spatial invariants which were designed to co-locate the cabin and its state components. For the sake of brevity we will not actually repeat these invariants in this section. Neither will we repeat the timing assumptions associated with the door closing and the beeping frequency. We recognize, however, that in the future it may turn out to be advantageous to reformulate all such spatial and temporal predicates in terms of actions and their execution counters.

Finally, we can specify the action which monitors the beeps to set *LCALL*, and define the synchronization between the floors and the cabin which is necessary to implement F37. There are two actions to monitor the beeps, one which listens up (F44a) and one which listens down (F44b). Taken alone, these actions could set *LCALL* without any attending beep. To prevent this, we synchronize the actions with all the actions that can beep in the proper direction. Thus, *MonitorUp#f* is synchronized with the *PressUp*, *PressDn*, and *Beep* actions for all floors above *f*. As the cabin moves away from floor *f*, *MonitorUp#f* is disabled by the **when** clause. Although the semantics of our notation introduce different *MonitorUp* and *MonitorDn* actions for each floor, it is easier to think of the system as consisting of exactly one copy of each action, with the actions moving with the elevator cabin..

F35: PressRq(f')#f **takes** $\neg$RQ(f') **to** RQ(f') **when** $\neg$OPEN

F36: ArriveDepart#f **takes**
ARRIVE $\wedge$ DIR(δ) **to** DEPART $\wedge$ DIR(δ)
when $\neg$NEEDED(δ) $\wedge$ LCALL(δ)

F37: Stop#f **takes**
ARRIVE $\wedge$ DIR(δ) **to** STOP $\wedge$ CLOSED $\wedge$ DIR(δ)
when NEEDED(δ) $\vee$ $\neg$LCALL(δ)

F38: OpenDoors#f **takes**
STOP $\wedge$ CLOSED $\wedge$ DIR(δ)
to STOP $\wedge$ OPEN $\wedge$ DIR(δ) $\wedge$ $\neg$LCALL(δ) $\wedge$ $\neg$RQ(f) $\wedge$
((DIR(1) $\wedge$ $\neg$UP) $\vee$ (DIR(–1) $\wedge$ $\neg$DN))
when NEEDED(δ)

F39: CloseDoors#f **takes**
STOP $\wedge$ OPEN $\wedge$ DIR(δ) **to** STOP $\wedge$ CLOSED $\wedge$ DIR(–δ)
when $\neg$LCALL(δ) $\wedge$ (LCALL(–δ) $\vee$ LREQ(–δ))

F40: CloseDepart#f **takes**
STOP $\wedge$ CLOSED $\wedge$ DIR(δ) **to** DEPART $\wedge$ DIR(δ)
when $\neg$NEEDED(δ) $\wedge$ LCALL(δ)

F41: ChangeDir#f **takes**
STOP $\wedge$ CLOSED $\wedge$ DIR(δ) **to** STOP $\wedge$ CLOSED $\wedge$ DIR(–δ)
when $\neg$NEEDED(δ) $\wedge$ $\neg$LCALL(δ) $\wedge$ (LCALL(δ) $\vee$ LREQ(δ))

F42: OpenDepart#f **takes**
STOP $\wedge$ OPEN $\wedge$ DIR(δ) **to** DEPART $\wedge$ CLOSED $\wedge$ DIR(δ)
when LCALL(δ)

F43: Arrive#f **takes**
DEPART $\wedge$ DIR(δ) **to** [ARRIVE $\wedge$ DIR(δ) $\wedge$ $\neg$LCALL(δ)]#(f+δ)

F44a: MonitorUp#f **takes** true **to** LCALL(1) **when** CABIN
F44b: MonitorDn#f **takes** true **to** LCALL(-1) **when** CABIN

F45a: MonitorUp#f **tracks** $\langle$ f' : fromto(f,1,f') :: Beep#f', PressUp#f', PressDn#f' $\rangle$
when CABIN#f
F45b: MonitorDn#f **tracks** $\langle$ f' : fromto(f,–1,f') :: Beep#f', PressUp#f', PressDn#f' $\rangle$
when CABIN#f

Fig. 26. Cabin actions

Discussion. Our treatment of actions deliberately avoided any attempt to introduce new formalisms or theory. The objective was simply to show that an elegant treatment of actions and their interactions in mobile systems is possible within the specification notation provided by the UNITY logic. The proposed notation is novel only in its tailoring to the specific context of mobile systems and is designed as an aid in the specification process. It simply imposes a specific structuring of the specifications and there is some evidence [24] that such structuring is essential as we make the eventual transition to industrial grade problems.

Also, out of the desire to stay as closely as possible within the realm of UNITY, we chose to treat each independent action and each possible interaction among actions as corresponding directly to statements in UNITY. This approach, which has the advantage of simplifying the presentation, is made possible because the number of locations involved in the problem is finite. If this is not the case, one has to consider alternative views of the underlying computation. One option is to turn to Swarm [3], a model which employs the UNITY logic but makes use of tuples and transactions in place of variables and assignment statements. An action would correspond directly to a dynamically created Swarm transaction. Swarm also provides certain forms of dynamic synchrony useful in realizing the **tracks** relation [19].

While the association with spatial considerations is unique to our work, the concept of action (event) is well established in the literature. Without going into a comprehensive discussion of the different ways actions are treated by others, we will try to draw some comparisons with some closely related work. Our *A takes P to Q* formulation bears some resemblance to the proposed *P leads-to Q via A* relation [10] where *P* and *Q* are predicates and *A* is an action (event) name for a system transition. While **leads-to-via** defines a system property which may be derived from the definition of *A*, **takes-to** is employed as a mechanism for actually defining the action. Both relations impose constraints on other actions by requiring them to either preserve *P* or establish *Q*. However, the **when** clause is specifically designed to accommodate an indeterminate reactive environment and **leads-to-via** has no notion of location associated with its semantics. Another model in which actions play a central role is TLA [12]. In TLA a state transition is associated with one and only one action. By contrast, our actions are designed to allow for state transitions potentially involving multiple, simultaneous interference-free actions. Indeed, our approach of making the actions local and using synchronization to implement global actions guarantees that the actions are interference-free. This is expected to be an asset as we move on beyond this feasibility demonstration to formulating a general theory of mobile systems. We view modularity and dynamic synchrony to be central concepts in such a theory.

7. Conclusions

In this paper we have presented a methodology for specifying, designing, and reasoning about mobile systems. Our goal was to demonstrate that existing models of concurrency can accommodate mobile computing and to illustrate ways to accomplish this. We used the UNITY logic and a simple elevator control problem to make our point but we believe that the fundamental ideas have immediate applicability to other models as well as to a broad class of problems. While some of the notation is novel, its main role is to encapsulate and make obvious certain fundamental or pragmatically important concepts; all reasoning is carried out within the established UNITY logic. Prototypical usage of the notation has been illustrated in the context of designing a specific system.

The notions of space, time, and actions are central to our thinking about mobile systems. We used space to capture formally the notions of mobility, locality, and distant interactions. The desire to separate local effects from global interactions guided the refinement process and the notation for spatial qualification of predicates made obvious the places where further refinements were needed. Explicit reasoning about space is not common in the formal methods community and this is the first attempt to factor it in the UNITY context.

Time, on the other hand, is a major research concern for the distributed computing community, as evidenced by a voluminous literature on this topic. We used timing constraints to prove that programs that meet them are guaranteed to behave in certain desirable ways. The introduction of time into UNITY is not a first but our particular strategy is novel and proved to be both minimalist and simple to use. Another attractive feature is the fact that it succeeds in keeping functional and timing considerations separate both at the specification and program levels. Finally, although not shown in this paper, one can relate timing and spatial properties and factor velocity into the reasoning process.

Much work remains to be done with actions. The primary motivation for actions is to avoid generating a concrete UNITY program and allow for a much broader set of implementation alternatives. Actions are simply abstract specifications for program statements having local effects. An interesting new idea is the introduction of abstract synchronization constraints for modeling distant interactions among actions. This promises to be an elegant mechanism for reasoning about interactions among agents and for dealing with system modularization. However, further study is required to evaluate the generality of the approach with respect to various classes of common interactions among mobile agents. Also, since this paper dealt with closed system specifications, the applicability to open system specifications has not yet been established.

A definitive model for mobile computing is unlikely to emerge soon, and many proposals will be put forth and evaluated in the years to come. This paper argues for a pragmatic approach that minimizes formal development and centers on the application of existing knowledge to the design task. Even though our specifications and refinements are formal, our ultimate goal is to develop a design methodology that embodies the discipline of formal thinking but may be applied in a less tedious manner.

References

[1] Abadi, M., and Lamport, L., "An old-fashioned recipe for real-time," in *Lecture Notes in Computer Science*, J. W. d. Bakker, C. Huizing, W. P. Roever, G. Rosenberg, Eds., Springer-Verlag, vol. 600, pp. 1-27, 1991.

[2] Chandy, K. M., and Misra, J., *Parallel Program Design: A Foundation*, Addison-Wesley, New York, NY, 1988.

[3] Cunningham, H. C., and Roman, G.-C., "A UNITY-Style Programming Logic for a Shared Dataspace Language," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 3, pp. 365-376, 1990.

[4] de Bakker, J. W., de Roever, W.-P., and Rozenberg, G., Eds., *Stepwise Refinement of Distributed Systems* , 430, Springer-Verlag, Berlin, 1989.

[5] Dijkstra, E. D., *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[6] Gries, D., *The Science of Programming*, Springer-Verlag, New York, NY, 1981.

[7] Henzinger, T. A., Manna, Z., and Pnueli, A., "Timed Transition Systems," in *Real-Time: Theory in Practice*, J. W. De Bakker, C. Huizing, W. P. de Roever, G. Rozenberg, Eds., Springer-Verlag, Berlin, vol. 600, pp. 226-251, 1991.

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

[9] Knapp, E., "An Exercise in the Formal Derivation of Parallel Programs: Maximum Flows in Graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 2, pp. 203-223, 1990.

[10] Lam, S. S., and Shankar, A. U., "A Relational Notation for State Transition Systems," *IEEE Transactions on Software Engineering*, vol. 16, no. 7, pp. 755-775, 1990.

[11] Lamport, L., "Reasoning about nonatomic actions," *10th ACM Conference on Principles of Programming Languages*, ACM, pp. 28-37, 1983.

[12] Lamport, L., "The temporal logic of actions," Digital Equipment Corporation, Systems Research Center, 79, 1991.

[13] Lynch, N., and Vaandrager, F., "Forward and Backward Simulations for Timing-Based Systems," in *Real-Time: Theory in Practice*, J. W. De Bakker, C. Huizing, W. P. de Roever, G. Rozenberg, Eds., Springer-Verlag, Berlin, vol. 600, pp. 397-446, 1991.

[14] Milner, R., *A Calculus for Communicating Systems*, G. Goos, J. Hartmanis, Eds., Lecture Notes in Computer Science, Springer-Verlag, New York, NY, vol. 92, 1980.

[15] Morgan, C. C., "The Specification Statement," *ACM TOPLAS*, vol. 10, pp. 403-419, 1988.

[16] Morris, J. M., "Laws of Data Refinement," *Acta Informatica*, vol. 26, pp. 287-308, 1989.

[17] Owicki, S., and Lamport, L., "Proving Liveness Properties of Concurrent Programs," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 455-495, 1982.

[18] Rescher, N., and Urquhart, A., *Temporal Logic*, Springer-Verlag, New York, 1971.

[19] Roman, G.-C., and Cunningham, H. C., "Reasoning about Synchronic Groups," in *Research Directions in High-Level Parallel Programming Languages*, J. P. Banâtre, D. L. Métayer, Eds., Springer-Verlag, New York, NY, vol. 574, pp. 21-38, 1992.

[20] Roman, G.-C., Gamble, R. F., and Ball, W. E., "Formal Derivation of Rule-Based Programs," *IEEE Transactions on Software Engineering*, vol. 19, no. 3, pp. 227-296, 1993.

[21] Roman, G.-C., and Wilcox, C. D., "Architecture-Directed Refinement," *IEEE Transactions on Software Engineering*, vol. 20, no. 4, pp. 239-258, 1994.

[22] Shankar, A. U., and Lam, S. S., "Time-dependent distributed systems: proving safety, liveness and real-time properties," *Distributed Computing*, vol. 2, pp. 61-79, 1987.

[23] Staskauskas, M., "A Formal Specification and Design of a Distributed Electronic Funds-Transfer Network," *IEEE Transactions on Computers*, vol. 37, no. 12, pp. 1515-1528, 1988.

[24] Staskauskas, M., "Formal Derivation of Concurrent Programs: An example from Industry," *IEEE Transactions on Software Engineering*, vol. 19, no. 5, pp. 503-528, 1993.

Appendix 1. Proofs

Notes on proving leads-to

The transitivity of **leads-to** is typically the center of any proof of a progress refinement. Thus, when proving **leads-to** properties, we format the proof as a series of steps, each making use of either an implication or a **leads-to** property, with the resulting proof following from the transitivity of **leads-to** and the fact that implication is **leads-to.**

Notes on proving unless

In the proofs that follow, **unless** properties are, for the most part, proven using the conjunction or disjunction rules for **unless**. However, in a number of cases, the proof takes on a form that is superficially similar to our proofs for **leads-to**, this in spite of the fact that **unless** is not transitive. That is, we start the with lhs of the property to be proven, massage it into a form that matches the lhs of some refined property, use the **unless** from that refined property to get its rhs, which we then massage into the rhs of the original property. This form of proof relies on two properties of **unless** which are not explicitly mentioned in these proofs. The first property is

$$\frac{p' \equiv p,\ p\ \mathbf{unless}\ q}{p'\ \mathbf{unless}\ q}$$

which allows us to massage the lhs of an **unless** using equivalence relations. The second property is the consequence weakening rule for **unless**, which allows us to massage the rhs of an **unless** using implications. Thus, our "transitive" proofs for **unless** start with the lhs of the property to be proven, massage it using only equivalance properties into the lhs of a refined **unless** property, the rhs of which is then massaged via implications into the form of the rhs of the original property. That is, we utilize the proof rule:

$$\frac{p' \equiv p,\ p\ \mathbf{unless}\ q,\ q \Rightarrow q'}{p'\ \mathbf{unless}\ q'}$$

when proving **unless**.

Some Lemmas

needed remains true across most movements.

L3.1: $\text{needed}(f,\delta)$ **unless** $\text{at}(f) \wedge \text{stop} \wedge \text{dir}(\delta) \wedge \text{open}$
L3.2: $\text{needed}(f,\delta) \wedge \text{at}(f') \wedge \text{depart} \wedge \text{dir}(\delta')$ **unless** $\text{needed}(f,\delta) \wedge \text{at}(f'+\delta') \wedge \text{arrive}$
L3.3: $\text{needed}(f,\delta) \wedge \text{at}(f') \wedge \text{arrive}$ **unless** $\text{needed}(f,\delta) \wedge \text{at}(f) \wedge ((\text{stop} \wedge \text{closed}) \vee \text{depart})$
L3.4: $\text{needed}(f,\delta) \wedge \text{at}(f') \wedge \text{stop} \wedge f \neq f'$ **unless** $\text{needed}(f,\delta) \wedge \text{at}(f') \wedge \text{depart}$

Similar lemmas hold for *call* and *rcall.*

Refinement 1

F14 $\text{needed}(f,\delta)$
$\Rightarrow$ { F3 }
$\text{needed}(f,\delta) \wedge \langle \exists f' :: \text{on}(f') \vee \text{depart}(f') \rangle$
$\Rightarrow$ { F4 }
$\text{needed}(f,\delta) \wedge (\langle \exists f',\delta' :: \text{on}(f') \wedge \text{dir}(\delta') \rangle \vee \langle \exists f',\delta' :: \text{depart}(f') \wedge \text{dir}(\delta') \rangle)$
$\Rightarrow$ { disjunction }
$\text{needed}(f,\delta) \wedge$
$(\langle \exists f',\delta' :: \text{on}(f') \wedge \text{dir}(\delta') \wedge (f \neq f' \vee \delta \neq \delta') \rangle \vee$ (1)
$(\text{on}(f) \wedge \text{dir}(\delta)) \vee$ (2)
$\langle \exists f',\delta' :: \text{depart}(f') \wedge \text{dir}(\delta') \rangle)$ (3)

(1) $\text{needed}(f,\delta) \wedge (\langle \exists f',\delta' :: \text{on}(f') \wedge \text{dir}(\delta') \wedge (f \neq f' \vee \delta \neq \delta') \rangle$

$\mapsto$ { F14.1 }

$\text{needed}(f,\delta) \wedge \text{on}(f) \wedge \text{dir}(\delta)$

$\mapsto$ { F14.2 }

$\text{stopped}(f) \wedge \text{dir}(\delta) \wedge \text{open}$ QED 1

(2) $\text{needed}(f,\delta) \wedge \text{on}(f) \wedge \text{dir}(\delta)$

$\mapsto$ { F14.2 }

$\text{stopped}(f) \wedge \text{dir}(\delta) \wedge \text{open}$ QED 2

(3) $\text{needed}(f,\delta) \wedge \langle \exists f',\delta' :: \text{depart}(f') \wedge \text{dir}(\delta') \rangle$

$\mapsto$ { L3.2, F14.3 }

$\text{needed}(f,\delta) \wedge \text{arrive}(f'+\delta')$

$\Rightarrow$ { definition *on* }

$\text{needed}(f,\delta) \wedge \text{on}(f'+\delta')$

$\Rightarrow$ { disjunction }

$\text{needed}(f,\delta) \wedge ((\text{on}(f'+\delta') \wedge f'+\delta'=f \wedge \delta'=\delta) \vee (\text{on}(f'+\delta') \wedge (f'+\delta' \neq f \vee \delta' \neq \delta)))$

$\mapsto$ { proofs of part 1, and 2 }

$\text{stopped}(f) \wedge \text{dir}(\delta) \wedge \text{open}$ QED 3, F14

Refinement 2

The proof is immediate from the induction principle for **leads-to.**

Refinement 3

We introduce the following metric, which measures distance not only between floors, but within floors as well. The proof follows immediately from the observation that every transition in F14 decreases this new metric, and from the induction principle for **leads-to.**

$$\text{dist2}(s, (j,\delta'),(i,\delta)) = \begin{cases} 4*\text{dist}((j,\delta'),(i,\delta)) & \text{if } s = \text{stop} \wedge \text{open} \\ 4*\text{dist}((j,\delta'),(i,\delta))+1 & \text{if } s = \text{stop} \wedge \text{closed} \\ 4*\text{dist}((j,\delta'),(i,\delta))+2 & \text{if } s = \text{arrive} \\ 4*\text{dist}((j,\delta'),(i,\delta))-1 & \text{if } s = \text{depart} \wedge (i \neq j \vee \delta \neq \delta') \\ 2(N-1)-1 & \text{if } s = \text{depart} \wedge i=j \wedge \delta=\delta' \end{cases}$$

Refinement 4

No proof is required.

Refinement 5

Refinement of *work_in*.

$\mathrm{at}(f) \wedge \mathrm{work\_in}(f,\delta)$

$\equiv$

$\mathrm{CABIN}\#f \wedge (\mathrm{needed}(f,\delta) \vee \langle \exists f' : \mathrm{fromto}(f,\delta,f') :: \mathrm{RQ}(f')\#f \vee \mathrm{CALL}\#f' \rangle$

$\equiv$

$\mathrm{CABIN}\#f \wedge$
$(\mathrm{rq}(f) \vee (\delta{=}1 \wedge \mathrm{up}(f)) \vee (\delta{=}{-1} \wedge \mathrm{dn}(f)) \vee \langle \exists f' : \mathrm{fromto}(f,\delta,f') :: \mathrm{RQ}(f')\#f \rangle \vee$
$\langle \exists f' : \mathrm{fromto}(f,\delta,f') :: \mathrm{CALL}\#f' \rangle)$

$\equiv$

$\mathrm{CABIN}\#f \wedge$
$((\delta{=}1 \wedge \mathrm{UP}\#f) \vee (\delta{=}{-1} \wedge \mathrm{DN}\#f) \vee \langle \exists f' : \mathrm{fromto}(f,\delta,f') \vee f{=}f' :: \mathrm{RQ}(f')\#f \rangle \vee$
$\langle \exists f' : \mathrm{fromto}(f,\delta,f') :: \mathrm{CALL}\#f' \rangle)$

$\equiv$

$\mathrm{CABIN}\#f \wedge$
$(\,[(\delta{=}1 \wedge \mathrm{UP}) \vee (\delta{=}{-1} \wedge \mathrm{DN}) \vee \langle \exists f' : \mathrm{fromto}(f,\delta,f') \vee f{=}f' :: \mathrm{RQ}(f') \rangle\,]\#f \vee$
$\langle \exists f' : \mathrm{fromto}(f,\delta,f') :: \mathrm{CALL}\#f' \rangle)$

$\equiv$

$[\mathrm{CABIN} \wedge ((\delta{=}1 \wedge \mathrm{UP}) \vee (\delta{=}{-1} \wedge \mathrm{DN}) \vee \langle \exists f' : \mathrm{fromto}(f,\delta,f') \vee f{=}f' :: \mathrm{RQ}(f') \rangle)\,]\#f$
$\vee$
$(\mathrm{CABIN}\#f \wedge \langle \exists f' : \mathrm{fromto}(f,\delta,f') :: \mathrm{CALL}\#f' \rangle)$

$\equiv$

$\mathrm{LREQ}(\delta)\#f \vee \mathrm{rcall}(f,\delta)$

Refinement 6

F14.1.1.1

$\mathrm{work\_in}(f,\delta') \wedge \mathrm{at}(f) \wedge \mathrm{arrive} \wedge \mathrm{dir}(\delta)$
$\equiv$ { definition of *work_in* }
$(\mathrm{LREQ}(\delta')\#f \vee \mathrm{rcall}(f,\delta')) \wedge \mathrm{at}(f) \wedge \mathrm{arrive} \wedge \mathrm{dir}(\delta)$
$\equiv$ { math }
$(\mathrm{LREQ}(\delta')\#f \wedge \mathrm{at}(f) \wedge \mathrm{arrive} \wedge \mathrm{dir}(\delta)) \vee$ (1)
$(\mathrm{rcall}(f,\delta') \wedge \mathrm{at}(f) \wedge \mathrm{arrive} \wedge \mathrm{dir}(\delta))$ (2)

1 $\mathrm{LREQ}(\delta')\#f \wedge \mathrm{at}(f) \wedge \mathrm{arrive} \wedge \mathrm{dir}(\delta)$
$\equiv$ { definitions of spatial predicates }
$[\mathrm{LREQ}(\delta') \wedge \mathrm{ARRIVE} \wedge \mathrm{DIR}(\delta)]\#f$
unless { F14.1.1.1.1 }
$[\neg\mathrm{NEEDED}(\delta) \wedge \mathrm{LCALL}(\delta) \wedge \mathrm{DEPART} \wedge \mathrm{DIR}(\delta)]\#f \vee [\mathrm{STOP} \wedge \mathrm{CLOSED} \wedge \mathrm{DIR}(\delta)]\#f$
$\equiv$ { definitions of spatial predicates }
$(\neg\mathrm{needed}(f,\delta) \wedge \mathrm{must\_go}(f,\delta) \wedge \mathrm{at}(f) \wedge \mathrm{depart} \wedge \mathrm{dir}(\delta)) \vee (\mathrm{at}(f) \wedge \mathrm{stop} \wedge \mathrm{closed} \wedge \mathrm{dir}(\delta))$ QED 1

2 $\mathrm{rcall}(f,\delta') \wedge \mathrm{at}(f) \wedge \mathrm{arrive} \wedge \mathrm{dir}(\delta)$
$\equiv$ { definitions of spatial predicates }
$\mathrm{rcall}(f,\delta') \wedge [\mathrm{ARRIVE} \wedge \mathrm{DIR}(\delta)]\#f$
unless { conjunction with F21 }
$\mathrm{rcall}(f,\delta') \wedge [\mathrm{ARRIVE} \wedge \mathrm{DIR}(\delta)]\#f \wedge (\mathrm{LCALL}(\delta)\#f \Leftrightarrow \mathrm{rcall}(f,\delta))$
$\equiv$ { math }
$[\mathrm{ARRIVE} \wedge \mathrm{DIR}(\delta) \wedge \mathrm{LCALL}(\delta')]\#f$
unless { F14.1.1.1.1. and cancellation }
$[\neg\mathrm{NEEDED}(\delta) \wedge \mathrm{LCALL}(\delta) \wedge \mathrm{DEPART} \wedge \mathrm{DIR}(\delta)]\#f \vee [\mathrm{STOP} \wedge \mathrm{CLOSED} \wedge \mathrm{DIR}(\delta)]\#f$
$\equiv$ { definition of spatial predicates }
$(\neg\mathrm{needed}(f,\delta) \wedge \mathrm{must\_go}(f,\delta) \wedge \mathrm{at}(f) \wedge \mathrm{depart} \wedge \mathrm{dir}(\delta)) \vee$ QED 2, F14.1.1.1
$(\mathrm{at}(f) \wedge \mathrm{stop} \wedge \mathrm{closed} \wedge \mathrm{dir}(\delta))$

F14.1.1.2

$work\_in(f,\delta') \wedge at(f) \wedge stop \wedge open \wedge dir(\delta)$

$\equiv$ { definition of *work_in* }

$(LREQ(\delta')\#f \vee rcall(f,\bar{\delta}')) \wedge at(f) \wedge stop \wedge open \wedge dir(\delta)$

$\equiv$ { math }

$(LREQ(\delta')\#f \wedge at(f) \wedge stop \wedge open \wedge dir(\delta)) \vee$ (1)

$(rcall(f,\delta') \wedge at(f) \wedge stop \wedge open \wedge dir(\delta))$ (2)

1 $LREQ(\delta')\#f \wedge at(f) \wedge stop \wedge open \wedge dir(\delta)$

$\equiv$ { definitions of spatial predicates }

$[LREQ(\delta') \wedge STOP \wedge OPEN \wedge DIR(\delta)]\#f$

unless { F14.1.1.2.1 }

$[LCALL(\delta) \wedge DEPART \wedge DIR(\delta)]\#f \vee [\neg LCALL(\delta) \wedge STOP \wedge CLOSED \wedge DIR(-\delta)]\#f$

$\equiv$ { definition of spatial predicates }

$(must\_go(f,\delta) \wedge at(f) \wedge depart \wedge dir(\delta)) \vee (\neg must\_go(f,\delta) \wedge at(f) \wedge stop \wedge closed \wedge dir(-\delta))$ QED 1

2 $rcall(f,\delta') \wedge at(f) \wedge stop \wedge open \wedge dir(\delta)$

$\equiv$ { definitions of spatial predicates }

$rcall(f,\delta') \wedge [STOP \wedge OPEN \wedge DIR(\delta)]\#f$

unless { conjunction with F21 }

$rcall(f,\delta') \wedge [STOP \wedge OPEN \wedge DIR(\delta)]\#f \wedge (LCALL(\delta)\#f \Leftrightarrow rcall(f,\delta))$

$\equiv$ { math }

$[STOP \wedge OPEN \wedge DIR(\delta) \wedge LCALL(\delta')]\#f$

unless { F14.1.1.1.1. and cancellation }

$[\neg NEEDED(\delta) \wedge LCALL(\delta) \wedge DEPART \wedge DIR(\delta)]\#f \vee [STOP \wedge CLOSED \wedge DIR(-\delta)]\#f$

$\equiv$ { definition of spatial predicates }

$(\neg needed(f,\delta) \wedge must\_go(f,\delta) \wedge at(f) \wedge depart \wedge dir(\delta)) \vee$ QED 2, F14.1.1.2

$(at(f) \wedge stop \wedge closed \wedge dir(-\delta))$

F14.1.1.3

$work\_in(f,\delta') \wedge at(f) \wedge stop \wedge closed \wedge dir(\delta)$

$\equiv$ { definition of *work_in* }

$(LREQ(\delta')\#f \vee rcall(f,\bar{\delta}')) \wedge at(f) \wedge stop \wedge closed \wedge dir(\delta)$

$\equiv$ { definition of spatial predicates }

$(LREQ(\delta')\#f \vee rcall(f,\delta')) \wedge [STOP \wedge CLOSED \wedge DIR(\delta)]\#f$

$\equiv$ { F23 }

$(LREQ(\delta')\#f \vee LCALL(\delta')\#f) \wedge [STOP \wedge CLOSED \wedge DIR(\delta)]\#f$

unless { F14.1.1.3.1 }

$[STOP \wedge OPEN \wedge DIR(\delta)]\#f \vee$

$[\neg NEEDED(\delta) \wedge LCALL(\delta) \wedge DEPART \wedge DIR(\delta)]\#f \vee$

$[\neg NEEDED(\delta) \wedge \neg LCALL(\delta) \wedge (LREQ(-\delta) \vee LCALL(-\delta)) \wedge STOP \wedge CLOSED \wedge DIR(-\delta)]\#f$

$\equiv$ { definition of spatial predicates }

$(at(f) \wedge stop \wedge dir(\delta) \wedge open) \vee$ QED F14.1.1.3

$(\neg needed(f,\delta) \wedge must\_go(f,\delta) \wedge at(f) \wedge depart \wedge dir(\delta)) \vee$

$(\neg needed(f,\delta) \wedge \neg must\_go(f,\delta) \wedge work\_in(f,-\delta) \wedge at(f) \wedge stop \wedge closed \wedge dir(-\delta))$

Refinement 7

Key invariant

I7 **inv.** $[ARRIVE \vee (STOP \wedge OPEN)]\#f \Rightarrow (LCALL(\delta)\#f \Rightarrow rcall(f,\delta))$

The proof is by contradiction. We present here the proof for the *arrive* state, the proof for the *stopped-open* state is identical. Assume we have:

$ARRIVE\#f \wedge LCALL(\delta)\#f \wedge \neg rcall(f,\delta)$

Then there are three possibilities: either *LCALL* was true immediately upon entering the *arrive* state, or it became true while in the *arrive* state, or *rcall* became false. The first is not possible by F21.1, since *LCALL* is reset upon entering the *arrive* state. The second is impossible by F25, since *LCALL* can only become true is *rcall* is true. Finally, the third is impossible, since *rcall(f,δ)* can only go from true to false when a request is serviced, and requests are not serviced in the *arrive* state, by F14. We have a contradiction, so the invariant holds.

Some lemmas

We need to show that *rcall* remains true across the **unless** in both F21.1 and F22.1. The proofs follow from F14 and the definition of *rcall.*

L7.1 $\text{ARRIVE\#f} \wedge \text{rcall}(f,\delta)$ **unless** $\neg\text{ARRIVE\#f}$

L7.2 $[\text{STOP} \wedge \text{OPEN}]\text{\#f} \wedge \text{rcall}(f,\delta)$ **unless** $\neg[\text{STOP} \wedge \text{OPEN}]\text{\#f}$

F21

$\text{ARRIVE\#f} \wedge \neg(\text{LCALL}(\delta)\text{\#f} \Leftrightarrow \text{rcall}(f,\delta))$

$\Rightarrow$ { math }

$(\text{ARRIVE\#f} \wedge \text{LCALL}(\delta)\text{\#f} \wedge \neg\text{rcall}(f,\delta)) \vee (\text{ARRIVE\#f} \wedge \neg\text{LCALL}(\delta)\text{\#f} \wedge \text{rcall}(f,\delta))$

$\Rightarrow$ { I7, first disjunct is false }

$\text{ARRIVE\#f} \wedge \neg\text{LCALL}(\delta)\text{\#f} \wedge \text{rcall}(f,\delta)$

$\Rightarrow$ { consequence weakening }

$\text{ARRIVE\#f} \wedge \text{rcall}(f,\delta)$

unless { F21.1, L7.1 }

$[\text{ARRIVE} \wedge \text{LCALL}(\delta)]\text{\#f} \wedge \text{rcall}(f,\delta)$

The other possibility, that both *LCALL* and *rcall* are false, follows immediately from I7, so we have the if-and-only-if. The proof for F22 is identical.

Refinement 8

F21.1

$[\text{ARRIVE} \wedge \neg\text{LCALL}(\delta)]\text{\#f} \wedge \text{rcall}(f,\delta)$

$\equiv$ { definition of *rcall* }

$[\text{ARRIVE} \wedge \neg\text{LCALL}(\delta)]\text{\#f} \wedge \langle \exists f' : \text{fromto}(f,\delta,f') :: \text{CALL\#f}' \rangle$

$\equiv$ { F26d }

$[\text{ARRIVE} \wedge \neg\text{LCALL}(\delta)]\text{\#f} \wedge \langle \exists f',i : \text{fromto}(f,\delta,f') :: [\text{CALL} \wedge \text{BEEP}(i)]\text{\#f}' \rangle$

unless { F28.1 }

$[\text{ARRIVE} \wedge \text{LCALL}(\delta)]\text{\#f} \wedge \text{BEEP}(i+1)\text{\#f}'$

$\Rightarrow$ { consequence weakening }

$[\text{ARRIVE} \wedge \text{LCALL}(\delta)]\text{\#f}$ QED F21.1

F22.1

$[\text{STOP} \wedge \text{OPEN}]\text{\#f} \wedge \text{rcall}(f,\delta)$

$\equiv$ { definition of *rcall* }

$[\text{STOP} \wedge \text{OPEN} \wedge \neg\text{LCALL}(\delta)]\text{\#f} \wedge \langle \exists f' : \text{fromto}(f,\delta,f') :: \text{CALL\#f}' \rangle$

$\equiv$ { F26d }

$[\text{STOP} \wedge \text{OPEN} \wedge \neg\text{LCALL}(\delta)]\text{\#f} \wedge \langle \exists f',i : \text{fromto}(f,\delta,f') :: [\text{CALL} \wedge \text{BEEP}(i)]\text{\#f}' \rangle$

unless {F28.2 }

$[\text{STOP} \wedge \text{OPEN} \wedge \text{LCALL}(\delta)]\text{\#f} \wedge \text{BEEP}(i+1)\text{\#f}'$

$\Rightarrow$ { consequence weakening }

$[\text{STOP} \wedge \text{OPEN} \wedge \text{LCALL}(\delta)]\text{\#f}$ QED F22.1

F23

$$[STOP \wedge CLOSED]\#f \Rightarrow (LCALL(\delta)\#f \Leftrightarrow rcall(f,\delta))$$

The proof, which is by contradiction, goes as follows. Assume that we have:

$$[STOP \wedge CLOSED]\#f \wedge ((\neg LCALL(\delta)\#f \wedge rcall(f,\delta)) \vee (LCALL(\delta)\#f \wedge \neg rcall(f,\delta))$$

In order for the first disjunct to be true, either the system entered the *stopped-closed* state with *rcall* and not *LCALL*, or *rcall* became true in this state without *LCALL*also becoming true. The first option is impossible by F21.1 and F24, which require that *LCALL* and *rcall* match when exiting the *arrive* state, and the second is disallowed by F28.3, which guarantees that when *rcall* becomes true, then *LCALL* will simultaneously become true. If the second disjunct is true, then it must be the case that *LCALL* became true after entering the *stopped-closed* state, since again F21.1 and F24 guarantee that upon entering the *stopped-closed* state, *rcall* and *LCALL* match. But by F25, if *LCALL* is false, it will remain so until *rcall* becomes true, so the second disjunct can never be true, giving us our contradiction.

F27

$[\neg CALL \wedge BEEP(i)]\#f$

unless { F27.1 }

$([CALL \wedge BEEP(i+1)]\#f \wedge \langle \exists f',\delta : fromto(f',\delta,f) :: LCALL(\delta)\#f' \rangle) \vee [CALL \wedge BEEP(i) \wedge ON]\#f$

$\Rightarrow$ { consequence weakening, definition of *fromto* }

$[CALL \wedge BEEP(i+1) \wedge \neg ON]\#f \vee [CALL \wedge BEEP(i) \wedge ON]\#f$ QED F27

F28

$F28.1 \wedge F28.2 \wedge F28.3$

$\Rightarrow$ { simple disjunction }

$[CALL \wedge BEEP(i)]\#f' \wedge fromto(f,\delta,f')$

unless $([ARRIVE \wedge LCALL(\delta)]\#f \wedge BEEP(i+1)\#f') \vee$

$([STOP \wedge OPEN \wedge LCALL(\delta)]\#f \wedge BEEP(i+1)\#f') \vee$

$([CALL \wedge BEEP(i+1) \wedge \neg ON]\#f' \wedge \langle \exists f'',\delta' : fromto(f'',\delta',f') :: LCALL(\delta')\#f'') \vee$

$[\neg CALL \wedge BEEP(i)]\#f'$

$\Rightarrow$ { by F14, calls remain until the cabin stops at the floor }

$[CALL \wedge BEEP(i)]\#f' \wedge fromto(f,\delta,f')$

unless $([ARRIVE \wedge LCALL(\delta)]\#f \wedge [CALL \wedge BEEP(i+1)]\#f') \vee$

$([STOP \wedge OPEN \wedge LCALL(\delta)]\#f \wedge [CALL \wedge BEEP(i+1)]\#f') \vee$

$([CALL \wedge BEEP(i+1) \wedge \neg ON]\#f' \wedge \langle \exists f'',\delta' : fromto(f'',\delta',f') :: LCALL(\delta')\#f'') \vee$

$[\neg CALL \wedge BEEP(i)]\#f'$

$\equiv$ { $f \neq f'$, F2b }

$[CALL \wedge BEEP(i)]\#f' \wedge fromto(f,\delta,f')$

unless $([ARRIVE \wedge LCALL(\delta)]\#f \wedge [CALL \wedge BEEP(i+1) \wedge \neg ON]\#f') \vee$

$([STOP \wedge OPEN \wedge LCALL(\delta)]\#f \wedge [CALL \wedge BEEP(i+1) \wedge \neg ON]\#f') \vee$

$([CALL \wedge BEEP(i+1) \wedge \neg ON]\#f' \wedge \langle \exists f'',\delta' : fromto(f'',\delta',f') :: LCALL(\delta')\#f'') \vee$

$[\neg CALL \wedge BEEP(i)]\#f'$

$\Rightarrow$ { consequence weakening }

$[CALL \wedge BEEP(i)]\#f' \wedge fromto(f,\delta,f')$

unless $[CALL \wedge BEEP(i+1) \wedge \neg ON]\#f' \vee$

$[CALL \wedge BEEP(i+1) \wedge \neg ON]\#f' \vee$

$[CALL \wedge BEEP(i+1) \wedge \neg ON]\#f' \vee$

$[\neg CALL \wedge BEEP(i)]\#f$

$\equiv$ { math }

$[CALL \wedge BEEP(i)]\#f' \wedge fromto(f,\delta,f')$

unless $[CALL \wedge BEEP(i+1) \wedge \neg ON]\#f' \vee [\neg CALL \wedge BEEP(i)]\#f'$

$\Rightarrow$ { f and δ are not used }

$[CALL \wedge BEEP(i)]\#f'$

unless $[CALL \wedge BEEP(i+1) \wedge \neg ON]\#f' \vee [\neg CALL \wedge BEEP(i)]\#f'$ QED F28

Refinement 9

F28.1

$[\text{ARRIVE} \wedge \neg\text{LCALL}(\delta)]\#f \wedge [\text{CALL} \wedge \text{BEEP}(i)]\#f' \wedge \text{fromto}(f,\delta,f')$
unless $[\text{ARRIVE} \wedge \text{LCALL}(\delta)]\#f \wedge \text{BEEP}(i+1)\#f'$

Our appoach to this proof is to show that, whenever the system is in a state satisfying the left-hand side of F28.1, then the only transitions allowed by the remainder of the specification will satisfy the right-hand side of F28.1. To begin the proof, we include in the formulation the current time, with the assumption that less than *b* seconds have elapsed since the cabin entered the *arrive* state:

$$\langle \exists\, t : \&t :: [\text{ARRIVE} \wedge \neg\text{LCALL}(\delta)]\#f \wedge \text{CALL}\#f' \wedge \text{fromto}(f,\delta,f') \wedge t \leq \mathbf{ARRIVE}\$+b \,\rangle$$

Th negation of this property is then:

$$\langle \forall\, t :: \neg\&t \vee \neg([\text{ARRIVE} \wedge \neg\text{LCALL}(\delta)]\#f \wedge \text{CALL}\#f' \wedge \text{fromto}(f,\delta,f') \wedge t > \mathbf{ARRIVE}\$+b) \,\rangle$$

By T2, there is always a time *t*, so the first disjunct is always false, leaving us with:

$$\langle \exists\, t : \&t :: (\neg\text{ARRIVE}\#f \vee \text{LCALL}(\delta)\#f \vee \neg\text{CALL}\#f' \vee \neg\text{fromto}(f,\delta,f') \vee t > \mathbf{ARRIVE}\$+b) \,\rangle$$

Because *f*, *δ*, and *f'* are all constants, the fourth disjunct is always false, so we the get:

$$\langle \exists\, t : \&t :: (\neg\text{ARRIVE}\#f \vee \text{LCALL}(\delta)\#f \vee \neg\text{CALL}\#f' \vee t > \mathbf{ARRIVE}\$+b) \wedge \text{fromto}(f,\delta,f') \,\rangle$$

Finally, because the cabin will not arrive at the floor with the call (since *fromto(f,δ,f')* remains true), then the call must remain by F13, which means the 3rd disjunct is always false:

$$\langle \exists\, t : \&t :: (\neg\text{ARRIVE}\#f \vee \text{LCALL}(\delta)\#f \vee t > \mathbf{ARRIVE}\$+b) \wedge \text{fromto}(f,\delta,f') \,\rangle$$

This gives us the following property, which is true because *P* **unless** $\neg P$ is always true:

$[\text{ARRIVE} \wedge \neg\text{LCALL}(\delta)]\#f \wedge [\text{CALL} \wedge \text{BEEP}(i)]\#f' \wedge \text{fromto}(f,\delta,f')$
unless $\langle \exists\, t : \&t ::$
$(\neg\text{ARRIVE}\#f \vee \text{LCALL}(\delta)\#f \vee t > \mathbf{ARRIVE}\$+b) \wedge \text{fromto}(f,\delta,f') \,\rangle$

To finish the proof, we must show that of the 8 possible combinations for the right-hand side, only those which satisfy F28.1 are possible.

A. $\neg\text{ARRIVE}\#f \wedge \text{LCALL}(\delta)\#f \wedge t > \mathbf{ARRIVE}\$+b \wedge \text{fromto}(f,\delta,f')$
B. $\neg\text{ARRIVE}\#f \wedge \text{LCALL}(\delta)\#f \wedge t \leq \mathbf{ARRIVE}\$+b \wedge \text{fromto}(f,\delta,f')$
C. $\neg\text{ARRIVE}\#f \wedge \neg\text{LCALL}(\delta)\#f \wedge t > \mathbf{ARRIVE}\$+b \wedge \text{fromto}(f,\delta,f')$
D. $\neg\text{ARRIVE}\#f \wedge \neg\text{LCALL}(\delta)\#f \wedge t \leq \mathbf{ARRIVE}\$+b \wedge \text{fromto}(f,\delta,f')$
E. $\text{ARRIVE}\#f \wedge \text{LCALL}(\delta)\#f \wedge t > \mathbf{ARRIVE}\$+b \wedge \text{fromto}(f,\delta,f')$
F. $\text{ARRIVE}\#f \wedge \text{LCALL}(\delta)\#f \wedge t \leq \mathbf{ARRIVE}\$+b \wedge \text{fromto}(f,\delta,f')$
G. $\text{ARRIVE}\#f \wedge \neg\text{LCALL}(\delta)\#f \wedge t > \mathbf{ARRIVE}\$+b \wedge \text{fromto}(f,\delta,f')$
H. $\text{ARRIVE}\#f \wedge \neg\text{LCALL}(\delta)\#f \wedge t \leq \mathbf{ARRIVE}\$+b \wedge \text{fromto}(f,\delta,f')$

Properties A, E, F, and H satisfy F28.1. By F31, it is not possible for ARRIVE to be false if *b* seconds have not elapsed, so 2 of the options are eliminated (B, D), leaving only C and G, which are eliminated by F30 and F32, since after *b* seconds have elapsed, a beep will have occurred, and so *LCALL* must have been established.

Appendix B. Intermediate programs.

```
program elevator1(N)

definitions
    ⟨ f,f',δ : 1 ≤ f ≤ N, 1 ≤ f' ≤ N, δ ∈ { -1, 1 } ::
        needed(f,δ) ≡ rq(f) ∨ (δ=1 ∧ up(f)) ∨ (δ=-1 ∧ dn(f));
        fromto(f,δ,f') ≡ sign(f'-f)=δ;
        must_go(f,δ) ≡ ⟨ ∃ f',δ' : fromto(f,δ,f') :: needed(f',δ') ⟩
        work_in(f,δ) ≡ needed(f,δ) ∨ must_go(f,δ)
    ⟩;

tuple types
    ⟨ f : 1 ≤ f ≤ N :: rq(f) ⟩;
    ⟨ f : 1 < f ≤ N :: dn(f) ⟩;
    ⟨ f : 1 ≤ f < N :: up(f) ⟩;

transaction types
    ⟨ f,δ : 1 ≤ f ≤ N, δ ∈ { -1, 1 } ::
        Arrive(f,δ) ≡
            ||  ¬needed(f,δ), must_go(f,δ) → Depart(f,δ)                          (F14.1.1a)
            ||  NOR → StoppedClosed(f,δ);                                          (F14.1.1b)

        StoppedOpen(f,δ) ≡
                must_go(f,δ) → Depart(f,δ), open†                                  (F14.1.2a)
            ||  ¬must_go(f,δ), work_in(f,-δ) → StoppedClosed(f,-δ)                 (F14.1.2b)
            ||  ¬must_go(f,δ), ¬work_in(f,-δ) → StoppedClosed(f,δ);

        StoppedClosed(f,δ) ≡
            ||  needed(f,δ), δ=1 → StoppedOpen(f,δ), rq(f)†, up(f)†               (F14.1.3a, F9a)
            ||  needed(f,δ), δ=-1 → StoppedOpen(f,δ), rq(f)†, dn(f)†              (F14.1.3a, F9b)
            ||  ¬needed(f,δ), must_go(f,δ) → Depart(f,δ)                           (F14.1.3b)
            ||  ¬needed(f,δ), ¬must_go(f,δ), work_in(f,-δ) → StoppedClosed(f,-δ);  (F14.1.3c)

        Depart(f,δ) ≡
                true → Arrive(f+δ,δ);                                              (F14.3)
    ⟩;

    ⟨ f : 1 ≤ f ≤ N ::
        Up(f) ≡
            f ≠ N, ¬on(f) → up(f);

        Dn(f) ≡
            f ≠ 1, ¬on(f) → dn(f);

        Rq(f) ≡
            ¬on(f) → rq(f);
    ⟩;

initialization
    StoppedOpen(1,1);

end
```