Design and Analysis of Embedded Real-Time Systems: An Elevator Case Study

Ying Zhang and Alan K. Mackworth*

Department of Computer Science
University of British Columbia
Vancouver, B.C., Canada, V6T 1Z2
E-mail: zhang@cs.ubc.ca, mack@cs.ubc.ca

Abstract

Constraint Nets have been developed as an algebraic on-line computational model of robotic systems. Timed ∀-automata have been studied as a logical specification language of real-time behaviors. A general verification method has been proposed for checking whether a constraint net model satisfies a timed ∀-automaton specification. In this paper, we illustrate constraint net modeling and timed ∀-automata analysis using an elevator example. We start with a description of the functions and user interfaces of a simple elevator system, and then model the complete system in Constraint Nets. The analysis of a well-designed elevator system should guarantee that any request will be served within some bounded time. We specify such requirements in timed ∀-automata, and show that the constraint net model of the elevator system satisfies the timed ∀-automaton specification.

1 Introduction

We present here a methodology for specifying, designing, modeling, simulating, controlling, analyzing and verifying complex artifacts that interact with a changing environment. Since elevator systems have been used in various communities as a benchmark example of such methodologies [6, 9, 5, 1, 2, 3], we use them here to demonstrate the use of our framework.

*Shell Canada Fellow, Canadian Institute for Advanced Research

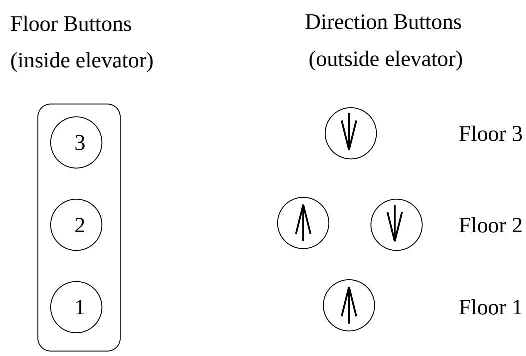

Figure 1: The user interface of a simple 3-floor elevator

2 A Simple Elevator System

A simple elevator system for an n-floor building consists of one elevator. Inside the elevator there is a board of n *floor buttons* each of which indicates a destination floor. Outside the elevator there are two *direction buttons* on each floor which call for service, one for up and the other for down (except the first floor and the top floor on which only one button is needed, see Fig. 1). Any button can be pushed at any time. Any floor button will be on from the time it is pushed until the elevator stops at the floor. Any direction button will be on from the time it is pushed until the elevator stops at the floor with the same direction. A more complex elevator might have open and close door buttons, and alarm or emergency buttons which, for simplicity, we will not model here. The atomic actions of an elevator at our first level of analysis consist of move-up or move-down one floor, serve-a-floor (stop at the floor, open and close the door) and stay-idle. The reason we initially consider serve-a-floor an atomic action is that the actions of stop, open and close do not interleave with any other atomic actions. For example, it is not possible to have the action sequence "open, move-up, close, ...". The complete elevator system consists of ELEVATOR BODY, ELEVATOR CONTROL and USER INTERFACE as shown in Fig. 2.

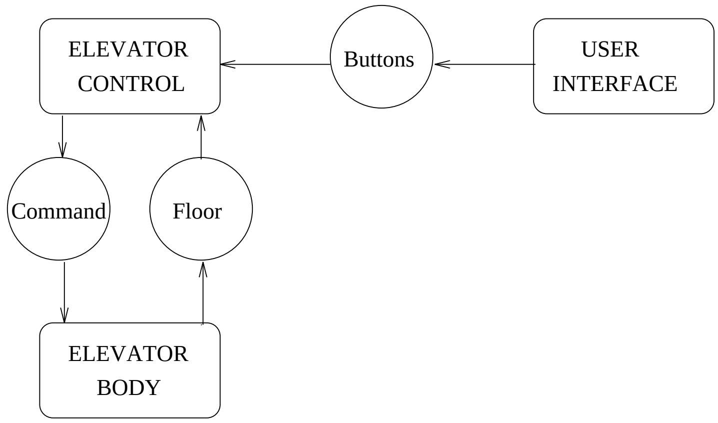

Figure 2: The complete elevator system

The rest of this paper is organized as follows. Section 2 briefly introduces Constraint Nets, and demonstrates constraint net modeling using the elevator example. Section 3 presents timed $\forall$-automata and gives the logical specification of the real-time behavior of the elevator system. Section 4 develops an algorithm which determines whether the constraint net model of the elevator system satisfies the $\forall$-automaton specification of the desired behavior. Section 5 discusses some general issues of this modeling and analysis approach.

3 The Constraint Net Model

3.1 Time, events and transductions

To start this section, we introduce some general concepts of dynamic systems which will be used later.

- *Variable trace*: A variable trace is a function from time to a domain of values. For simplicity, we shall consider time as the set of natural numbers or the set of non-negative real numbers with arithmetic ordering. The former is a *discrete time structure* and the later is a *dense time structure*.

- *Event trace*: An event trace is a variable trace whose domain is boolean. An event in any event trace is a transition from 0 to 1 or 1 to 0.

- *Transduction*: A transduction is a mapping from a tuple of input traces to an output trace which is causal, viz. the output value at any time is determined by the input values prior to or at that time. Transductions can be considered as transformational processes. For example, a temporal integration is a transduction. We will see that any deterministic automaton defines a transduction. Clearly, transductions are closed under composition. Transductions are the basic concept in on-line models, analogous to functions in off-line models. *Transliterations* and *delays* are two elementary types of transductions for dynamic systems.

- *Transliteration*: A transliteration $f_{\mathcal{T}}$ is a pointwise extension of function f, that is, the output value at any time is the function applied to the input value at that time. Let v be the input variable trace then we have $f_{\mathcal{T}}(v) = \lambda t.f(v(t))$.

- *Unit delay*: A unit delay $\delta(init)$ is a transduction defined on discrete time structures. Let v be the input variable trace then we have

$$\delta(init)(v) = \lambda k. \begin{cases} init & \text{if } k = 0 \\ v(k-1) & \text{otherwise.} \end{cases}$$

- *Transport delay*: A transport delay $\Delta(\delta)(init)$, where $\delta \geq 0$ is a real number, is a transduction defined on dense time structures. Let v be the input variable trace then we have

$$\Delta(\delta)(init)(v) = \lambda t. \begin{cases} init & \text{if } t < \delta \\ v(t-\delta) & \text{otherwise.} \end{cases}$$

- *Event-driven transductions*: any transduction defined on a discrete time structure can be extended to an event-driven transduction with an additional input of an event trace. The set of events in the event trace becomes the time structure of the transduction. The output value of the transduction holds between events.

- *Dynamics of robotic system*: A robotic system consists of a robot coupled to its environment. A robot is composed of a plant coupled to a controller. In Fig. 2, ELEVATOR BODY is the plant, ELEVATOR CONTROL is the controller and USER INTERFACE is the environment. A robotic system can be modeled as a set of transductions. For the elevator example we have

$$\begin{aligned} FloorTrace &= Elevator\,(CommandTrace) \\ CommandTrace &= Control\,(ButtonsTrace, FloorTrace) \\ ButtonsTrace &= Interface\,() \end{aligned}$$

  The overall behavior of the system is not determined by any one of the transduction alone, but emerges from the coupling of the interactions among all the transductions. Formally, the trajectory of the system is the solution of the set of equations.

3.2 Constraint nets

A *constraint net* is a triple $CN \equiv \langle Lc, Td, Cn \rangle$, where Lc is a set of *locations*, Td is a set of *transductions*, each of which is associated with a tuple of *input ports* and an *output port*; Cn is a set of directed *connections* between locations and ports of transductions. Topologically, a constraint net is a bipartite directed graph where locations are represented by circles, transductions are represented by boxes and connections are represented by arcs, each from a port of a transduction to a location or vice versa. A location is an *input* iff there is no connection pointing to it otherwise it is an *output*. For a constraint net CN, the set of input locations is denoted by $I(CN)$, the set of output locations is denoted by $O(CN)$. A constraint net is *closed* iff there are no input locations otherwise it is *open*.

Semantically, a constraint net is a set of equations on a set of variables corresponding to Lc, where each left-hand side is an individual output location and each right-hand side is an expression composed of transductions and locations. The semantics of a constraint net is defined to be the least fixpoint of the set of equations [12].

A *module* is a triple $CN(I, O)$ where CN is a constraint net, $I \subseteq I(CN)$ (resp. $O \subseteq O(CN)$) is a subset of the input (resp. output) locations of CN; $I \cup O$ defines the *interface* of the module. Complex modules can be hierarchically constructed from simple ones using three operations: *composition*, *coalescence* and *hiding* [12].

4 A Discrete Model of the Elevator System

We can model dynamic systems at any level of abstraction in constraint nets. For the elevator example, we shall first design a control strategy where discrete modeling is appropriate. In this example, each atomic action of the elevator takes one time unit.

Transducers are the basic building blocks for discrete modeling. A *state transducer* is a quadruple $\langle I, Q, q_0, f \rangle$ where I is the set of inputs, Q is the set of states, $q_0 \in Q$ is the initial state and $f : I \times Q \to Q$ is the *state transition function*. A state transducer can be represented by an open constraint net with a transliteration $f_{\mathcal{T}}$ and a unit delay $\delta(q_0)$. A state transducer defines a transduction from an input trace to a state trace. For the elevator example, the elevator body is modeled as a state transducer (Fig. 3), $Elevator = \langle Command, Floor, 1, NextFloor \rangle$, where $Command$ is $\{up, down, serve, stay\}$, $Floor$ is $\{1, 2, \ldots, n\}$, $NextFloor$ is the state transition function defined as

$$NextFloor(c, f) = \begin{cases} f+1 & \text{if } c = up \\ f-1 & \text{if } c = down \\ f & \text{otherwise.} \end{cases}$$

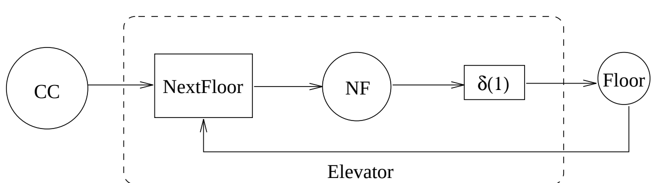

Figure 3: The *Elevator* module

A *transducer* is a tuple $\langle I, Q, q_0, f, O, f^o \rangle$ where $\langle I, Q, q_0, f \rangle$ is a state transducer, O is the set of outputs and $f^o : I \times Q \to O$ is the *output function*. A transducer can be represented by an open constraint net composed of a state transducer and a transliteration. A transducer defines transductions from an input trace to a state trace and an output trace. For the elevator example, we model the kernel of the control system as a transducer (Fig. 4),

$$Control1 = \langle Inputs, State, idle, CurrentState, Command, CurrentCommand \rangle$$

where $Inputs = Floor \times UButtons \times DButtons \times FButtons$, and $UButtons = DButtons = FButtons = \{0,1\}^n$, n is the number of floors, $State = \{up, down, idle\}$ is the set of control states, $CurrentState$ is a state transition function, $CurrentCommand$ is an output function. This transducer defines transductions from floor and button traces to state and command traces. For a well-designed elevator controller, the state is necessary for remembering the current direction of motion, so that the elevator will move persistently in one direction until there is no request in that direction. We will see that this strategy will bound the waiting time for any request. In order to define $CurrentState$ and $CurrentCommand$ we define some

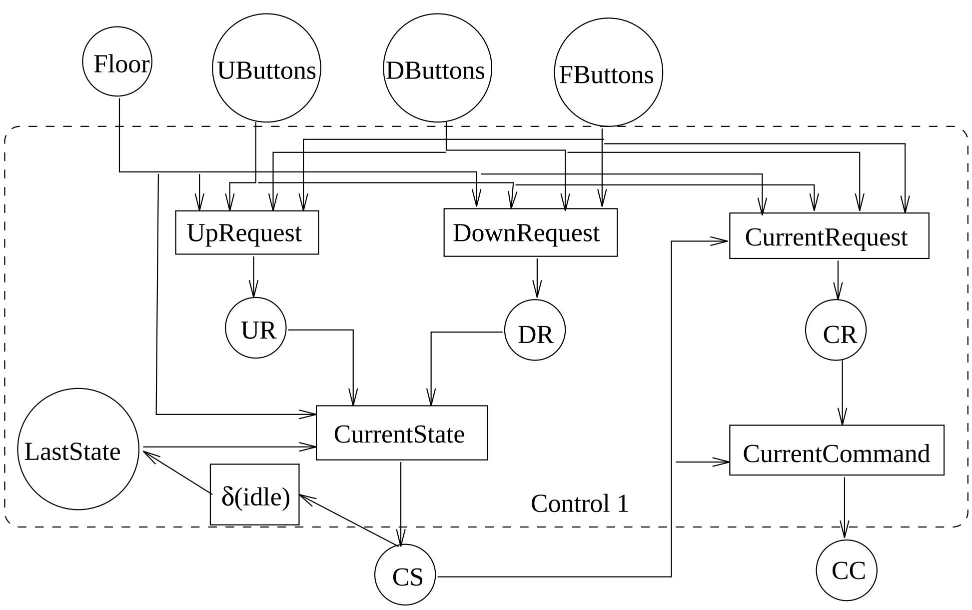

Figure 4: The $Control1$ module

predicates on the current situation.

- UR indicates whether or not there is any request for the elevator to go up:

$$\begin{aligned} UR &= UpRequest(f, ub, db, fb) \\ &= ub(f) \bigvee_{i>f} (ub(i) \vee db(i) \vee fb(i)). \end{aligned}$$

- DR indicates whether or not there is any request for the elevator to go down:

$$\begin{aligned} DR &= DownRequest(f, ub, db, fb) \\ &= db(f) \bigvee_{i<f} (ub(i) \vee db(i) \vee fb(i)). \end{aligned}$$

The state transition function can be defined as follows:

$$\begin{aligned} CS &= CurrentState(f, ub, db, fb, ls) \\ &= \begin{cases} up & \text{if } UR \wedge (ls \neq down \vee \neg DR) \\ down & \text{if } (\neg UR \wedge f > 1) \vee (DR \wedge ls = down) \\ idle & \text{otherwise.} \end{cases} \end{aligned}$$

In English, if there is a request for the elevator to go up and either the last state is up or there is no request to go down, the elevator will be in the up state; if there is no request to go up and the elevator is not at the first floor, or the last state is down and there is a request to go down, then the elevator will be in the down state; otherwise the elevator will be idle, that is, the elevator will be parked at the first floor if there are no more requests.

Let CR indicates whether or not there is any request for the elevator to stop and serve the current floor:

$$\begin{aligned} CR &= CurrentRequest(f, ub, db, fb, ls) \\ &= \begin{cases} db(f) \vee fb(f) & \text{if } CS = down \\ ub(f) \vee fb(f) & \text{otherwise.} \end{cases} \end{aligned}$$

In English, if there is an internal request to arrive at this floor or there is an external request to go in the same direction as the elevator, there is a request at this floor.

The output function can be defined as follows:

$$\begin{aligned} CC &= CurrentCommand(f, ub, db, fb, ls) \\ &= \begin{cases} serve & \text{if } CR \\ CS & \text{otherwise.} \end{cases} \end{aligned}$$

In English, if there is a request at this floor, the elevator will stop to serve the floor (open the door, let passagers to go in and out, then close the door), otherwise the elevator will pass this floor without stopping.

Generalizing, we can model a discrete dynamic system in a constraint net composed only of transliterations and unit delays. Any output location of a unit delay is called a *state* location. To model the complete control of the elevator system, we need to define another two modules which turn off the corresponding button when the request is served. We use a tuple of flip-flops for this purpose. A flip-flop is a transducer with the first input as set and the second input as reset. Formally,

$$FlipFlop(set, reset, state) = \begin{cases} 0 & \text{if } reset \\ 1 & \text{otherwise if } set \\ state & \text{otherwise.} \end{cases}$$

A unit delay is added to form the transducer (Fig. 5(a)). This flip_flop is designed so that *reset* has higher priority than *set*. For the elevator example, this means that the elevator can not stop for more than one unit time.

The reset signals are created by the *ClearBoard* module (Fig. 5(b)) which sets the corresponding bit when the request is served.

Summarizing, the elevator system is a constraint net composed of four modules: *Elevator*, *Control*1, *FlipFlop* and *ClearBoard* (Fig. 6). We have implemented the discrete model of the elevator system in Strand88 [4], a concurrent logic programming language (Appendix A.1). It is easy to simulate discrete constraint nets in Strand88 where a variable trace is an infinite list, and both transliterations and unit delays can be represented.

```
%f(+in, -out) is a function. fT(+in_trace, -out_trace) is a transliteration.
fT([I|Is], OS) :- f(I, O), OS := [O|Os], fT(Is, Os).
%delay(+init, +in_trace, -out_trace)
delay(Init, In, Out) :- Out := [Init|In].
```

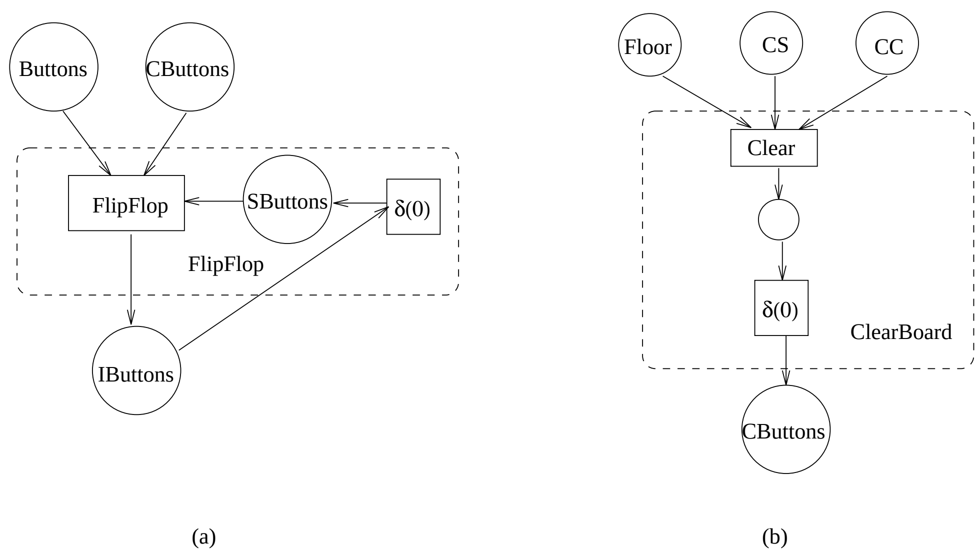

Figure 5: (a) The $FlipFlop$ module (b) The $ClearBoard$ module

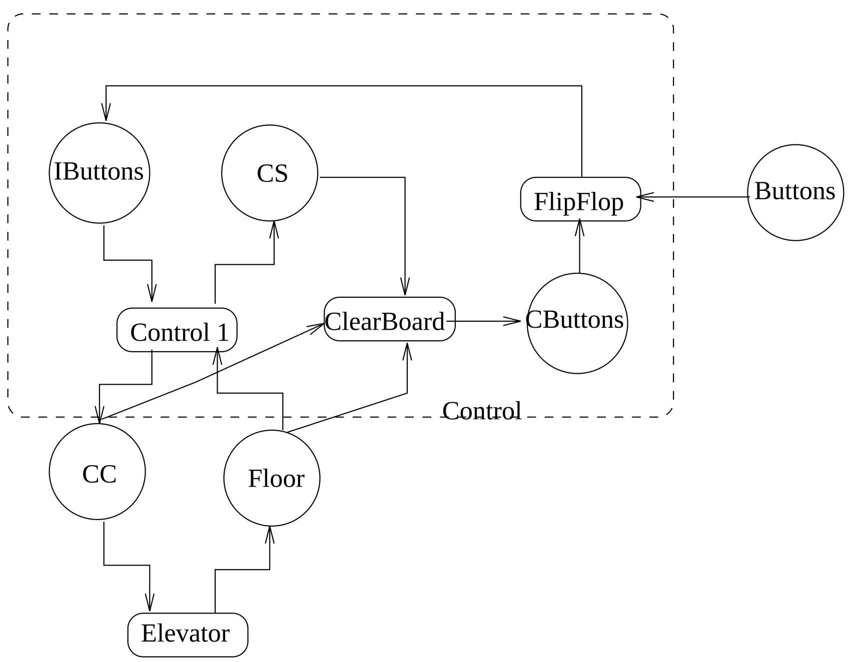

Figure 6: The constraint net of the elevator system

5 A More Realistic Model in Constraint Nets

More details should be considered for designing a real elevator system. In the previous modeling of the elevator system, atomic actions are primitives. Now we shall model how these actions are carried out by the lower level control system, which is modeled as an analog controller. Many physical systems and analog control systems are better modeled as differential equations in dense time structures. Now we shall model how the discrete event-driven controller interacts with the continuous dynamic system.

A dynamic system in a dense time structure can be modeled by constraint nets composed of transliterations, transport delays and event-driven transductions. In continuous modeling, we always consider integration as a basic transduction, even though it can be defined in terms of a limiting behavior generated by an infinitesimal transport delay.

We have implemented this more realistic model of the elevator system in ALERT, A Laboratory of Embedded Real-Time systems, developed in Simulink. ALERT is a visual programming and simulation environment based on the semantics of constraint nets. ALERT consists of various building blocks, from simple modules of event logics, linear and nonlinear transductions to complex constraint solvers. ALERT provides hierarchical structures for constructing complex modules from simple ones.

The overall structure of the elevator system modeled in ALERT is shown in Fig. 7.

For this more realistic modeling, the *Elevator* module is further decomposed into two modules. One is the body of the elevator described by Newton's Law $F - K_V\dot{h} = \ddot{h}$ where F is the motor force, K_V is the friction coefficient and h is the current height of the elevator (Fig. 8). (We assume the mass is 1 since it can be scaled by F and K_V. We ignore the constant force of gravity since it can be added to F to compensate.) This module defines a transduction from the force trace to the height trace.

The other module is the lower level controller of the elevator. $Control0$ transforms commands from the higher level controller $Control1$ into forces to drive the elevator and sends back the state of the elevator, which are the current floor number and whether or not the elevator

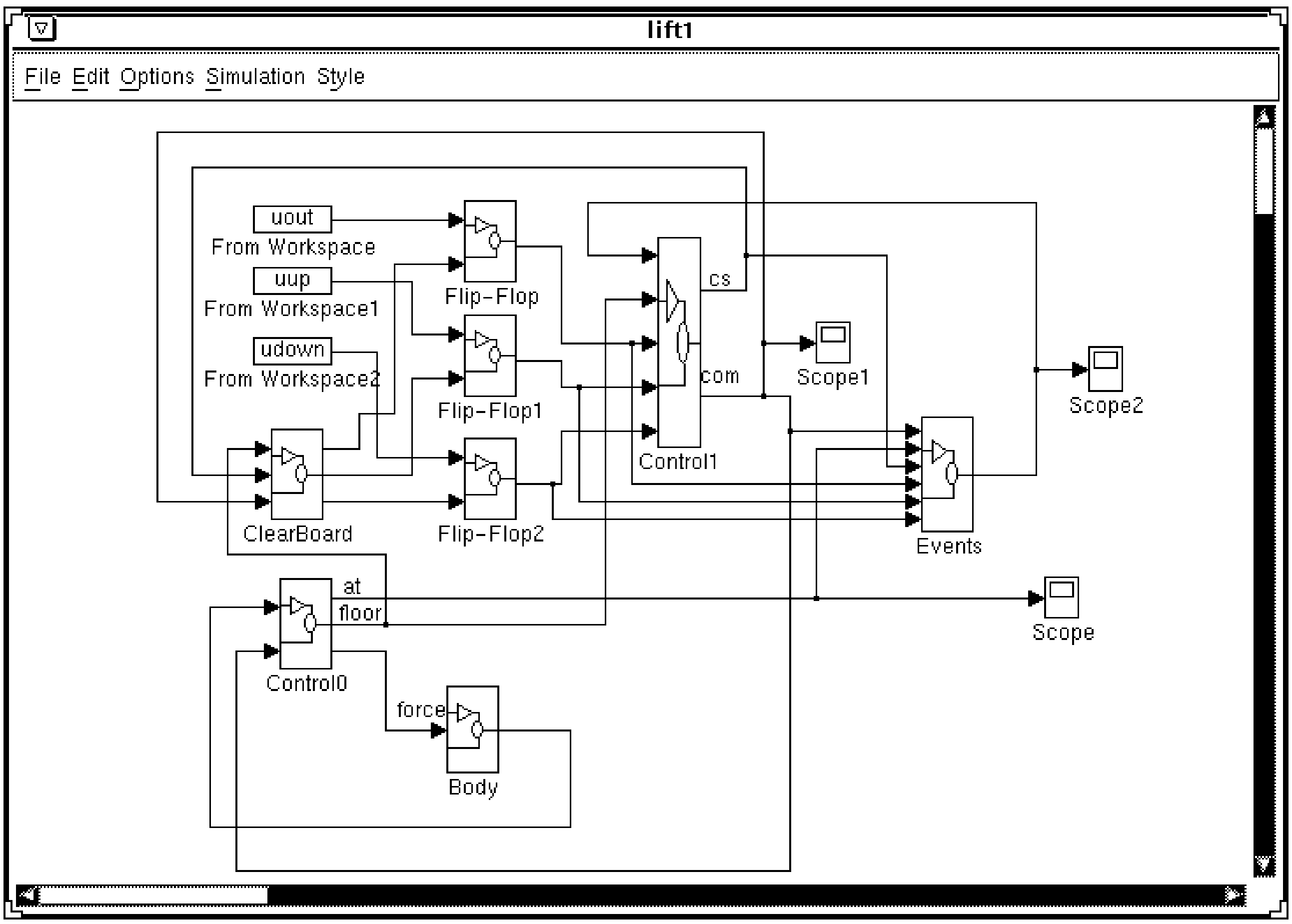

Figure 7: The elevator system in ALERT (Locations are not explicitly represented here; however, any wire is interpreted as a location. Scopes are display devices.)

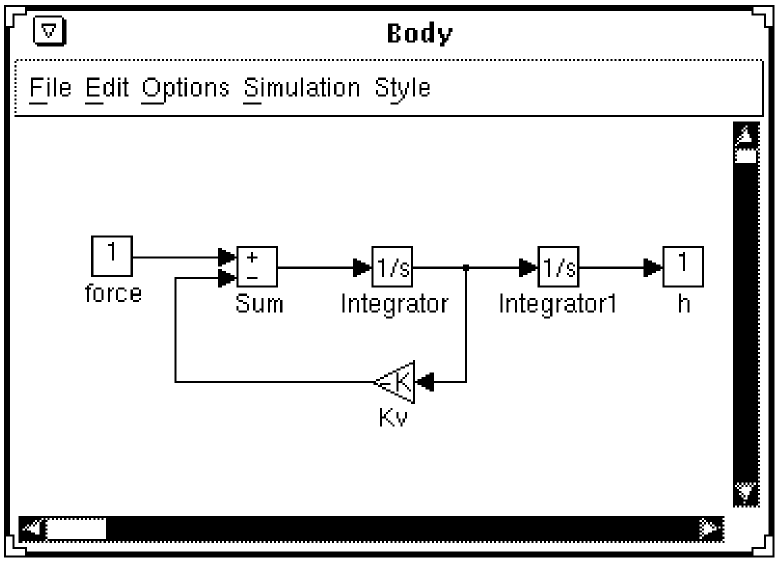

Figure 8: The dynamics of the elevator body

is close to the floor (within 15 cm), to the higher level controller and the event generator. Fig. 9 shows the lower level controller where commands *up*, *down* and *stop* are 1, −1 and 0 respectively. For up and down, a constant force is applied; for stop, proportional control is applied where K_P is the gain. (K_P should be chosen so that the elevator can stop at the floor within a specified error.) An arbitration is used to combine the two according to the current command. This controller can be implemented in analog circuits.

$Control1$, $ClearBoard$ and $FlipFlops$ function in the same way as in the discrete model. However, $ClearBoard$ and $FlipFlops$ are digital circuits implemented at a fast sampling rate (0.1 sec.) so that any input signal can be responded to quickly. $Control1$ is now an event-driven transduction where events are generated by the event module in Fig. 10.

There are three basic types of events: (1) someone pushes a button in the elevator's idle state, (2) the elevator becomes close to a floor (within 15 cm) and (3) the elevator has served the floor for a certain time; the time of the transport delay (say 4 sec.) is the time for serving the floor. The "event or" of these three events excites the $Control1$ module to produce a new output.

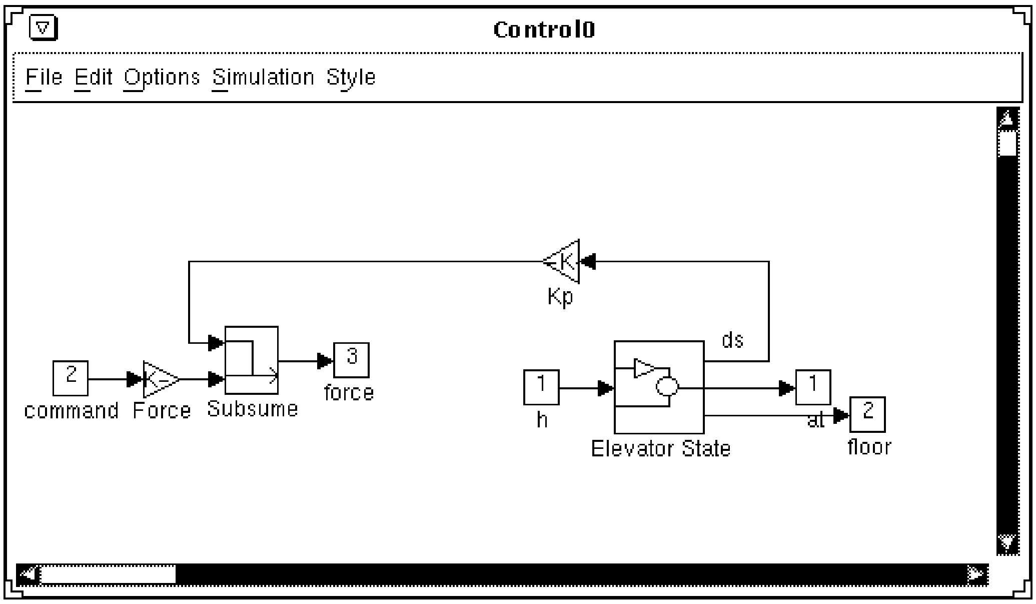

Figure 9: The lower level analog control

6 The Timed ∀-Automaton Specification

While modeling focuses on the underlying structure of a system, the organization and coordination of components or subsystems, the overall behavior of the modeled system is not explicitly expressed. However, for many situations, it is important to specify some global properties and guarantee that these properties hold in this design. For example, a well-designed elevator system should service any request within a bounded waiting time.

A logical specification can serve this purpose. In this section, we shall present timed ∀-automata for specifying behaviors in discrete time structures, and demonstrate how to use this logical specification to state desired properties of an elevator system.

6.1 State formulas

A *state formula* Fs is a first-order formula:

$$Fs \equiv false \mid T_1 = T_2 \mid p(T_1, \ldots, T_n) \mid Fs_1 \rightarrow Fs_2 \mid \exists x Fs$$

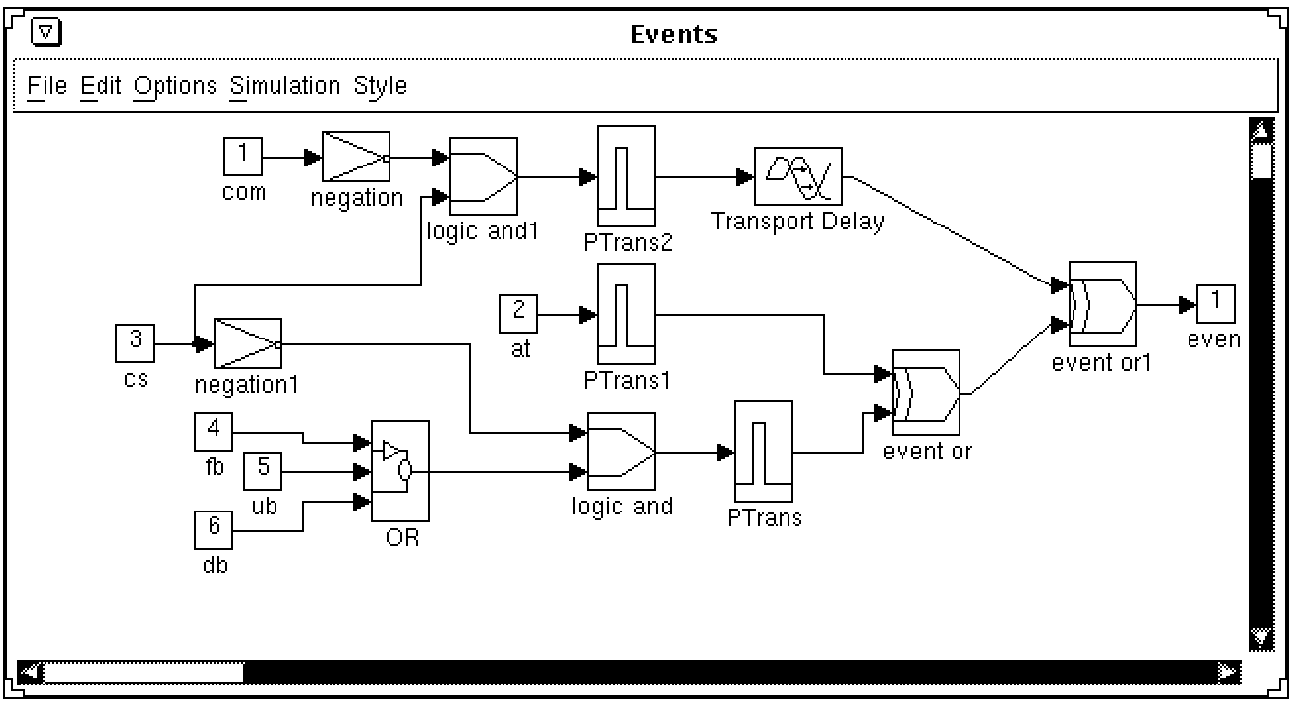

Figure 10: The event generator (PTrans generates an event if its input changes from 0 to 1)

where T_i is a term defined on locations and parameters, p is a predicate and x is a parameter.

A frame $\mathcal{F}_r$ is a pair $\langle A, I\rangle$ where A is a multi-sorted algebra and I is an interpretation which maps each predicate p to a set of tuples in A. A model $\mathcal{M}$ is a pair $\langle \mathcal{F}_r, \sigma\rangle$ where σ is a mapping from locations to variable traces. Let v_i be a valuation of the locations at a given time i, i.e. $v_i(x) = \sigma(x)(i)$ and v_i^* be the valuation extended to terms. A model $\mathcal{M}$ satisfies a state formula F at time i, written $\mathcal{M} \models_i F$, as follows:

- $\mathcal{M} \not\models_i false$;
- $\mathcal{M} \models_i T_1 = T_2$ iff $v_i^*(T_1) = v_i^*(T_2)$;
- $\mathcal{M} \models_i p(T_1, \ldots, T_n)$ iff $(v_i^*(T_1), \ldots, v_i^*(T_n)) \in I(p)$;
- $\mathcal{M} \models_i F_1 \rightarrow F_2$ iff $\mathcal{M} \models_i F_1$ implies $\mathcal{M} \models_i F_2$;
- $\mathcal{M} \models_i \exists x F$, iff $\exists v \in A$, $\mathcal{M} \models_i F[v/x]$, where $F[v/x]$ stands for substitution.

A model of a constraint net CN is $\langle \mathcal{F}_r, \sigma\rangle$ such that σ is consistent with the semantics of CN, i.e. $\forall o \in O(CN)$, $\sigma(o) = F_o(\sigma(i))$ where F_o is the transduction corresponding to the output location o.

For the elevator example, there are three kinds of requests: to go to a particular floor after entering the elevator, or to go up or down when waiting for the elevator. The following are some examples of state formulas:

- $R2 \equiv (FButtons(2) = 1)$ denotes that "there is a request to go to the second floor".
- $R2S \equiv (CC = stop \wedge Floor = 2)$ denotes that "the request to go to the second floor is served".
- $RU2 \equiv (UButtons(2) = 1)$ denotes that "there is a request to go up at the second floor".
- $RU2S \equiv (CS = up \wedge CC = stop \wedge Floor = 2)$ denotes that "the request to go up at the second floor is served".

6.2 Timed $\forall$-automaton

A $\forall$-automaton [8] $\mathcal{A}$ is a quintuple $\langle Q, R, S, e, c\rangle$ where Q is a finite set of *automaton states*, $R \subseteq Q$ is a set of *recurrent states* and $S \subseteq Q$ is a set of *stable states*. Let $\mathcal{F}s$ be the set of state formulas, $e : Q \rightarrow \mathcal{F}s$ is an *entry condition* function such that $\bigvee_{q\in Q} e(q) = true$, and $c : Q \times Q \rightarrow \mathcal{F}s$ is a *transition condition* function such that $\forall q \in Q, \bigvee_{q'\in Q} c(q, q') = true$. A *run* of $\mathcal{A}$ over a model $\mathcal{M}$ on a discrete time structure is a sequence of states $q_0, q_1, q_2, \ldots$ such that (1) $\mathcal{M} \models_0 e(q_0)$; and (2) for all time points $i > 0$, $\mathcal{M} \models_i c(q_{i-1}, q_i)$. Let $Inf(r) \subseteq Q$ denote the set of automaton states that appear infinitely many times in r. A run r is defined to be *accepting* iff: (a) $Inf(r) \cap R \neq \emptyset$; or (b) $Inf(r) \subseteq S$. A $\forall$-automaton $\mathcal{A}$ *accepts* a model $\mathcal{M}$, written $\mathcal{M} \models \mathcal{A}$, iff *all* possible runs of $\mathcal{A}$ over $\mathcal{M}$ are accepting. $\mathcal{A}$ accepts a constraint net CN, written $CN \models \mathcal{A}$, iff $\mathcal{A}$ accepts all models of CN. It has been shown [8] that the specification power of $\forall$-automata is identical to that of ETL, an extended linear discrete temporal logic [11].

A graphical representation of a $\forall$-automaton is a directed graph with nodes as automaton states and arcs as transitions. Each state in R is marked with $\diamond$ and each state in S is marked with $\Box$. If $e(q) \neq false$, there is at least one arrow to node q. If $c(q, q') \neq false$, there is at least one arc from q to q'. Each node, arrow and arc are labeled by a state formula F, such that $e(q) = (\bigvee_{a\in Arrows} Fa) \wedge Fq$ and $c(q, q') = (\bigvee_{a\in Arcs} Fa) \wedge Fq'$. By default, nodes, arrows or arcs without labels are labeled with the formula $true$.

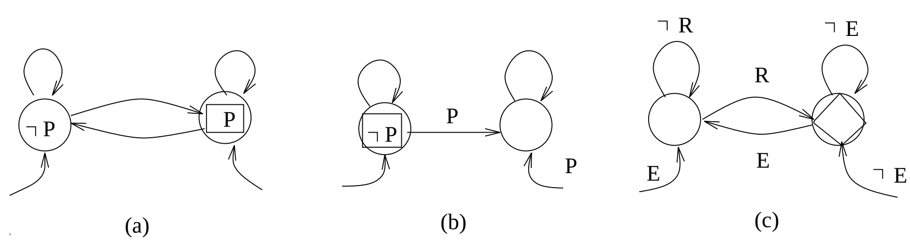

Figure 11: $\forall$-automata

Some examples of $\forall$-automata are shown in Fig. 11. Fig. 11(a) accepts a computation which satisfies $\neg P$ only finitely many times, Fig. 11(b) accepts a computation which never satisfies P and Fig. 11(c) accepts a computation which will satisfy R in the finite future whenever it satisfies E.

For the elevator example, bounded time responses "the request to go to the second floor will be served in finite time" and "the request to go up at the second floor will be served in finite time" are represented in Fig. 12 (a) and (b) respectively.

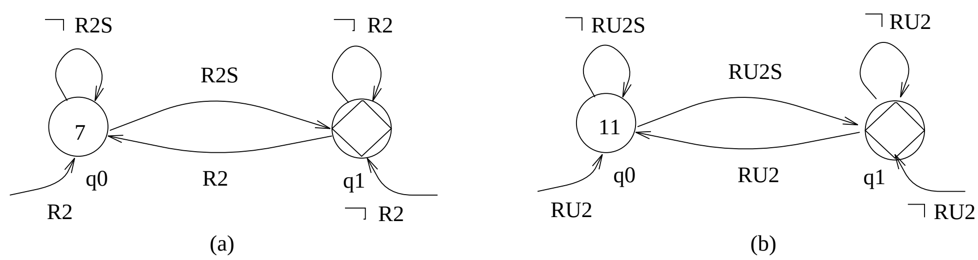

Figure 12: Specifications of bounded time responses for the elevator

To guarantee real-time response, we extend $\forall$-automata to timed $\forall$-automata by adding a new class of automaton state, *timed states*: $T \subseteq Q$. Associated with any $q \in T$, there is a natural number n_q. A run is accepting if, in addition, for any $q \in T$ and any run segment s_q of consecutive q's, the length of the segment $|s_q|$ is bounded above by n_q. Graphically, a T-state q is denoted by n_q.

For the elevator example, let 7 be associated with q_0 in Fig. 12 (a) and 11 be associated with q_0 in Fig. 12 (b). The new automata specify the properties: "the request to go to the second floor will be served within 7 time units" and "the request to go up at the second floor will be served within 11 time units".

7 The Verification Method

There is a general method for checking that a timed $\forall$-automaton specification is satisfied by a constraint net on a discrete time structure [13]. In this section, we first introduce this verification method and then develop algorithms for finite systems.

7.1 The general method

Let S be the set of state locations, $O = O(CN)\backslash S$ be the set of the other output locations, $I = I(CN)$ be the set of input locations and A be the domain of values. Generally, a constraint net on a discrete time structure can be written as two sets of equations: $s' = f_s(i, s)$; $o = f_o(i, s)$ where s,$s' : S \to A$ is a state tuple, $i : I \to A$ is an input tuple, f_s is a tuple of state transition functions, $o : O \to A$ is an output tuple, and f_o is a tuple of output functions. Therefore, such a constraint net is globally a transducer $\langle \mathcal{I}, \mathcal{S}, s_0, f_s, \mathcal{O}, f_o \rangle$ where $\mathcal{I} = \{i | i : I \to A\}$ is the set of inputs, $\mathcal{S} = \{s | s : S \to A\}$ is the set of states, s_0 is the initial state, f_s is the state transition function, $\mathcal{O} = \{o | o : O \to A\}$ is the set of outputs and f_o is the output function.

We write $\{\varphi\}CN\{\psi\}$ to denote the verification condition:

$$\varphi[f_o(i, s)/o] \bigwedge s' = f_s(i, s) \to \psi[i'/i, s'/s, f_o(i', s')/o]$$

where φ and ψ are state formulas; φ characterizes pre-conditions and ψ indicates post-conditions. For example, if ES is the elevator system in Fig. 6, we have

$$\{FButton(2) = 1 \wedge CC = down \wedge Floor = 3\}ES\{FButton(2) = 0\}$$

Model checking of a constraint net CN against its specification $\mathcal{A}$ involves three phases: *Phase 1*: Associate with each automaton state $q \in Q$ an assertion $\alpha_q \in L_s$, called the *invariant* at q, such that the following requirements are satisfied:

- *Initiality*: $[s_0 \wedge e(q)] \to \alpha_q, \forall q \in Q$.
- *Consecution*: $\{\alpha_q\}CN\{c(q, q') \to \alpha_{q'}\}, \forall q, q' \in Q$.

Phase 2: Associate with each automaton state $q \in Q$ a ranking function $\rho_q : \mathcal{I} \times \mathcal{S} \to \mathcal{W}$, where $\mathcal{W}$ is a *well-founded* set, i.e., $\forall w_0 \in \mathcal{W}$, any decreasing sequence $w_0 > w_1 > \ldots$ is finite, such that the following requirements are satisfied:

- *Definedness*: $\alpha_q \to \exists w.\rho_q = w, \forall q \in Q$.

- *Non-increase*: $\{\alpha_q \wedge \rho_q = w\}CN\{c(q,q') \rightarrow \rho_{q'} \leq w\}, \forall q' \in S$.

- *Decrease*: $\{\alpha_q \wedge \rho_q = w\}CN\{c(q,q') \rightarrow \rho_{q'} < w\}, \forall q' \in Q \backslash (R \cup S)$.

Phase 3: Associate with each $q \in T$ a timing function, $\gamma_q : \mathcal{I} \times \mathcal{S} \rightarrow \mathcal{N}$ where $\mathcal{N}$ is the set of natural numbers, such that the following requirements are satisfied:

- *Boundedness*: $\alpha_q \rightarrow 1 \leq \gamma_q \leq n_q, \forall q \in T$;

- *Decrease*: $\{\alpha_q \wedge \gamma_q = w\}CN\{c(q,q) \rightarrow \gamma_q < w\}, \forall q \in T$.

The verification rule is sound and complete, that is, if we succeed in finding invariants α_q, a well-founded set $\mathcal{W}$, ranking functions ρ_q and timing functions γ_q, such that all the requirements are satisfied, then this establishes the validity of $\mathcal{A}$ over CN; on the other hand, if $\mathcal{A}$ is valid over CN, then there are invariants α_q, a well-founded set $\mathcal{W}$, ranking functions ρ_q and timing functions γ_q, such that all the requirements are satisfied [8].

7.2 The algorithm

A constraint net is *finite* iff the domain of values is finite. For example, finite state transducers are represented by finite constraint nets. Digital sequential circuits can be represented by finite constraint nets since the domain of values, boolean, is finite. We can automate the model checking process for finite constraint nets as long as $\{\alpha_q\}$ is known as a *priori*. A finite constraint net is a transducer of finite input-state pairs. For each $q \in Q$, $e(q)$ and α_q corresponds to a set of input-state pairs of the constraint net. For each $q, q' \in Q$, $c(q,q')$ corresponds to a binary relation of the input-state pairs. The algorithm consists of three phases: (1) `Initiality and Consecution`, (2) `Ranking` and (3) `Timing`.

The algorithm for the first phase is a direct translation of the verification rules (Fig. 13).

The algorithm of the second phase consists of two steps. The first step of this algorithm is to generate a state transition graph. An input-state pair (i,s) is in the graph iff $\alpha_q(i,s)$ for some stable or bad automaton state q. There is a transition between (i,s) and (i',s') iff

```
Algorithm: Initiality and Consecution
for all q in Q do
    if not itest(q) return false
for all q, q' in Q do
    if not ctest(q, q') return false
itest(q):
    for all input i and initial state s0 do
      if e(q)(i,s0) and not a(q)(i,s0)
      return false
    return true
ctest(q, q'):
    for all (i,s) and (i',s') do
      if a(q)(i,s) and s'=fs(i,s) and c(q,q')(i,s,i',s') and
         not a(q')(i',s') return false
    return true
```

Figure 13: The algorithm for initiality and consecution

$\alpha_q(i, s) \wedge \alpha_{q'}(i', s') \wedge c(q, q')(i, s, i', s')$. An input-state pair (i, s) is called bad iff $\alpha_q(i, s)$ for some bad automaton-state q. The property of the graph we want to check is: there is no loop consisting of any bad input-state pairs (Fig. 14).

```
Algorithm: Ranking
1. /* Generate a state transition graph <V,E> */
   for all q in B = Q \ R do
    for all (i,s) do
     if a(q)(i,s) put (i,s) in V
   for all q, q' in Q \ R do
    for all (i,s) and (i',s') do
     if a(q)(i,s) and s' = fs(i,s) and c(q,q')(i,s,i',s')
     put (i,s,i',s') in E
2. /* Check the acyclicity of bad states in <V,E> */
   for all (i,s) in V with a(q)(i,s) for some bad state q do
    for all path p from (i,s) do
     if p ends at (i,s) return false
   return true
```

Figure 14: The algorithm for ranking

The algorithm of the third phase is similar to the second phase except that we need to check the maximum length of each timed state in all the transition paths in the graph (Fig. 15).

We have implemented this model checking algorithm in Quintus Prolog (Appendix A.2). The specifications in the previous section are checked out using this algorithm by letting $\alpha_{q0} = R2$ (resp. $RU2$) and $\alpha_{q1} = R2S$ (resp. $RU2S$). The discrete constraint net model of the elevator system is given in "liftimp.pl" (Appendix A.3); the timed $\forall$-automata are specified in "liftspe1.pl" and "liftspe2.pl" respectively (Appendix A.4). If $n = 4$, the state transition graphs for the elevator controller are shown in Fig. 16 (a) and (b) respectively, where dashed transitions are disabled in our control strategy. For simplicity, a state in the graph is a cluster of input-state pairs.

```
Algorithm: Timing
for all q in T do
    if not ttest(q) return false
    return true
ttest(q):
    1. /* Generate a state transition graph <V,E> */
       for all (i, s) do
        if a(q)(i,s) put (i,s) in V
       for all (i,s) and (i',s') do
        if a(q)(i,s) and s' = fs(i,s) and c(q,q)(i,s,i,s')
           put (i,s,i',s') in E
    2. /* Check the longest path */
       for all (i,s) in V with no input edges do
        for all path p from (i,s) do
         if |p| > time(q) or p has loop return false
       return true
```

Figure 15: The algorithm for timing

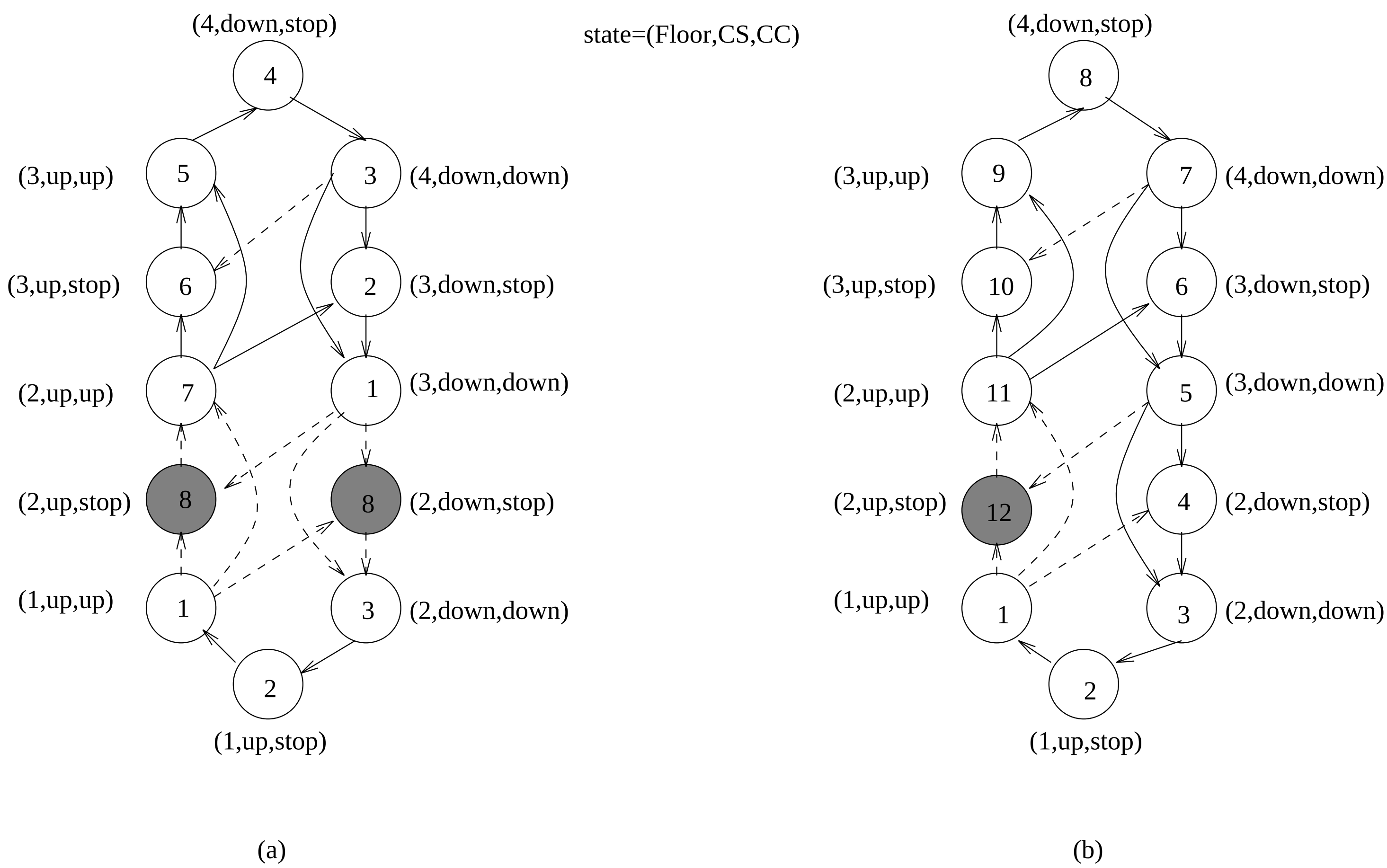

Figure 16: State transition graphs for the elevator system where unshaded nodes are bad nodes, and dashed lines are disabled transitions (a)[resp. (b)] corresponds to the specification in Fig. 12 (a)[resp. (b)].

8 Conclusions

We have discussed modeling and analysis in the context of an elevator system. Now we shall make some general comments on this approach and draw some conclusions.

8.1 Modeling

An appropriate model of behavioral dynamics in robotic systems should satisfy the following properties (modified from [7, 10]):

- it is a real-time model where time can be represented explicitly;

  A robot coupled to its environment is a complex dynamic system. The interaction between the robot and the environment is often constrained by time.

- it can model the dynamics of the environment as well as the the dynamics of the plant and the dynamics of the control;

  A robot is an open system. The behavior of the system is meaningless without considering the dynamics of its environment. It is important to represent the *structural congruence* between the dynamics of the system and the dynamics of its external environment, i.e. to model the environment as a machine.

- it can represent dense, as well as discrete, dynamic systems, asynchronous processes, and the coordination of processes with different dynamics;

  The control system of a robot may consist of multiple components with both analog and digital circuits. The communication between different counterparts may incorporate both time-driven and event-driven conventions.

- it can provide multiple levels of abstraction.

  Like most complex systems, the organization of control systems should be modular and hierarchical. With this organization, a system can be incrementally designed, debugged and verified.

Finally, one of the most important characteristics for a model is:

- it is formal and rigorous.

  The model should have formal syntax and semantics, so that the system being designed or modeled can be specified or analyzed without ambiguity. It is important to understand the overall behavior of a system during the design period and guarantee that some properties of the system (e.g. safety and stability) will hold when it is embedded in an environment. Both simulation and computer-aided theorem-proving can assist this process of analysis.

Constraint Net is a model developed to satisfy these requirements.

8.2 Specification

While modeling focuses on the underlying structure of a system, and the organization and coordination of components and subsystems, the overall behavior of the modeled system is not explicitly expressed. A logical specification imposes constraints on the system's global behaviors. The need to have a formal specification is two-fold:

1. *for formal verification*: a logical specification defines a desired global behavior of the system under design, which can be verified against the model of the system formally; and more importantly,

2. *for systematic design*: properties like goal achievement with a priority hierarchy can guide the design of hierarchical control systems.

8.3 Verification

There are two ways of verifying a system:

1. *via simulation*: we can always attempt to verify a system via exhaustive testing or simulation. For many systems, it is the only option.

2. *via formal verification*: for discrete time systems, computer-aided model checking or theorem proving is possible. Furthermore, for finite systems automatic verification can be achieved.

However, neither simulation nor formal verification is sufficient to guarantee well-behavedness. First, the model is only an abstraction of the real system, the real system probably has unmodeled aspects which may cause problems. Second, the specification maybe incomplete: it is hard to enumerate all the desired properties, and there maybe properties which are not expressible in the given specification language.

Even though verification is not sufficient, it will decrease run-time problems for the designed system. Proper system design is based on an appropriate level of abstraction and a sufficient set of desired properties.

Acknowledgements: Thanks to Peter Caines, Hector Levesque, Ray Reiter and Maarten van Emden for stimulating us to work on this problem. This work was supported in part by the Natural Sciences and Engineering Research Council, the Institute for Robotics and Intelligent Systems and the Canadian Institute for Advanced Research.

References

[1] H. Barringer. Up and down the temporal way. Technical report, University of Manchester, England, September 1985.

[2] P.E. Caines, T. Mackling, and Y.J. Wei. A logical controller for CIAR1 elevator system by cocolog, 1992. Unpublished.

[3] D.N. Dyck and P.E. Caines. The logical control of an elevator, 1992. Unpublished.

[4] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming*. Prentice Hall, 1989.

[5] R. Hale. Using temporal logic for prototyping: The design of a lift controller. In H.S.M. Zedan, editor, *Real-Time Systems, Theory and Applications*. Elsvier Science Publishers B.V. (North-Holland), 1990.

[6] M.A. Jackson. *System Development*. Prentice-Hall, Englewood Cliffs, 1983.

[7] J. Lavignon and Y. Shoham. Temporal automata. Technical Report STAN-CS-90-1325, Robotics Laboratory, Computer Science Department, Stanford University, Stanford, CA 94305, 1990.

[8] Z. Manna and A. Pnueli. Specification and verification of concurrent programs by ∀-automata. In *Proc. 14th Ann. ACM Symp. on Principles of Programming Languages*, pages 1–12, 1987.

[9] B. Sanden. An entity-life modeling approach to the design of concurrent software. *Communication of the ACM*, 32(3):230 – 243, March 1989.

[10] Y. Shoham. Agent-oriented programming. Technical Report STAN-CS-1335-90, Computer Science Department, Stanford University, 1990.

[11] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72 – 99, 1983.

[12] Y. Zhang and A. K. Mackworth. Constraint Nets: A semantic model of real-time embedded systems. Technical Report 92-10, Department of Computer Science, University of British Columbia, 1992.

[13] Y. Zhang and A. K. Mackworth. Will the robot do the right thing? Technical Report 92-31, Department of Computer Science, University of British Columbia, 1992.

A Appendix

A.1 Strand88 Implementation

```
-compile(free).
-exports([commTd/6, clearT/6, floorTd/2, board/4, boardTd/3, system/6]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                   Functions                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
comm(F, LCS, UP, DOWN, OUT, COM, CS) :-
    urequest(F, UP, DOWN, OUT, UR),
    drequest(F, UP, DOWN, OUT, DR),
    currents(F, LCS, UR, DR, CS),
    crequest(F, CS, UP, DOWN, OUT, CR),
    command(CS, CR, COM).

currents(_, idle, 1, _, CS) :- CS := up.
currents(_, up, 1, _, CS) :- CS := up.
currents(_, down, 1, 0, CS) :- CS := up.
currents(F, _, 0, _, CS) :- F > 1|CS := down.
currents(_, down, _, 1, CS) :- CS := down.
currents(1, _, 0, 0, CS) :- CS := idle.

command(_CS, 1, COM) :- COM := serve.
command(CS, 0, COM) :- COM := CS.

floor(F, down, NF) :- NF is F - 1.
floor(F, up, NF) :- NF is F + 1.
floor(F, serve, NF) :- NF := F.
floor(F, idle, NF) :- NF := F.

clear(F, up, serve, CUP, CDOWN, COUT) :-
    reset(4, CDOWN),
    clear1(F, COUT),
    clear1(F, CUP).

clear(F, down, serve, CUP, CDOWN, COUT) :-
    reset(4, CUP),
    clear1(F, COUT),
    clear1(F, CDOWN).
```

```
clear(_, _, _, CUP, CDOWN, COUT) :- otherwise|
    reset(4, CUP),
    reset(4, CDOWN),
    reset(4, COUT).

reset(0, X) :- X := [].
reset(I, X) :- I > 0| I1 is I - 1, X := [0|X1], reset(I1, X1).

clear1(F, X) :-
      reset(4, X0),
      set(F, X0, X).

set(1, [_|X0], X) :- X := [1|X0].
set(N, [0|X0], X) :- N > 1| N1 is N - 1, X := [0|X1], set(N1, X0, X1).

crequest(F, down, _, DOWN, OUT, CR) :-
       element(F, DOWN, E1),
       element(F, OUT, E2),
       or(E1, E2, CR).

crequest(F, _, UP, _, OUT, CR) :- otherwise |
       element(F, UP, E1),
       element(F, OUT, E2),
       or(E1, E2, CR).

urequest(4, _, _, _, UR) :- UR := 0.
urequest(F, UP, DOWN, OUT, UR) :- otherwise |
       F1 is F + 1,
       element(F, UP, E0),
       elements(F1, UP, Es1),
       elements(F1, DOWN, Es2),
       elements(F1, OUT, Es3),
       or(Es1, E1),
       or(Es2, E2),
       or(Es3, E3),
       or([E0, E1, E2, E3], UR).

drequest(1, _, _, _, DR) :- DR := 0.
drequest(F, UP, DOWN, OUT, DR) :- otherwise |
       F1 is F - 1,
```

```
       element(F, DOWN, E0),
       elemente(F1, UP, Es1),
       elemente(F1, DOWN, Es2),
       elemente(F1, OUT, Es3),
       or(Es1, E1),
       or(Es2, E2),
       or(Es3, E3),
       or([E0, E1, E2, E3], DR).

element(1, [X|_], E) :- E := X.
element(N, [_|Xs], E) :- N > 1| N1 is N - 1, element(N1, Xs, E).

elements(1, X, Es) :- Es := X.
elements(N, [_|Xs], Es) :- N > 1| N1 is N - 1, elements(N1, Xs, Es).

elemente(1, [X|_], Es) :- Es := [X].
elemente(N, [X|Xs], Es) :- N > 1| Es := [X|Es1], N1 is N - 1,
  elemente(N1, Xs, Es1).

or(1, _, X) :- X := 1.
or(0, 1, X) :- X := 1.
or(_, _, X) :- otherwise | X := 0.

or([], X) :- X := 0.
or([E|Es], X) :- or(Es, X1), or(E, X1, X).

board([], [], [], NB) :- NB := [].
board([C|Cs], [U|Us], [B|Bs], NBS) :-
      ff(C, U, B, NB),
      NBS := [NB|NBs],
      board(Cs, Us, Bs, NBs).

ff(1, 0, _, NB) :- NB := 0.
ff(_, 1, _, NB) :- NB := 1.
ff(0, 0, B, NB) :- NB := B.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%           Transliterations                                  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
commT([], [], [], [], [], COMS, NCSS) :- COMS := [], NCSS := [].
commT([F|Fs], [CS|CSs], [UP|UPs], [DOWN|DOWNs], [OUT|OUTs], COMS, NCSS) :-
```

```
     comm(F, CS, UP, DOWN, OUT, COM, NCS),
     COMS := [COM|COMs], NCSS := [NCS|NCSs],
     commT(Fs, CSs, UPs, DOWNs, OUTs, COMs, NCSs).

floorT([], [], NFS) :- NFS := [].
floorT([F|Fs], [Com|Coms], NFS) :-
     floor(F, Com, NF),
     NFS := [NF|NFs],
     floorT(Fs, Coms, NFs).

clearT([], [], [], CUPS, CDOWNS, COUTS) :-
     CUPS := [], CDOWNS := [], COUTS := [].
clearT([F|Fs], [Cs|Css], [Com|Coms], CUPS, CDOWNS, COUTS) :-
     clear(F, Cs, Com, CUP, CDOWN, COUT),
     CUPS := [CUP|CUPs],
     CDOWNS := [CDOWN|CDOWNs],
     COUTS := [COUT|COUTs],
     clearT(Fs, Css, Coms, CUPs, CDOWNs, COUTs).

boardT([], [], [], NBS) :- NBS := [].
boardT([C|Cs], [U|Us], [B|Bs], NBS) :-
      board(C, U, B, NB),
      NBS := [NB|NBs],
      boardT(Cs, Us, Bs, NBs).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%              Unit Delay                                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

delay(I, NS, S) :- S := [I|NS].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%              Transducers                                         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
floorTd(Coms, Fs) :-
     delay(1, NFs, Fs),
     floorT(Fs, Coms, NFs).

commTd(Fs, UPs, DOWNs, OUTs, COMs, CSs) :-
     delay(idle, CSs, PCSs),
     commT(Fs, PCSs, UPs, DOWNs, OUTs, COMs, CSs).
```

```
boardTd(Cs, Us, Bs) :-
     reset(4, I),
     delay(I, Bs, PBs),
     boardT(Cs, Us, PBs, Bs).

clearTd(FS, CS, COMS, CUPS, CDOWNS, COUTS) :-
     reset(4, IUPS),
     reset(4, IDOWNS),
     reset(4, IOUTS),
     delay(IUPS, NUPS, CUPS),
     delay(IDOWNS, NDOWNS, CDOWNS),
     delay(IOUTS, NOUTS, COUTS),
     clearT(FS, CS, COMS, NUPS, NDOWNS, NOUTS).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%             Elevator     System                                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
system(UUPS, UDOWNS, UOUTS, FS, COMS, CS) :-
     boardTd(CUPS, UUPS, UPS),
     boardTd(CDOWNS, UDOWNS, DOWNS),
     boardTd(COUTS, UOUTS, OUTS),
     clearTd(FS, CS, COMS, CUPS, CDOWNS, COUTS),
     floorTd(COMS, FS),
     commTd(FS, UPS, DOWNS, OUTS, COMS, CS).
```

A.2 Model Checker in Quintus Prolog (rtmc.pl)

```
:- dynamic bstate/2, sstate/2, bstrans/4, tstate/2, ttrans/4.

:- ensure_loaded(library(basics)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                       Model Checker                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

model_checker :-
     initiality,
     consecution,
     ranking, !,
```

```
     timing.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                      Initiality                                  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

initiality :-
     setof(Q, isa(Q), Qs),
     initiality(Qs).

isa(Q) :-
     a(Q, _).

initiality([Q|Qs]) :-
     itest(Q),
     initiality(Qs).

initiality([]).

itest(Q) :-
     e(Q, Pe), !,
     a(Q, Pa),
     itest(Pe, Pa).

itest(_).

itest(Pe, Pa) :-
     ifail(Pe, Pa), !, fail.

itest(_, _).

ifail(P, P) :- !, fail.

ifail(Pe, Pa) :-
     start(S),
     callP(Pe, [I, S]),
     \+ callP(Pa, [I, S]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                       Consecution                                  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
consecution :-
     setof((Q, Q1), isc(Q, Q1), Ts),
     consecution(Ts).

isc(Q, Q1) :-
     c(Q, Q1, _).

consecution([(Q, Q1)|Ts]) :-
     ctest(Q, Q1),
     consecution(Ts).

consecution([]).

ctest(Q, Q1) :-
     c(Q, Q1, Pc),
     a(Q, Pa),
     a(Q1, Pa1),
     ctest(Pa, Pc, Pa1).

ctest(Pa, Pc, Pa1) :-
     cfail(Pa, Pc, Pa1), !, fail.

ctest(_, _, _).

cfail(_, P, P) :- !, fail.

cfail(Pa, Pc, Pa1) :-
     callP(Pa, [I, S]),
     trans(I, S, S1),
     callP(Pc, [I, S, I1, S1]),
     \+ callP(Pa1, [I1, S1]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                           Ranking                                  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ranking :-
     rinit,
     bad(Bq, T),
     states(bstate, Bq),
```

```
     stable(Sq),
     states(sstate, Sq),
     append(Bq, Sq, Qq),
     ptrans(bstrans, Qq, Qq),
     rtest(T).

rinit :-
     retractall(bstate(_,_)),
     retractall(sstate(_,_)),
     retractall(bstrans(_,_,_,_)).

states(_, []).
states(P, [Q|Qs]) :-
     a(Q, Pa),
     setof1([I,S], callP(Pa, [I, S]), ISs),
     asserts(P, ISs),
     states(P, Qs).

asserts(_, []).
asserts(P, [S|Ss]) :-
     P1 =.. [P|S],
     assert1(P1),
     asserts(P, Ss).

assert1(P) :- P, !.

assert1(P) :- assert(P).


ptrans(_, [], _).
ptrans(P, [Q|Qs], Q1s) :-
     ptrans1(P, Q, Q1s),
     ptrans(P, Qs, Q1s).

ptrans1(_, _, []).
ptrans1(P, Q, [Q1|Q1s]) :-
     ptrans2(P, Q, Q1),
     ptrans1(P, Q, Q1s).

ptrans2(P, Q, Q1) :-
     setof1([I,S,I1,S1],
```

```
     consistent(Q,Q1,I,S,I1,S1), Ts),
     asserts(P, Ts).

consistent(Q, Q1, I, S, I1, S1) :-
     a(Q, Pa),
     c(Q, Q1, Pc),
     callP(Pa, [I,S]),
     trans(I, S, S1),
     callP(Pc, [I,S,I1,S1]).

rtest(T) :-
     setof1([I,S], bstate(I,S), ISs),
     rtest(ISs, T).

rtest(ISs, -1) :- noloops(ISs).
rtest(ISs, T) :- bdtimes(ISs, T).

noloops([]).
noloops([S|Ss]) :-
     noloop(S),
     noloops(Ss).

noloop(IS) :- loop_path(IS), !, fail.
noloop(_) :- !.

loop_path(IS) :- loop_path(IS, [IS]).

loop_path([I,S], [[I1,S1]|Path]) :-
     bstrans(I1,S1,I,S),
     write([[I,S],[I1,S1]|Path]), nl.

loop_path(IS, [[I1,S1]|Path]) :-
     bstrans(I1,S1,I2,S2),
     \+ member([I2,S2], [[I1,S1]|Path]),
     loop_path(IS, [[I2,S2],[I1,S1]|Path]).

bdtimes([], _).
bdtimes([S|Ss], T) :-
     bdtime(S, T),
     bdtimes(Ss, T).
```

```
bdtime(S, T) :- btime(S, T), !, fail.
bdtime(_, _) :- !.

btime(IS, T) :- btime(IS, ([IS], 0), T).

btime(_, ([[I1,S1]|Path], T1), T2) :-
     T1 >= T2, !,
     write([[I1,S1]|Path]), nl.

btime([I,S], ([[I1,S1]|Path], _), _) :-
     bstrans(I1,S1,I,S), !,
     write([[I,S],[I1,S1]|Path]), nl.

btime(IS, ([[I1,S1]|Path], T1), T) :-
%     write([[I1,S1]|Path]), nl,
%     write(T1),nl,
      bstrans(I1,S1,I2,S2),
      \+ member([I2,S2], [[I1,S1]|Path]),
     (bstate(I2,S2), T2 is T1 + 1;
      sstate(I2,S2), T2 is T1),
      btime(IS, ([[I2,S2],[I1,S1]|Path],T2), T).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                      Timing                                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

timing :-
     time(TQs),
     timing(TQs).

timing([]).
timing([Tq|TQs]) :-
     ttest(Tq),
     timing(TQs).

ttest((Q, N)) :-
     tinit,
     states(tstate, [Q]),
     ptrans2(ttrans, Q, Q),
     maxt(N).
```

```
tinit :-
     retractall(tstate(_,_)),
     retractall(ttrans(_,_,_,_)).

maxt(N) :-
     setof1([I,S], beginstate(I,S), Ss),
     longtime(Ss, N).

beginstate(I,S) :- tstate(I,S), \+ ttrans(_,_,I,S).

longtime([], _).
longtime([S|Ss], N) :-
     lt(S, N),
     longtime(Ss, N).

lt(IS, N) :- bad_path(IS, N), !, fail.
lt(_, _) :- !.

bad_path(IS, N) :- bad_path(IS, N, ([IS], 0)).

bad_path([I,S], N, (Path, Tc)) :-
     ttrans(I,S,I1,S1),
     (member([I1,S1],Path); (T1 is Tc + 1, T1 >= N)),
     write([[I1,S1]|Path]), nl.

bad_path([I,S], N, (Path, Tc)) :-
     ttrans(I,S,I1,S1),
     T1 is Tc + 1,
     bad_path([I1,S1], N, ([[I1,S1]|Path], T1)).

callP(true, _) :- !.

callP(false, _) :- !, fail.

callP([], _) :- !.

callP([P|Ps], L) :- !,
     callP(P, L),
     callP(Ps, L).

callP(-P, L) :- !,
```

```
     Pd =.. [P|L],
     \+ call(Pd).

callP(P, L) :- !,
     Pd =.. [P|L],
     call(Pd).

setof1(X, Y, Z) :-
     setof(X, Y, Z), !.

setof1(_X, _Y, []).
```

A.3 Prolog Implementation (liftimp.pl)

```
:- dynamic floor/1, on/2.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                Initial Condition                     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

start([_,_,_,_]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                State Transitions                     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

trans(_, [F, LS, CS, CS1], [F1, LS1, CS1,CS2]) :-
    nextf(F, LS, F1),
    curcs(F, CS, CS1),
    curcs(F1, CS1, CS2),
    nextls(F, LS, CS1, LS1).

nextf(F, up, F1) :- F1 is F + 1.
nextf(F, down, F1) :- F1 is F - 1.
nextf(F, stop, F).

curcs(F, up, up) :- \+ height(F).
curcs(F, up, down) :- \+ up_request(F), F > 1.
curcs(F, down, down) :- F > 1.
```

```
curcs(F, down, up) :- \+ down_request(F), \+height(F).

nextls(_F, LS, _, stop) :- \+ LS == stop.
nextls(_F, stop, up, up).
nextls(_F, stop, down, down).
nextls(F, up, up, up) :- F1 is F + 1, \+ request(F1,up).
nextls(F, down, down, down) :- F1 is F - 1, \+ request(F1,down).

up_request(F) :-
      (on(F1, _), F1 > F); on(F, up).

down_request(F) :-
      (on(F1, _), F1 < F); on(F, down).

request(F,S) :-
      on(F,floor); on(F,S).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%             Set the Number of Floors                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

nfloors(N) :- assert(height(N)), nfloor(N).

nfloor(1) :-
    assert(floor(1)).

nfloor(N) :-
    assert(floor(N)),
    N1 is N - 1,
    nfloor(N1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                   Elevator States                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

lift_state(up).
lift_state(down).
lift_state(stop).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%                    Control States                            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

control_state(up).
control_state(down).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                    Predicates a(q)                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
ru2(_, [F, LS, CS, CS1]) :-
    floor(F),
    lift_state(LS),
    control_state(CS),
    control_state(CS1),
    constraint(F, LS, CS1),
    \+ [F, LS, CS1] == [2, stop, up],
    assert(on(2,up)).

r2(_, [F, LS, CS, CS1]) :-
    floor(F),
    lift_state(LS),
    control_state(CS),
    constraint(F, LS, CS1),
    \+ [F, LS] == [2, stop],
    assert(on(2,floor)).

constraint(F, stop, down) :- height(F).
constraint(1, stop, up).
constraint(F, stop, _) :- \+height(F), F > 1.
constraint(F, down, down) :- F > 1.
constraint(F, up, up) :- height(H), F < H.

nru2(_, [2, stop, CS, up]) :- control_state(CS).

nr2(_, [2, stop, CS, CS1]) :- control_state(CS), control_state(CS1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                   Predicates c(q,q')                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ru2(_,_, I, S) :- ru2(I, S).
```

```
r2(_,_,I,S) :- r2(I,S).

nru2(_,_,I,S) :- nru2(I, S).
nr2(_,_,I,S) :- nr2(I, S).
```

A.4 Behavioral Specifications

The following is the specification "liftspe1.pl" corresponding to Fig.12 (a).

```
(q0, r2).
a(q1, nr2).

e(q0, [r2]).
e(q1, [nr2]).

c(q0, q0, [r2]).
c(q0, q1, [nr2]).
c(q1, q1, [nr2]).
c(q1, q0, [r2]).

stable([]).
bad([q0], -1).
time([(q0, 7)]).
```

The following is the specification "liftspe2.pl" corresponding to Fig. 12 (b).

```
a(q0, ru2).
a(q1, nru2).

e(q0, [ru2]).
e(q1, [nru2]).

c(q0, q0, [ru2]).
c(q0, q1, [nru2]).
c(q1, q1, [nru2]).
c(q1, q0, [ru2]).

stable([]).
bad([q0], -1).
time([(q0,11)]).
```

The following is an example of model checking.

```
?- [rtmc].
?- [liftimp].
?- nfloors(4).
?- [liftspe1].
?- model_checker.
yes
?-
```