Computer Science Journal of Moldova, vol.4, no.1(10), 1996

Design of the real time systems using temporal logic specifications: a case study

A.Ursu V.Dubenetsky G.Gruita

Abstract

An implementation method for real time systems is proposed in this article. The implementation starts with the design of the functional specifications of the systems behaviour. The functional specifications are introduced as a set of rules describing the partial time ordering of the actions performed by the system. These rules are then written in terms of temporal logic formulae. The temporal logic formulae are checked using Z.Manna-P.Wolper satisfiability analysis procedure [1]. It is known that this procedure generates a state-graph which can be regarded as a state-based automaton of the system. The sate-based automaton is used then to generate the dual (inverted) automaton of the system. The dual automaton is called action-based automaton and can be created using the procedure proposed by authors in [4,5]. Using the action-based automaton of the system the design method introduced in [5,6] is applied to implement the system driver in a systematic manner which can be computerised.
The method proposed in this paper is an efficient complementation and generalisation of the results [4,5,6] mentioned above.
The method is used for a case study. An elevator control system is designed using the proposed method. The design is carried out in a systematic manner which includes:

a) design of functional specifications,

b) design of temporal logic specifications,

c) satisfiability analysis of temporal logic specifications,

d) design of the state-based automaton of the specifications,

e) design of the action-based automaton of the system,

f) design of the transition activation conditions,

g) design of the action activation conditions,

h) design of the functional model of the elevator control system,

i) implementation of the elevator's actions,

j) design of the elevator control system driver.

1 Introduction

Temporal logic formalism has been successfully used in concurrent and distributed systems specification and verification. The temporal logic operators [1,2] turned out to be very suitable in specifying different properties related adequately to such systems dealing with safety and liveness. The safety and liveness properties are easily specified in terms of temporal logic formulae. These properties are very important for real time systems too. The designers of a real time system have to implement the software which will work properly. Unfortunately the information about the system properties dealing with safety and liveness even if it is important for designer can not be used directly in the design of the real time control systems.

Some important results in this direction have been obtained by Z.Manna and P.Wolper [1], and E.M.Clarke and E.A.Emerson [3], which have proposed different approaches to satisfiability analysis of temporal logic specifications. Both approaches allow to generate the automaton of the specifications to be used in correctness analysis and in implementation of the real time systems. The automaton is an oriented graph, the nodes of which correspond to the states of the system and the edges of which correspond to the transitions between the states. Each state is specified by a set of temporal logic formulae and each edge of the automaton is specified by a propositional variable of the specifications.

This automaton is a state-based automaton which can be seen as a finite-state machine and used to synthesise the synchronisation part of the system. Nonetheless the finite state machines obtained in this manner are not very suitable for event- or action-based implementation of software systems. An action-based implementation requires a finite-state machine the nodes of which correspond to actions and the edges of which correspond to the control flow of the actions.

The paper shows that the initial (state-based) automaton of the temporal logic specifications can be used in the action-based approach of implementation of the real time software systems via an inversion procedure which translates the initial automaton into an inverse one. The inverted automaton consists of a set of nodes corresponding to the set of possible actions of the system and a set of edges corresponding to the control flow of the actions.

The rest of the paper is organised as follows. Section 2 introduces our method of real time systems design using logic specifications. Section 3 presents a case study which consists in designing an elevator control system using the proposed method. Section 4 presents conclusions.

2 The design method of real time systems using temporal logic specifications

In this paragraph the main steps of the proposed design method are introduced in a general framework.

The method consists of the following steps:

- design of the functional specifications of the system;
- explicit design of each rule of the system behaviour;
- translation of the rules into temporal logic formulae and design of the temporal logic specifications of the system;
- satisfiability analysis of temporal logic specifications;
- design of the initial (state-based) automata of the system using the satisfiability analysis result of temporal logic specifications;
- design of the dual (action-based) automata of the system using the initial automata via an inversion procedure;

- design of the automata transition activation conditions;
- design of action activation conditions of the action-based automata;
- design of the control structure of the system driver;
- design of the functional structure of the system;
- implementation of the system actions;
- implementation of the system control driver.

3 Design of an elevator control system: a case study

3.1 Short description of an elevator control system — the functional specifications

The functional specifications of an elevator control system are introduced in this section in a simplified manner.

The control system is dedicated to control an elevator in a building. The elevator is suspended from a motor-driven winch, the motor being capable of responding to the commands *Move_Up, Move_Down, Stop, Open_Door* and *Close_Door*. All the introduced commands have their obvious meaning. At each floor there are buttons to summon the elevator for upwards and downwards travel (only the former at the ground floor, and only the latter at the top floor); inside the elevator there is a button for each floor to direct the elevator to travel to that floor. At each floor there is a sensor switch that assumes the 'closed' position when the elevator is within 10 cm of its rest position at the floor, and is otherwise in its 'open' position.

3.2 The functioning rules of the elevator control system

According to the description of the elevator control system we introduce here the following functioning rules:

Rule 1: Initially the elevator is located at the first floor and the first action to be executed by the elevator control system must be the action *Move_Up()* or *Open_Door()*;

Rule 2: Always a *Move_Up()* action must be followed sooner or later by a new *Move_Up()* action or by a *Stop()* action;

Rule 3: Always a *Move_Down()* action must be followed sooner or later by a new *Move_Down()* action or by a *Stop()* action;

Rule 4: Always a *Stop()* action must be followed sooner or later by a new *Stop()* action or by one of the following actions: *Open_Door(), Move_Down(), Move_Up()*;

Rule 5: Always an *Open_Door()* action must be followed sooner or later by a *Close_Door()* action;

Rule 6: Always a *Close_Door()* action must be followed sooner or later by one of the following actions: *Open_Door(), Move_Up(), Move_Down()*;

Rule 7: This rule specifies the single event condition, that is only one action can be executed at any time. This means that the actions of the elevator control system are executed sequentially.

3.3 Temporal logic specifications of the elevator control system

According to the description of the elevator control system each operating rule of the elevator can be specified by the following temporal logic formulae:

$$\neg Pr\, U(Move\_Up \vee Open\_Door); \tag{1}$$

$$[]\,(Move\_Up \supset \bigcirc(\neg Pr\, U(Move\_Up \vee Stop))); \tag{2}$$

$$[]\,(Move\_Down \supset \bigcirc(\neg Pr\, U(Move\_Down \vee Stop)); \tag{3}$$

$$[]\,(Stop \supset \bigcirc(\neg Pr\, U Stop \vee Open\_Door \vee Move\_Up \vee Move\_Down)); \tag{4}$$

$$[]\,(Open\_Door \supset \bigcirc(\neg Pr\, U(Open\_Door \vee Close\_Door))); \tag{5}$$

$$[]\,(Close\_Door \supset \bigcirc(\neg Pr\, U(Open\_Door \vee Move\_Up \vee Move\_Down)); \tag{6}$$

$$Pr \equiv Move\_Up \vee Move\_Down \vee Open\_Door \vee Close\_Door \vee Stop \tag{7}$$

3.4 Satisfiability analysis of the specifications

For satisfiability analysis of the specifications we have used the Z.Manna and P.Wolper procedure [1]. Here we perform the satisfiability analysis of the formulae (1-7) in a similar manner as in [1]. The difference consists in the new notation for the formulae manipulation.

The procedure [1] postulates the single-event condition according to which only single events can occur at a time. The satisfiability analysis consists in giving true values to a proposition variable and checking the satisfiability of the remaining formulae of the specifications. This operation is carried out for all propositional variables one by one.

Let us start by the analysis of the formulae (1-7).

We consider initially the state when $Move\_Up = true$. We have:

Initial formula		Set of next-state formulae	
(1)	$\Rightarrow$	$true$;	
(2)	$\Rightarrow$	$\bigcirc(2)$,	
		$\bigcirc(\neg Pr\, U(Move\_Up \vee Stop)$	(8)
(3)	$\Rightarrow$	$\bigcirc(3)$;	
(4)	$\Rightarrow$	$\bigcirc(4)$;	
(5)	$\Rightarrow$	$\bigcirc(5)$;	
(6)	$\Rightarrow$	$\bigcirc(6)$;	

For this manipulation the following notation can be introduced:

$$(1-6) \overset{Move\_Up}{\longrightarrow} (2-6, \neg Pr\, U(Move\_Up \vee Stop))$$

or the notation:

$$(1-6) \overset{Move\_Up}{\longrightarrow} (2-6, 8).$$

Let us simplify the initial set of formulae (1-6) for $Open\_Door = true$. We have:

Initial formula		Set of next-state formulae	
(1)	$\Rightarrow$	$true$;	
(2)	$\Rightarrow$	$\bigcirc(2)$,	
(3)	$\Rightarrow$	$\bigcirc(3)$;	
(4)	$\Rightarrow$	$\bigcirc(4)$;	
(5)	$\Rightarrow$	$\bigcirc(5)$;	
		$\bigcirc(\neg Pr\, U(Open\_Door \vee Close\_Door))$;	(9)
(6)	$\Rightarrow$	$\bigcirc(6)$;	

We introduce for this manipulation the following notation:

$$(1-6) \overset{Open\_Door}{\longrightarrow} (2-6, Pr\, U(Open\_Door \vee Close\_Door));$$

or to the notation:

$$(1-6) \overset{Open\_Door}{\longrightarrow} (2-6, 9).$$

Let us continue for $Move\_Down = true$. We have:

Initial formula		Set of next-state formulae
(1)	$\Rightarrow$	$false$;

For this manipulation we introduce the notation:

$$(1-6) \overset{Move\_Down}{\longrightarrow} (nil).$$

Analogously we get:

$$(1-6) \overset{Stop}{\longrightarrow} (nil);$$

$$(1-6) \overset{Close\_Door}{\longrightarrow} (nil);$$

Let us continue the satisfiability analysis of the formulae (2-6,8). Consider *Move_Up* = *true*. Performing the simplification we have:

Initial formula		Set of next-state formulae
(2)	$\Rightarrow$	$\bigcirc(2)$,
		$\bigcirc(\neg Pr\, U(Move\_Up \vee Stop)$
(3)	$\Rightarrow$	$\bigcirc(3)$;
(4)	$\Rightarrow$	$\bigcirc(4)$;
(5)	$\Rightarrow$	$\bigcirc(5)$;
(6)	$\Rightarrow$	$\bigcirc(6)$;
(8)	$\Rightarrow$	$true$

This corresponds to:

$$(2-6,8) \overset{Move\_Up}{\longrightarrow} (2-6, \neg Pr\, U(Move\_Up \vee Stop))$$

that is to:

$$(2-6,8) \overset{Move\_Up}{\longrightarrow} (2-6,8).$$

Let us simplify now the set of formulae (2-6,8) for *Stop* = *true*. We have:

Initial formula		Set of next-state formulae	
(2)	$\Rightarrow$	$\bigcirc(2)$,	
(3)	$\Rightarrow$	$\bigcirc(3)$;	
(4)	$\Rightarrow$	$\bigcirc(4)$;	
		$\bigcirc(\neg Pr\,U\,Stop \vee Open\_Door \vee$	
		$Move\_Up \vee Move\_Down)$	(10)
(5)	$\Rightarrow$	$\bigcirc(5)$;	
(6)	$\Rightarrow$	$\bigcirc(6)$;	
(8)	$\Rightarrow$	$true$	

This corresponds to:

$$(2{-}6,8) \xrightarrow{Stop} (2{-}6,(Pr\,U\,Stop \vee Open\_Door \vee Move\_Up \vee Move\_Down))$$

that is to:

$$(2-6,8) \xrightarrow{Stop} (2-6,10).$$

Let us continue for $Move\_Down = true$. We have:

Initial formula		Set of next-state formulae
(1)	$\Rightarrow$	$false$;

For this manipulation we introduce like above the notation:

$$(2-6,8) \xrightarrow{Move\_Down} (nil).$$

Analogously we get:

$$(2-6,8) \xrightarrow{Open\_Door} (nil);$$

$$(2-6,8) \xrightarrow{Open\_Door} (nil);$$

The formula manipulations performed in this manner allows us to obtain a transitive closure of next-state sets of formulae in a finite number of steps as presented in Table 1. According to [1] that allows to generate the finite state machine of the specifications (1–7).

Table 1

The satisfiability analysis tableau

Nr		*Move_Up*	*Move_Down*
		1	2
1	$\{1–6\}$	$\{\bigcirc 2 - \bigcirc 6, \bigcirc(\neg Pr\, U\, (Move\_Up \vee Stop))\}$	*nil*
2	$\{2-6, (\neg Pr\, U\, (Move\_Up \vee Stop))\} = \{2-6, 8\}$	$\{\bigcirc 2 - \bigcirc 6, \bigcirc(\neg Pr\, U\, (Move\_Up \vee Stop))\}$	*nil*
3	$\{2-6, (\neg Pr\, U\, (Open\_Door \vee Close\_Door))\} = \{2-6, 9\}$	*nil*	*nil*
4	$\{2-6, (\neg Pr\, U\, (Stop \vee Open\_Door \vee Move\_Up \vee Move\_Down))\} = \{2-6, 10\}$	$\{\bigcirc 2 - \bigcirc 6, \bigcirc(\neg Pr\, U\, (Move\_Up \vee Stop))\}$	$\{\bigcirc 2 - \bigcirc 6, \bigcirc(\neg Pr\, U\, (Move\_Down \vee Stop))\}$
5	$\{2-6, (\neg Pr\, U\, (Move\_Down \vee Stop))\} = \{2-6, 11\}$	*nil*	$\{\bigcirc 2 - \bigcirc 6, \bigcirc(\neg Pr\, U\, (Move\_Down \vee Stop))\}$
6	$\{2-6, (\neg Pr\, U\, (Open\_Door \vee Move\_Up \vee Move\_Down))\} = \{2-6, 12\}$	$\{\bigcirc 2 - \bigcirc 6, \bigcirc(\neg Pr\, U\, (Move\_Up \vee Stop))\}$	$\{\bigcirc 2 - \bigcirc 6, \bigcirc(\neg Pr\, U\, (Move\_Down \vee Stop))\}$

Table 1 (continue)

Nr	*Open_Door*	*Close_Door*	*Stop*
	3	4	5
1	$\{\bigcirc 2-\bigcirc 6, \bigcirc(\neg Pr\, U\, (Open\_Door \vee Close\_Door))\}$	*nil*	*nil*
2	*nil*	*nil*	$\{\bigcirc 2-\bigcirc 6, \bigcirc(\neg Pr\, U\, (Stop \vee Open\_Door \vee Move\_Up \vee Move\_Down))\}$
3	$\{\bigcirc 2-\bigcirc 6, \bigcirc(\neg Pr\, U\, (Open\_Door \vee Close\_Door)))\}$	$\{\bigcirc 2-\bigcirc 6, \bigcirc(\neg Pr\, U\, (Open\_Door \vee Move\_Up \vee Move\_Down))\}$	*nil*
4	$\{\bigcirc 2-\bigcirc 6, \bigcirc(\neg Pr\, U\, Open\_Door \vee Close\_Door)))\}$	*nil*	$\{\bigcirc 2-\bigcirc 6, \bigcirc(\neg Pr\, U\, (Stop \vee Open\_Door \vee Move\_Up \vee Move\_Down))\}$
5	*nil*	*nil*	$\{\bigcirc 2-\bigcirc 6, \bigcirc(\neg Pr\, U\, (Stop \vee Open\_Door \vee Move\_Up \vee Move\_Down))\}$
6	$\{\bigcirc 2-\bigcirc 6, \bigcirc(\neg Pr\, U\, Open\_Door \vee Close\_Door)))\}$	*nil*	*nil*

3.5 Design of the initial (state-based) automaton

The formula manipulations performed in previous section allows us to obtain a transitive closure of next-state sets of formulae in a finite number of steps which according to [1] allows us to generate the state-based automaton model of the elevator control system as depicted in Fig.1.

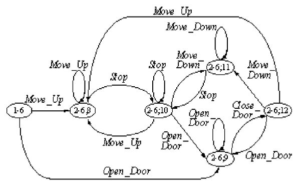

Fig.1. The state-based automaton of the elevator control system

3.6 Design of the action-based automaton of the elevator control system

To implement the synchronisation part of the elevator control system it is necessary to invert the state-based automaton into its dual action-based automaton using a special inversion procedure, as introduced in [4].

This inversion procedure consists of the following steps:

- replace each edge of the initial automaton by a node of the dual automaton keeping for the dual nodes the names of the corresponding edges of the initial automaton;

- connect each pair of dual nodes (a, b) by a directed edge named l from a to b if the corresponding edges (a, b) of the initial automaton form a directed chain via the node l.

Some examples of such inversion are represented in [5].

Using this inversion procedure the action-based automaton of the specifications is obtained as shown in Fig.2.

The action-based automata can be successfully used in the implementation of elevator control system. The action-based automata are in the focus of the implementation method proposed in [5]. This method is an action-oriented method the main idea of which is introduced in [6].

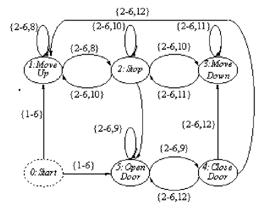

Fig.2. The action-based automaton of elevator control system

3.7 Design of the transition activation conditions of the action-based automata

To implement the activation conditions of the action-based automata transitions it is necessary to introduce for each node of the automaton a propositional variable to indicate the current action which has been executed in the last moment. This variable takes two logical values.

The true value corresponds to the fact that the action associated to this node has been executed last time. Then in the general framework of the execution process a transition between two nodes of the automaton can be activated if the action associated to the source node has been executed and the environment accepts this transition.

The verbal expression like "the environment accepts this transition" should be implemented by a propositional or predicative formula which have to be introduced by the designer.

To implement the transition activation conditions of the elevator control system it is necessary to introduce:

- an array of Boolean variables by one for each node of the action-based automaton to denote the current state of the automaton;

```
Const
   Number_of_actions = 6;
Var
   flag: array [0..Number_of_actions–1] of Boolean;
```

- a predicate $at(N)$ to denote the current floor the elevator passes; this predicate is true if the elevator is within the floor N;

- three predicates *Floor(N), Up_Floor(N), Down_Floor(N)* to denote the state of the internal buttons of the elevator; *Floor(N)* is true if the button N is held; *Up_Floor(N)* is true if there exists at least one button M, such that $M > N$ which is held; *Down_Floor(N)* is true if there exists at least one button M, such that $M > N$ which is held;

- three predicates *Button(N), Up_Button(N), Down_Button(N)* to denote the state of the external buttons of the elevator; *Button(N)* is true if the button N is held; *Up_Button(N)* is true if there exists at least one button M, such that $M > N$ which is held; *Down_Button(N)* is true if there exists at least one button M, such that $M > N$ which is held;

- two propositions *Any_Floor* and *Any_Button* to denote if there exists some requests from internal and external buttons respectively;
- the proposition *Open* to denote the request from the OPEN button;
- the constants: *Home* to denote the initial floor number and *Last* to denote the maximal floor number;
- the timer function *Time()* to return the current time;
- the *Delay* variable to denote the delay time for opened door.

It is important to understand that the array of *Boolean* variables introduced above are dedicated to store the information about the environment like a state vector. This is very important for the correct implementation of the action-based automata in a structural manner.

Using the introduced variables, predicates, function and constants the transition activation conditions can be introduced as presented in Table 2.

3.8 Design of the action activation conditions of the action-based automaton

The design of the action activation conditions of the action-based automaton of the elevator control system can be carried out using the disjunction of the transition activation conditions for each node of the action-based automaton.

An action can be executed if the environment allows this execution.

For example, according to Fig. 2 the action $1 : Move\_Up$ can be executed in the next time if one of the following conditions hold:

- action 0 : *Start* has been executed and the transition $0 \rightarrow 1$ can be activated;
- action $1 : Move\_Up$ has been executed and the transition $1 \rightarrow 1$ can be activated;

- action 2 : *Stop* has been executed and the transition $2 \to 1$ can be activated;
- action 4 : *Close_Door* has been executed and the transition $4 \to 1$ can be activated.

To take into account the state of the environment it is necessary to introduce for each action a logical function the value of which is determined by the current state of the environment. If the current value of this function is true the action can be activated immediately, otherwise if the value of this function is false the action is kept passive till its activation function becomes true.

We call such functions activation conditions due to their special use in system development. To keep the values of this functions it is preferable to save them in some special variables, called activation condition variables, which can be introduced in the manner as follows:

```
Var
  condition: array[0..Number_of_actions-1] of Boolean;
```

Each activation condition variable $condition[i]$, $i := 1..Number\_of\_actions - 1$ is associated to one action i : *Action* and determines if the action can be executed in the current moment. The i : *Action* action can be executed if the corresponding variable $condition[i]$ associated to this action (that is to the corresponding node of the action-based automaton) has a true value.

Having introduced these variables the main problem now consists in determining the predicates (logical functions) to be associated to them. One can suggest different approaches to determine these predicates called also activation conditions. But in order to introduce them in a systematic way we suggest here to use the transition activation condition design above which is very suitable for this aim.

Each transition activation condition specifies the state of the environment when the transition is ready for execution. Thus an action can be activated when it serves as a sink node for a transition which has been executed in the last moment. Since a node can serve as a sink

Table 2

Transition activation conditions

Transition	Activation condition
$0 \to 1$	$flag[0] \wedge at(N) \wedge (N = Home) \wedge \neg Floor(N) \wedge \neg Button(N) \wedge (Up\_Floor(N) \vee Up\_Button(N))$
$0 \to 5$	$flag[0] \wedge at(N) \wedge (N = Home) \wedge ((Floor(N) \vee Button(N))$
$1 \to 1$	$flag[1] \wedge at(N) \wedge \neg(N = Home) \wedge \neg Floor(N) \wedge \neg Button(N) \wedge (Up\_Floor(N) \vee Up\_Button(N))$
$1 \to 2$	$flag[1] \wedge at(N) \wedge (N = Last \vee Floor(N) \vee Button(N))$
$2 \to 1$	$flag[2] \wedge at(N) \wedge \neg Floor(N) \wedge \neg Button(N) \wedge (Up\_Floor(N) \vee Up\_Button(N))$
$2 \to 2$	$flag[2] \wedge at(N) \wedge \neg Any\_Floor \wedge \neg Any\_Button$
$2 \to 3$	$flag[2] \wedge at(N) \wedge \neg Floor(N) \wedge \neg Button(N) \wedge (Down\_Floor(N) \vee Down\_Button(N)) \wedge \neg(Up\_Floor(N) \vee Up\_Button(N))$
$2 \to 5$	$flag[2] \wedge at(N) \wedge (Floor(N) \vee Button(N))$
$3 \to 2$	$flag[3] \wedge at(N) \wedge (N = Home) \vee (Floor(N) \vee Button(N))$
$3 \to 3$	$flag[3] \wedge at(N) \wedge \neg Floor(N) \wedge \neg Button(N) \wedge (Down\_Floor(N) \vee Down\_Button(N))$
$4 \to 1$	$flag[4] \wedge at(N) \wedge \neg Floor(N) \wedge \neg Button(N) \wedge (Up\_Floor(N) \vee Up\_Button(N))$
$4 \to 3$	$flag[4] \wedge at(N) \wedge \neg Floor(N) \wedge \neg Button(N) \wedge (Down\_Floor(N) \vee Down\_Button(N)) \wedge \neg(Up\_Floor(N) \vee Up\_Button(N))$
$4 \to 5$	$flag[4] \wedge at(N) \wedge (Floor(N) \vee Button(N))$
$5 \to 4$	$flag[5] \wedge at(N) \wedge \neg(Timer() < Delay)$
$5 \to 5$	$flag[5] \wedge at(N) \wedge (Timer() < Delay) \wedge Open$

node for a lot of transitions simultaneously, the activation condition associated to this node should be represented by the disjunction of the activation conditions of all the transitions incident to this node.

Using this idea the activation conditions for all the actions of the action-based automaton of the elevator control system can be introduced as follows in Table 3.

Table 3

Actions activation conditions

Actions	Transitions	Activation conditions
$condition[0]$		$flag[0]$
$condition[1]$	$0 \rightarrow 1$	$(flag[0] \wedge at(N) \wedge (N = Home) \wedge \neg Floor(N) \wedge \neg Button(N) \wedge (Up\_Floor(N) \vee Up\_Button(N))) \vee$
	$1 \rightarrow 1$	$(flag[1] \wedge at(N) \wedge \neg(N = Home) \wedge \neg Floor(N) \wedge \neg Button(N) \wedge (Up\_Floor(N) \vee Up\_Button(N))) \vee$
	$2 \rightarrow 1$	$(flag[2] \wedge at(N) \wedge \neg Floor(N) \wedge \neg Button(N) \wedge (Up\_Floor(N) \vee Up\_Button(N))) \vee$
	$4 \rightarrow 1$	$(flag[4] \wedge at(N) \wedge \neg Floor(N) \wedge \neg Button(N) \wedge (Up\_Floor(N) \vee Up\_Button(N)))$
$condition[2]$	$1 \rightarrow 2$	$(flag[1] \wedge at(N) \wedge (N = Last \vee Floor(N) \vee Button(N))) \vee$
	$2 \rightarrow 2$	$(flag[2] \wedge at(N) \wedge \neg Any\_Floor \wedge \neg Any\_Button) \vee$
	$3 \rightarrow 2$	$(flag[3] \wedge at(N) \wedge (N = Home) \vee Floor(N) \vee Button(N)))$
$condition[3]$	$2 \rightarrow 3$	$(flag[2] \wedge at(N) \wedge \neg Floor(N) \wedge \neg Button(N) \wedge (Down\_Floor(N) \vee Down\_Button(N)) \wedge$

Actions	Transitions	Activation conditions
	$3 \to 3$	$\neg(Up\_Floor(N) \vee Up\_Bottom(N)))\vee$ $(flag[3] \wedge at(N) \wedge \neg Floor(N)\wedge$ $\neg Button(N) \wedge ((Down\_Floor(N)\vee$ $Down\_Button(N)))\vee$
	$4 \to 3$	$(flag[4] \wedge at(N) \wedge \neg Floor(N)\wedge$ $\neg Button(N) \wedge (Down\_Floor(N)\vee$ $Down\_Button(N))\wedge$ $\neg(Up\_Floor(N) \vee Up\_Bottom(N)))$
$condition[4]$	$5 \to 4$	$flag[5] \wedge at(N)\wedge$ $\neg(Timer() < Delay)$
$condition[5]$	$0 \to 5$	$(flag[0] \wedge at(N) \wedge (N = Home)\wedge$ $((Floor(N) \vee Button(N)))\vee$
	$2 \to 5$	$(flag[2] \wedge at(N)\wedge$ $(Floor(N) \vee Button(N)))\vee$
	$4 \to 5$	$(flag[4] \wedge at(N)\wedge$ $(Floor(N) \vee Button(N)))\vee$
	$5 \to 5$	$(flag[5] \wedge at(N)\wedge$ $(Timer() < Delay) \wedge Open)$

3.9 Design of the control structure of the system driver

Using the action activation conditions the control structure of the elevator control system can be designed. The control structure can be represented as a loop structure nesting a set of *if* statements. Each *if* statement tests a *condition*$[i]$ condition.

If the condition is true the corresponding action of the elevator control system is executed. If the *condition*$[i]$ formulae are built up correctly only one *condition*$[i]$ expression must by true at a time. Thus the control structure may be represented as follows:

```
Begin
  While true do
    if condition[0] then Start();
    if condition[1] then Move_Up();
    if condition[2] then Stop();
    if condition[3] then Move_Down();
    if condition[4] then Close_Door();
    if condition[5] then Open_Door();
  end;
end;
```

This structure represents the driver of the elevator control system. The driver is dedicated to manage the execution of the elevator system actions.

3.10 Design of the functional structure of the elevator control system

The functional structure of the elevator control system is dedicated to develop the general functional architecture of the designed system. The functional model of a system is required to structure the implementation of the system actions and of the system driver. To represent these models some key constructions of the specification language IKARS are used [6].

The upper level of the functional model of the elevator control system can be represented in the framework of the IKARS language as follows:

```
System Elevator
  OWN
    Action Do_Start(),
    Action Do_Move_Up(),
    Action Do_Stop(),
    Action Do_Move_Down(),
    Action Do_Close_Door(),
    Action Do_Open_Door()
  End OWN;


  Action Do_Start()
    OWN
```

```
      Action Start(),
      Action Update_Flags_Start()
    End OWN
  End Do_Start;


  Action Do_Move_Up()
    OWN
      Action Move_Up(),
      Action Update_Flags_Move_Up()
    End OWN
  End Do_Move_Up;


  . . .
```

<the functional specifications of the actions *Do_Stop(), Do_Move_Down()* and *Do_Close_Door()* are omitted>

```
  . . .


  Action Do_Open_Door()
    OWN
      Action Do_Open_Door(),
      Action Update_Flags_Open_Door()
    End OWN
  End Do_Open_Door;
End Elevator.
```

Each action in the specification like *Do_Action()* corresponds to an action like *Action()* but differs from it by the fact that the action *Do_Action()* is a compound action consisting of the *Action()* action and of the *Update_Flags_Action()*. Each flag updating action is dedicated to set and reset the action flags so that the actions will be executed according to the partial order defined by the action based automaton (Fig.2). That is a *Do_Action()* action consists of two subactions: *Action()* which is the action itself, and the flag updating action *Update_flags_Action()*. The action *Update_flags_Action()* is dedicated to change the action flag values after each action execution to modify properly the state of the environment. As mentioned above the information about the environment like a state vector is very important for the correct implementation of the action-based automata.

According to the functional model the new compound actions have been introduced. That is why the control structure introduced above must be rewritten now as:

```
Begin
  While true do
    if condition[0] then Do_Start();
    if condition[1] then Do_Move_Up();
    if condition[2] then Do_Stop();
    if condition[3] then Do_Move_Down();
    if condition[4] then Do_Close_Door();
    if condition[5] then Do_Open_Door();
  end;
end;
```

3.11 Implementation of the elevator system actions and of the system control driver

To implement the actions of the functional model of the elevator system actions we introduce for each action *Do_Action()* a procedure. In a Pascal-like manner this procedure can be written as follows:

```
Procedure Do_Action();
   Const action_number=<the action name>

   Procedure Action();
```

```
    Begin
        <application based implementation of the Action() action>
  End;

  Procedure Update_flags_Action();
    Var
      i: integer;
    Begin
      for i:= 0 to Number_of_actions-1 do
        flag[i]:=false;
      flag [action_number]:=true;
    End;

  Begin
    Action();
    Update_flags_Action();
  End;
```

This is only the skeleton of the *Do_Action()* implementation.

The elevator control system actions can be implemented in this way. It is strongly recommended to implement these actions as procedures in a library unit.

```
Unit Elevator_Actions;

Procedure Do_Start();
  Const action_number =0

  Procedure Start();
    Begin
        <application based implementation of the Start() action>
  End;

  Procedure Update_flags_Start();
    Var
      i: integer;
    Begin
      for i:= 0 to Number_of_actions-1 do
        flag[i]:=false;
      flag [action_number]:=true;
    End;

  Begin
    Action();
    Update_flags_Start();
  End;

  ...
```

<the implementation of the actions *Do_Move_Up(), Do_Stop()*, *Do_Move_Down()* and *Do_Close_Door()* are omitted>

```
  ...
  Procedure Do_Open_Door();
    Const action_number = 5

    Procedure Open_Door();
      Begin
        <application based implementation of the Open_Door() action>
    End;

    Procedure Update_flags_Open_Door();
      Var
        i: integer;
      Begin
        for i:= 0 to Number_of_actions-1 do
          flag[i]:=false;
        flag[action_number]:=true;
      End;

  Begin
    Action();
    Update_flags_Open_Door();
  End;
  end;
```

The unit introduced above can be used in the implementation of the elevator control system driver.

```
Program Elevator_Driver;
uses Elevator_Actions;
Const
  Number_of_actions = 6;
Var
  flag: array [0..Number_of_actions-1] of Boolean;
  condition:array[0..Number_of_actions-1] of Boolean;

Begin
Init_flags;
  While true do
  Begin
    Update_conditions;
    if condition[0] then Do_Start();
    if condition[1] then Do_Move_Up();
    if condition[2] then Do_Stop();
    if condition[3] then Do_Move_Down();
    if condition[4] then Do_Close_Door();
    if condition[5] then Do_Open_Door();
  end;
end.
```

4 Conclusions

Temporal logic formalism has proved its applicability in specification and design of distributed and concurrent systems. Some practical results in this area the reader can find in [1,2,5,7,8].

However all known applications of temporal logic in distributed system design deal mainly with the general framework of the design process.

The case study of the elevator control system shows how the presented method can be used in the design of real time system. This case study completes the picture of the methods introduced in [5,6] and covers all the design steps: from temporal logic specifications to the implementation of the system code.

References

[1] Z.Manna, P.Wolper. Synthesis of Communicating processes from temporal logic specifications, ACM TOPLAS, 6, 1984, pp.68–93.

[2] A.Pnueli. Applications of Temporal Logic to the Specification and Verification of reactive systems: a survey of current trends, in: LNCS 224, 1986, pp.510–584.

[3] E.M.Clarke, E.A.Emerson. Design and synthesis of synchronisation skeletons using Branching Time Temporal logic, in: Proc. Workshop on Logics of Programs, LNCS, Vol. 131, 1981, pp.52–71.

[4] A.Ursu, S.Zaporojan. Design of the Action-based Automata of the Distributed Systems Protocols on the basis of Temporal Logic Specifications, in: Proceedings of the 5th Symposium on Automatic Control and Computer Science 1995, October 21-27, 1995, Iasi, Romania, Vol.1, pp.423–428.

[5] G.C.Gruita. Design and validation of a simplified communication protocol using temporal logic specifications, License Diploma Thesis, UTM, Chisinau, Moldova, 1995.

[6] IKARS, The work group report, Electrotechnical University, Sankt-Petersburg, Russia, 1990.

[7] A.Ursu, V.Dubenetsky, V.Besliu. Development and application of temporal logic specifications in distributed systems design, in: Proceedings of the 9th Romanian Symposium on Computer Science ROSYCS'93, November 12-13, 1993, Iasi, Romania, pp.526–544.

[8] A.Ursu, V.Besliu, S.Zaporojan, V.Dubenetsky. Design and analysis of temporal logic specifications of distributed systems communication protocols, in: Proceedings of the International Conference on Technical Informatics ConTI'94, November, 16-19, 1994, Timisoara, Romania, Vol.4, pp.1–10.

A.Ursu, G.Gruita,
Information Technology Department,
Technical University of Moldova,
Stefan cel Mare 168,
2012, Chisinau, Republic of Moldova,
phone: (+373-2) 497018, 497014;
fax: (+373-2) 247114

Received 3 January, 1996

V.Dubenetsky
Information processing and control systems Department,
State electrotechnical University of Sankt-Petersburg,
str.Popov, 5,
197022, Sankt-Petersburg, Russia,
phone: (+842) 2347321